Benefits and Challenges of Model-based Software Engineering: Lessons Learned based on Qualitative and Quantitative Findings

Katerina Goseva-Popstojanova
West Virginia University
Morgantown, WV, USA
katerina.goseva@mail.wvu.edu

Thomas Kyanko
Berkshire Grey
Pittsburgh, PA, USA
tkyanko@gmail.com

Noble Nkwocha
NASA Independent Verification & Validation Facility
Fairmont, WV, USA
noble.n.nkwocha@nasa.gov

***Abstract*—Even though Model-based Software Engineering (MBSwE) techniques and Autogenerated Code (AGC) have been increasingly used to produce complex software systems, there is only anecdotal knowledge about the state-of-the-practice. Furthermore, there is a lack of empirical studies that explore the potential quality improvements due to the use of these techniques. This paper presents in-depth qualitative findings about development and Software Assurance (SWA) practices and detailed quantitative analysis of software bug reports of a NASA mission that used MBSwE and AGC. The mission's flight software is a combination of handwritten code and AGC developed by two different approaches: one based on state chart models (AGC-M) and another on specification dictionaries (AGC-D). The empirical analysis of fault proneness is based on 380 closed bug reports created by software developers. Our main findings include: (1) MBSwE and AGC provide some benefits, but also impose challenges. (2) SWA done only at a model level is not sufficient. AGC code should also be tested and the models and AGC should always be kept in-sync. AGC must not be changed manually. (3) Fixes made to address an individual bug report were spread both across multiple modules and across multiple files. On average, for each bug report 1.4 modules, that is, 3.4 files were fixed. (4) Most bug reports led to changes in more than one type of file. The majority of changes to auto-generated source code files were made in conjunction to changes in either file with state chart models or XML files derived from dictionaries. (5) For newly developed files, AGC-M and handwritten code were of similar quality, while AGC-D files were the least fault prone.**

I. Introduction

Model-based Software Engineering (MBSwE) techniques and auto-generated code (AGC) are increasingly used to produce complex software, including software for safety- and mission-critical systems. MBSwE and AGC promise many benefits, including higher productivity and improved quality. Despite these widely assumed benefits, the actual situation is not simple and clear cut. Rather, MBSwE and AGC offer opportunities, but also impose challenges.

Although model-based development and AGC have been used for some time, there is only anecdotal knowledge about different forms of AGC and the spectrum of used SWA techniques. Furthermore, there is a lack of empirical studies that explore the potential quality improvements due to the use of MBSwE and AGC and, therefore, our understanding of the impact of these technologies is limited.

The work presented in this paper is based on using as a case study a NASA mission that used MBSwE and AGC approaches. The mission's flight software is a combination of handwritten code and AGC developed by two different MBSwE approaches, one using state charts and another using specification dictionaries that contain command and telemetry data. Approximately 18% of the mission code was auto-generated using these two methods. 56% of the total source code was newly developed and the rest was either re-engineered or reused. The empirical analysis of software fault proneness is based on 380 closed bug reports created by software developers. Specifically, our work is focused on the following research questions:

RQ1: What approaches to MBSwE and AGC were used by the NASA mission?

RQ2: What development and SWA practices were used for models and auto-generated code?

RQ3: Did MBSwE and AGC approaches help improving software quality?

Unlike the extensive research on software faults and failures in general (e.g., [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]), the empirical analysis of software developed using MBSwE is still very scarce, limited to only several works [15], [16], [17]. This paper aims to fill that gap by providing in-depth qualitative findings related to the use of two MBSwE approaches and the SWA practices, as well as detailed quantitative analysis of the software faults and fixes, at both module and file level, broken down by development approach (i.e., AGC-M, AGC-D, and handwritten) and heritage (i.e., newly developed, re-engineered, and reused).

The rest of the paper is organized as follows. The related works are described in section II. Section III describes the MBSwE approaches and SWA practice used by the NASA mission, i.e., addresses RQ1 and RQ2. Detailed analyses of the developers' bug reports is presented in section IV, thus addressing RQ3. Section V presents the summary of the main findings. The threats to validity are described in section VI and the paper is concluded in section VII.

II. Related Work

Significant prior research work exists on characteristics of software faults and failures for systems that did not use MBSwE. We briefly discuss these works to allow for comparison of our findings with the general trends (e.g., distribution of faults across software modules).

Fenton and Ohlsson used a large telecommunication application to study a range of software engineering hypotheses, including the Pareto principle of distribution of faults and failures [1]. Several research works by Hamill and Goseva-Popstojanova were based on data extracted from a large NASA mission and were focused on characterizing and quantifying the relationships among faults, failures, and fixes [2], [3], [4]. Another study based on space mission data [5], conducted by Grottke et al., analyzed the anomalies from the flight software of 18 JPL space missions and classified them as Bohrbugs and Mandelbugs. Alonso et al., in a follow up work, analyzed the mitigation associated with the Bohrbugs and Mandelbugs [6]. In a closely related work, based on the analysis of bug reports of four open-source software systems, Cotroneo et al. [7] classified the software bugs as Bohrbugs, non-aging-related Mandelbugs, and aging-related bugs.

The information extracted from bug tracking systems of open source applications was used for research with different goals. Bug reports of 12 open source programs were explored by Duraes and Madeira [8] with a goal to establish the fault representativeness for software fault injection experiments. Xia et al. classified the faults of four open source applications into several categories [9].

Other papers explored software faults for different application domains. The study by Gashi et al. explored the bug reports of four off-the-shelf SQL servers [10]. Maji et al. explored the manifestation of failures in Android and Symbian [11]. Ocariza et al. studied the error messages printed by JavaScript as it executes in popular websites [12] and explored JavaScript faults based on bug reports extracted from 12 bug repositories [13]. Frattini et al. analyzed the bug reports from the open source cloud platform Apache Virtual Computing Lab [14].

MBSwE is believed to provide advantages over traditional methods, for example in the embedded systems domain [18]. NASA studied how MBSwE can be used for mission-critical software development [19]. Other studies were focused on particular aspects of MBSwE, such as for example verification of models [20], [21]. Several studies explored the benefits and limitations of MBSwE and AGC using survey and interviews based approaches [18], [22], [23], [24]. Surveys and interviews are opinion-based studies that provide valuable information, but also have some limitations.

However, even though MBSwE and AGC have been around for a while, not many research works were focused on collecting empirical data and quantifying the impact of these technologies to productivity and quality. A study by Mohagheghi and Dehlen [25], which was based on review of 25 papers that dealt with model-based development and auto-generated code, reported both productivity gains and losses, and some quality improvements, but these results were mainly based on small scale studies and typically did not include quantitative data. Only one paper (out of 25) provided quantitative data related to software quality and reported that the two auto-coded features of a Motorola's network element had lower fault densities than the three handwritten features [15]. However, that study did not account for the fact that auto-generated code is typically larger in size, which may have caused a misleading reduction in fault density (i.e., faults per KLOC).

Another quantitative study by Nugroho and Chaudron compared the quality (measured as fault density) of Java classes developed using UML modeling (MC) with the classes that were not modeled (NMC), based on data extracted from an industrial application [16]. The results showed that the fault densities of MC and NMC were similar. Then, the authors performed a pairwise sampling and ran statistical test on a random sample of 96 classes, which showed smaller fault density of MC classes. A follow up study [17], based on the same industrial application, extended the work presented in [16] by considering, in addition to fault density, the effect of the UML modeling on the fault resolution time (i.e., fix time).

Zhang and Patel described the use of agile model-based software development for a real-time telecommunication system [26] and stated that "the quality of automatically generated code in terms of defects density is significantly higher than manual code". However, no actual data or any baseline were presented [26]. The work by Lucredio et al. provided quantitative analysis of the MBSwE impact on software reuse based on analysis of three exploratory studies [27]. The results showed that complex technical domains have more to gain from MBSwE. However, the effect of MBSwE on software quality was not considered in [27].

The work presented in this paper is based on using a NASA mission as a case study, which is an observational, evidence-based approach that is complementary to surveys and interviews. The only other works that attempted to quantify software quality were based on a Motorola's telecommunication application [15] and a healthcare system from Netherlands [16], [17]. None of these works included reused software modules. In comparison with these related works, our study provides in-depth qualitative findings related to the use of two MBSwE approaches and the SWA practices, as well as detailed quantitative analysis of the software faults and fixes, at both module and file level, broken down by development approach (i.e., AGC-M, AGC-D, and handwritten) and heritage (i.e., newly developed, re-engineered, and reused).

III. MBSwE AND SWA PRACTICE

In this paper we use as a case study a NASA flight mission, which was broken into 67 modules. The mission's flight software was developed using three approaches:

1) Handwritten code resulting from traditional development process,
2) MBSwE approach 1: auto-generated code from statechart models created in MagicDraw, and

3) MBSwE approach 2: auto-generated code based on input from specification dictionaries.

The process of auto-generating code from statechart models is presented in Figure 1. MagicDraw files (*.mdxml*) contained statechart models and were used as input to an in-house developed autocoder running in the Quantum Framework to generate C code, referred to as *AGC-M* in this paper. The parts that were auto-generated using statechart models were typically the logic portions of that module.

The process used to auto-generate code from specification dictionaries is shown in Figure 2. The dictionaries, which contain command and telemetry data, were exported to an XML format (*.xml* files), and then passed to two in-house developed autocoders, which produced C source code files, referred to as *AGC-D* in this paper. Code developed using MBSwE approach 2 was mainly structural, with no behavioral details. The upper part of Figure 2 (from right to left) shows a Verification and Validation (V&V) approach used by the mission. Using an automated event extractor, XML files (*AG-XML*) with the command and telemetry data were auto-generated from the handwritten code and compared with the dictionary .xml files. If there was a problem, changes were made and the process was repeated.

Out of 67 modules, only two were fully auto-generated: one was 100% AGC-M and the other was 100% AGC-D. The remaining modules with AGC contained handwritten code. Additionally, some modules contained both AGC-M and AGC-D. At module level these modules were classified as AGC-M because, in most cases, they had more AGC-M than AGC-D code. Each individual file was either 100% handwritten, 100% AGC-M, or 100% AGC-D. The mission used a consistent file naming convention, which allowed us to accurately classify the files based on the specific development method used.

Note that the three autocoders were developed in-house with the same rigorousness as the flight software. We are not aware of any faults found in AGC-M or AGC-D flight software files that were due to faults in the autocoders.

The developers were allowed to choose whether to handwrite the code or use the MBSwE approach 1. A total of seven modules included AGC-M. In general, developers felt that some specifics of the flight software could not be accounted for by the used MBSwE tools. Several challenges were related to the development and use of models. First, extra effort was needed to develop the models, which may be overlooked or underestimated. Further, it was an open question when to stop including details in the model. Developers also noted that models tend to obscure the lower level details present in the AGC-M.

As expected, there were challenges related to the readability of AGC-M because it is not intuitive and lacks comments. Developers experience showed that the readability of AGC-M improved once they acquired knowledge and understood the patterns used by MagicDraw.

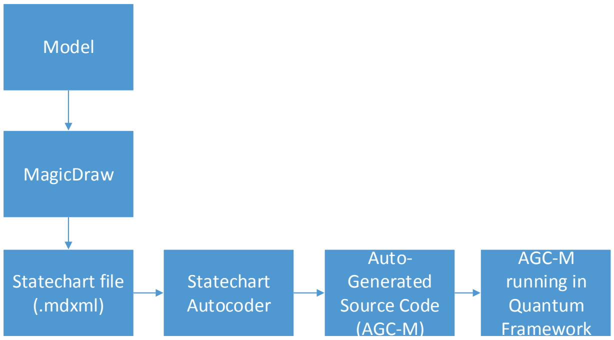

Model
MagicDraw
Statechart file (.mdxml)
Statechart Autocoder
Auto-Generated Source Code (AGC-M)
AGC-M running in Quantum Framework

Figure 1: MBSwE approach 1: Auto-code generation using statechart models (AGC-M)

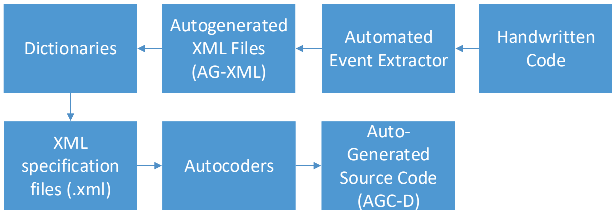

Dictionaries
Autogenerated XML Files (AG-XML)
Automated Event Extractor
Handwritten Code
XML specification files (.xml)
Autocoders
Auto-Generated Source Code (AGC-D)

Figure 2: MBSwE approach 2: Auto-code generation using dictionaries (AGC-D)

The software assurance practices of MBSwE based on statechart models are described next. The developers conducted design reviews using less detailed state charts, but did not find testing statechart models to be very useful. The fact that off-nominal cases were not available in the modeling phase was a drawback. (They were incorporated into the models later in the lifecycle.) Based on the experience with previous missions, both models and AGC-M were committed in the version control system; once committed, AGC-M was not changed manually. Furthermore, the models and AGC-M were always kept in-sync and AGC-M was tested using the same process as handwritten code.

Mission developers experienced several challenges related to the maintenance of the AGC-M. One challenge was due to the fact that the mission's flight software is a combination of handwritten and auto-generated code. In addition, AGC-M may be hard to maintain onboard because the modeling suite, and thus the models, are not available on the onboard computer console.

For the MBSwE approach 2, which resulted in AGC-D, the dictionaries were used as requirements documents and the code auto-generation ensured that the requirements were in-sync with the code. In general, the development team found the approach based on dictionaries much more useful than MBSwE approach 1.

Development of the mission's flight software took advantage of modules developed for previous missions, as well as modules developed as part of a NASA generalized framework. Specifically, a module was considered reused

Table I: Distribution of module sizes in LLOC, by development approach and heritage

	AGC-M			AGC-D			Handwritten		
LLOC	New	ReE	ReU	New	ReE	ReU	New	ReE	ReU
<1000	0	1	0	11	7	1	4	1	9
1000-1999	1	0	0	3	5	3	0	0	1
2000-2999	0	0	0	4	1	0	0	0	1
3000-3999	1	1	0	1	3	0	0	0	1
4000-4999	0	0	0	0	2	0	1	0	0
>=5000	3	0	0	1	0	0	0	0	0
Subtotal	5	2	0	20	18	4	5	1	12
Total	7			42			18		

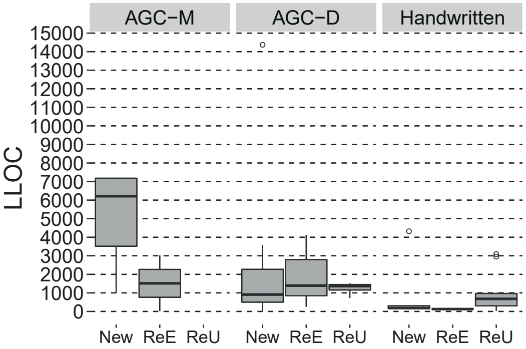


Figure 3: Boxplots of module sizes

(ReU) if less then 10% of its code was changed, re-engineered (ReE) if from 10% to 50% of the code was changed, or new (New) if more that 50% of the code was changed. We refer to 'reused', 're-engineered', and 'newly developed' designations collectively as module *heritage*.

The distributions of module sizes in Logical (i.e., non-comment) Lines of Code (LLOC), broken down by development approach and heritage, are given in Table I and the boxplots are shown in Figure 3. (Note that two AGC-D modules, one with <1000 LLOC and another with 1000-1999 LLOC, are not shown in Table I and Figure 3 as their heritage was unknown.) As can be seen from Figure 3, the five newly developed modules that have AGC-M are the largest in size, from around 1,000 to over 7,000 thousands LLOC. The smallest AGC-M module was re-engineered and is the only module that has 100% AGC-M. The other six modules (five new and one re-engineered) have from 27% to 42% auto-generated code. The twenty newly generated AGC-D modules have sizes from 15 to 14,371 LLOC. The smallest module was 100% AGC-D. The other modules have from around 2% to 48% of AGC-D. The modules consisting only of handwritten code are the smallest across all heritages. Specifically, the five newly developed, fully handwritten modules have sizes from 156 to 4,319 LLOC.

IV. Analysis of bug reports and associated fixes

To avoid potential confusion we start with defining the terms failure, fault (i.e., bug), and fix [3]. A *failure* is a departure of the system or system component behavior from its required behavior. A *fault* is an accidental condition, which if encountered, may cause the system or system component to fail to perform as required. In this paper, the terms *fault* and *bug* are used interchangeably. A *fix* refers to changes made to correct the fault(s) associated with bug reports. Fixes associated with an individual bug report may be located in one or more files, one or more modules, and made to one or more software artifacts. The most similar to the term fix appears to be the term repair, which is defined as correction of faults that have resulted from errors in external design, internal design, or code [28]. Note that repair is part of fault removal [29].

A. Dataset description

The dataset of developers' issues contained a total of 544 issues, stored in the JIRA issue tracking system. 474 issues were marked "Defect" (i.e., bug), and of these 439 issues were marked "Closed". The closed bug reports spanned a period of more than three years, from early 2012 to mid 2015. The bug reports included fields on how bugs were found, descriptions, lists of modules affected, and files changed to resolve each bug report. The analysis presented in this section is based on 380 closed bug reports (out of 439) which contained information about files that were fixed to address the bug report. Because the mission used a consistent file naming convention we were able to associate files with modules and determine if they were auto-generated, and if so, how they were generated. We wrote scripts to extract file names from the developers' issue tracking system and classified them by the development approach and heritage (i.e., reuse) using a combination of automated and manual classification.

B. Analysis of bug reports

First, we studied the development and SWA activities that led to detection of software bugs. As can be seen from Table II, the majority of bugs were found during Flight Software Integration Testing (FIT), which is focused on testing different modules together. System Integration and Testing

Table II: Number of bug reports created during different development and V&V activities

Activity	# of bug reports
FSW Integration Testing (FIT)	225
System Integration and Testing (SIT)	81
Unit Testing	23
BIT	18
Assembly, Test, and Launch Operations	15
Development	11
Other	3
Problem Investigation	3
Operations	1
Total	380

Table III: Distribution of the number of modules fixed together to address individual bug report. 'N/A' refers to bug reports that listed fixes to project-wide files.

# of modules fixed together	# of bug reports
1	286
2	66
3	11
4	7
5	3
9	2
11	1
N/A	4
Total	380

(SIT) was the second most successful activity; it involved performing software testing with simulated hardware.

In the following two subsections we study the fixes, i.e., changes made to fix fault(s) associated with the 380 closed bug reports, first at module level and then at file level.

C. Analysis of fixes at module level

In general, more that one module may be fixed as a result of each bug report. As shown in Table III, 25% of bug reports, which is a considerable percentage, led to changes in two or more modules. The average number of modules fixed together for addressing an individual bug report was 1.4. However, as can be seen in Table III the distribution is skewed, with some bug reports leading to fixes across large number of modules (up to 11). These results are consistent with our results from previous analysis of bug reports of another NASA mission, that have shown that 18% of bug reports led to fixes in more than one module [3].

For the analysis at a module level, if a bug report led to changes in only one module, even if several files in that module were changed, we counted it as one fix. If a bug report led to changes in multiple modules, we counted one fix for each module.

Our analysis showed that the distribution of number of fixes across modules followed the Pareto principle, with relatively few modules containing the majority of fixes. Specifically, 55% of all fixes were associated with only 20% of modules. This finding agrees with other works [1], [2], [30], [31] that have consistently found that majority of faults (and fixes) reside in around 20% of the lines of code, files, or packages, depending on the unit.

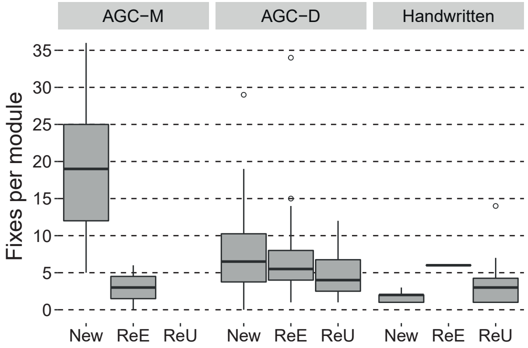


Figure 4: Boxplots of number of fixes per module

Next, we studied the fault proneness at module level, broken down by development approach and heritage. As can be seen from Figure 4 and Table IV, among newly developed modules, AGC-M modules had significantly higher mean and median fixes per module than AGC-D and handwritten modules. On the other side, re-engineered AGC-M modules had smaller mean and median number of fixes per module than AGC-D and handwritten modules. Reused handwritten modules had smaller mean and median number of fixes per module than the AGC-D modules. Within each development approach, as one may expect, new modules had the highest mean and median fixes per module, followed by re-engineered and reused modules. Note that the only exception is the result for handwritten, re-engineered code, which was based on only one handwritten, re-engineered module.

When the number of fixes was normalized by size (measured in KLLOC), the mean and median number of fixes per KLLOC had similar values for newly developed modules across all development approaches, with AGC-M modules having slightly lower values (see Table IV). Note that this result, to some extent may be due to the fact that the module sizes of the AGC-M modules were typically much larger than the sizes of AGC-D and handwritten modules, thus leading to smaller fix densities. With respect to the effect of the heritage (i.e., reuse) within each development approach, we observed the same trend as in case of the mean and median fixes per module – newly developed modules had the highest fix density, followed by re-engineered and reused modules (again with an exception of the single re-engineered, handwritten module).

D. Analysis of fixes at file level

For the analysis at file level, all changes made to a given file due to a single bug report were counted as a single fix. If a bug report led to changes in multiple files, it was counted once for each file being fixed. Note that for the file level analysis, the development approach was unique and known for each individual file (based on the naming convention).

Table IV: Basic statistics of number of fixes at module level, by development approach and heritage

Dev approach	Heritage	# of Modules	Total LLOC	Mean LLOC per module	Total fixes	Median fixes per module	Mean fixes per module	Median Fixes / KLLOC	Mean Fixes / KLLOC
AGC-M	New	5	25,119	5,024	97	19.00	19.40	4.99	4.22
	ReE	2	3,029	1,514	6	3.00	3.00	0.99	0.99
	ReU	0	0	0	0	0.00	0.00	0.00	0.00
AGC-D	New	20	37,669	1,883	164	6.50	8.20	6.60	6.23
	ReE	18	32,940	1,830	136	5.50	7.56	3.75	5.12
	ReU	4	4,969	1,242	21	4.00	5.25	2.70	4.01
Handwritten	New	5	5,118	1,024	9	2.00	1.80	6.33	6.86
	ReE	1	126	126	6	6.00	6.00	47.62	47.62
	ReU	12	11,427	952	46	3.00	3.83	4.07	7.45

Table V: Number of times different file types were fixed

File type	# of times files were fixed
Test files	1,936
Handwritten code	1,226
AGC-M	110
Other	87
.xml	60
AGC-D	47
.mdxml	23
Total	3,592

For the 380 bug reports, a total of 1,273 files were fixed 3,592 times. In other words, on average, 3.4 files were fixed to address an individual bug report. The breakdown of the number of fixes for each file type is given in Table V.

As described in subsection IV-A, for our case study, each bug report was linked to the files that have been fixed, which allowed us to associate the faults to the changes made to fix these faults and carry on in-depth analysis. Based on the combinations of file types changed together for addressing a single bug report, we identified the following five classes:

1) *Handwritten files*, i.e., fixes made to source files that were not auto-generated.
2) *MagicDraw (.mdxml) files, fixed with or without AGC-M files and/or handwritten files.*
3) *Dictionary XML files (.xml) fixed with or without AGC-D file and/or handwritten files.*
4) Combinations of classes 2 and 3.
5) *Test files and/or other miscellaneous file types*, such as compilation configuration scripts.

For each of the five classes, Table VI presents the different combination of file types that were fixed together. Based on Table VI, we made the following observations:

- Most bug reports led to changes in more than one type of files. This result may partially be due to the fact that most bug reports in our dataset were reported during integration testing (i.e., FIT and SIT).
- Most fixes required changes to handwritten source files (class 1). This was expected as handwritten code was the largest portion of the code base (see Table VII).
- Less bug reports led to changes in AGC files (classes 2, 3, and 4). The majority of changes to AGC files were made in conjunction to changes in either MagicDraw models (.mdxml) or XML files derived from dictionaries (.xml). This finding confirmed that the AGC and the models were kept in-sync.
- Changes to .xml files, in combination with other file types (classes 3 and 4) were the second most common class of changes. .xml and AG-XML files were most often changed in combination with handwritten files. This suggests that changes were made to the dictionaries and handwritten code as part of the same fixes.

Next, we analyze the basic statistics of fixes at file level for source code files (AGC-M, AGC-D, and Handwritten). As can be seen in Table VII, file sizes (in LLOC) across the three development approaches are much more balanced than module sizes. In case of newly developed files, the median number of fixes per file of AGC-M is slightly higher than for handwritten code (i.e., 3 vs. 2 fixes per file). The mean number of fixes per file of AGC-M and handwritten code have comparable values (i.e., 3.21 and 3.24, respectively). The AGC-D files are the least fault prone, with zero median fixes per file and an order of magnitude smaller mean number of fixes per file. Reused AGC-D files had higher mean number of fixes per file than new and re-engineered AGC-D files, likely because the system and context were not exactly the same.

Since file sizes are comparable, the fix density (i.e., number of fixes per KLLOC) at file level makes more sense than at module level. Based on the results shown in Table VII, it appears that newly developed AGC-M files have significantly higher median and mean number of fixes per KLLOC than newly developed handwritten files (i.e., 30.77 and 139.72 vs. 19.80 and 73.01 fixes per KLLOC). The AGC-D files again had zero median number of fixes per KLLOC, and the lowest mean number of fixes per KLLOC (3.97). We again observed that reused AGC-D files had higher mean number of fixes per KLLOC than both new and reused AGC-D files.

Figures 5a, 5b, and 5c present the boxplots of file sizes for all files, .c files, and .h files, respectively. As can be seen from Figures 5b and 5c .h files are significantly smaller

Table VI: Combinations of file types fixed together for addressing individual bug reports

Class	ID	Filetypes fixed	# of bug reports	Total
1	1	Handwritten code, Test files	146	248
	2	Handwritten code	90	
	3	Handwritten code, Other, Test files	10	
	4	Handwritten code, Other	2	
2	5	.mdxml, AGC-M, Handwritten, Other, Test files	6	15
	6	.mdxml, AGC-M, Handwritten, Test files	3	
	7	.mdxml, AGC-M	2	
	8	.mdxml, AGC-M, Handwritten	2	
	9	.mdxml, Handwritten	1	
	10	.mdxml, AGC-M, Test files	1	
3	11	AG-XML, Handwritten, Test files	35	100
	12	AG-XML, Handwritten	18	
	13	.xml, AG-XML, Handwritten, Test files	14	
	14	.xml, AG-XML, Handwritten	9	
	15	AG-XML, Handwritten, Other, Test files	4	
	16	.xml, AG-XML, AGC-D, Handwritten, Other, Test files	2	
	17	.xml, AGC-D, Handwritten	2	
	18	.xml, AGC-D, Handwritten, Test files	2	
	19	.xml, AG-XML, AGC-D, Handwritten, Test files	2	
	20	.xml, AGC-D, Handwritten, Other, Test files	2	
	21	.xml, AG-XML, Handwritten, Other, Test files	2	
	22	.xml, AGC-D, Handwritten, Other	1	
	23	.xml, AG-XML, AGC-D, Handwritten	1	
	24	AG-XML	1	
	25	AG-XML, AGC-D, Handwritten, Other, Test files	1	
	26	.xml, AGC-D	1	
	27	.xml, Handwritten, Test files	1	
	28	.xml	1	
	29	AGC-D	1	
4	30	.mdxml, AG-XML, AGC-M, Handwritten, Test files	3	13
	31	.mdxml, AG-XML, AGC-M, Handwritten, Other, Test files	3	
	32	.xml, AGC-M, Handwritten, Test files	2	
	33	.mdxml, AG-XML, AGC-M, Handwritten, Other	1	
	34	.xml, AG-XML, AGC-M, Handwritten	1	
	35	.xml, AGC-M, Handwritten	1	
	36	.xml, AGC-M, Test files	1	
	37	.xml, AGC-M	1	
5	38	Test files	2	4
	39	Other, Test files	1	
	40	Other	1	
		Total Bug Reports		380

than .c files, for all development approaches and heritages. Therefore, we decided to conduct the analysis separately for .c and .h files. Figures 5a-5c also show that handwritten code had wider distribution of file sizes than AGC.

Tables VIII and IX show the basic statistics for .c and .h files broken down by development approach and heritage. Figures 6a, 6b, and 6c present the boxplots of fixes per file for all files (i.e., .c and .h together), .c, and .h files, respectively. Figures 7a, 7b, and 7c present the boxplots of fixes per KLLOC for all files, .c, and .h files, respectively.

As can be seen from Table VIII and Figures 6a, 6b, and 6c, both median and mean number of fixes per file are slightly higher for .c files than for .h files. Furthermore, newly developed AGC-M .c and .h files have higher median number of fixes per file and similar mean number of fixes per file with newly developed handwritten files. Similarly as for all files together, AGC-D .c and .h files were the least fault prone, with zero median and an order of magnitude lower mean fixes per file. (Due to the skewness of the data, in these cases medians are better representation of central tendency than means.)

When it comes to the fix density (i.e., the number of fixes per KLLOC), as can be seen from Table IX and Figures 7a, 7b, and 7c, we observed that .h files have significantly higher fix density than .c files, for all statistics. This is mostly due to the fact that .h files have an order of magnitude smaller file sizes that .c files (see Table VIII and Figures 5b and 5c). For newly developed files, AGC-M .c files have slightly lower median fix density than newly developed handwritten files, while AGC-M .h files have significantly higher median fix density than handwritten files. The same is true for the mean fix density. The mean values, however, are higher than the median values as they are more sensitive to outliers.

V. Summary of the results and lessons learned

Table X summarizes the qualitative and quantitative findings, organized by the research questions RQ1 - RQ3.

Table VII: Basic statistics of the number of fixes at file level, by development approach and heritage

Development approach	Heritage	# of files	Total LLOC	Mean LLOC per file	Total fixes	Median fixes per file	Mean fixes per file	Median fixes / KLLOC	Mean fixes KLLOC
AGC-M	New	38	5,855	154	122	3.00	3.21	30.77	139.72
	ReE	6	297	50	0	0.00	0.00	0.00	0.00
	ReU	0	0	0	0	0.00	0.00	0.00	0.00
AGC-D	New	88	9,866	112	21	0.00	0.24	0.00	3.97
	ReE	68	5,055	74	8	0.00	0.12	0.00	2.07
	ReU	14	459	33	5	0.00	0.36	0.00	12.00
Handwritten	New	207	52,185	252	671	2.00	3.24	19.80	73.01
	ReE	132	30,743	233	400	2.00	3.03	18.93	74.88
	ReU	74	15,937	215	136	1.00	1.84	14.65	44.65

Table VIII: Basic statistics of the number of fixes at file level, by development approach and heritage, split by .c and .h files

Dev approach	Heritage	# of files		Total LLOC		Mean LLOC per file		Total fixes		Median fixes per file		Mean fixes per file	
		(.c)	(.h)	(.c)	(.h)	(.c)	(.h)	(.c)	(.h)	(.c)	(.h)	(.c)	(.h)
AGC-M	New	15	23	5,135	720	342	31	56	66	4.00	3.00	3.73	2.87
	ReE	2	4	231	66	116	16	0	0	0.00	0.00	0.00	0.00
	ReU	0	0	0	0	0	0	0	0	0.00	0.00	0.00	0.00
AGC-D	New	44	44	6,551	3,315	149	75	11	10	0.00	0.00	0.25	0.23
	ReE	34	34	3,286	1,769	97	52	5	3	0.00	0.00	0.15	0.09
	ReU	7	7	285	174	41	25	3	2	0.00	0.00	0.43	0.29
Handwritten	New	128	79	45,317	6,868	354	87	455	216	2.00	2.00	3.55	2.73
	ReE	87	45	26,573	4,170	305	93	265	135	2.00	2.00	3.05	3.00
	ReU	48	26	14,320	1,617	298	62	94	42	1.00	1.00	1.96	1.62

Table IX: Basic statistics of the number of fixes per KLLOC at file level, split by .c and .h files

Dev approach	Heritage	Median fixes / KLLOC		Mean fixes / KLLOC	
		(.c)	(.h)	(.c)	(.h)
AGC-M	New	11.83	100.00	14.79	221.19
	ReE	0.00	0.00	0.00	0.00
	ReU	0.00	0.00	0.00	0.00
AGC-D	New	0.00	0.00	2.97	4.97
	ReE	0.00	0.00	1.98	2.17
	ReU	0.00	0.00	10.21	13.79
Handwritten	New	14.87	52.63	22.44	154.93
	ReE	12.39	37.66	28.63	164.28
	ReU	8.89	38.69	20.68	88.91

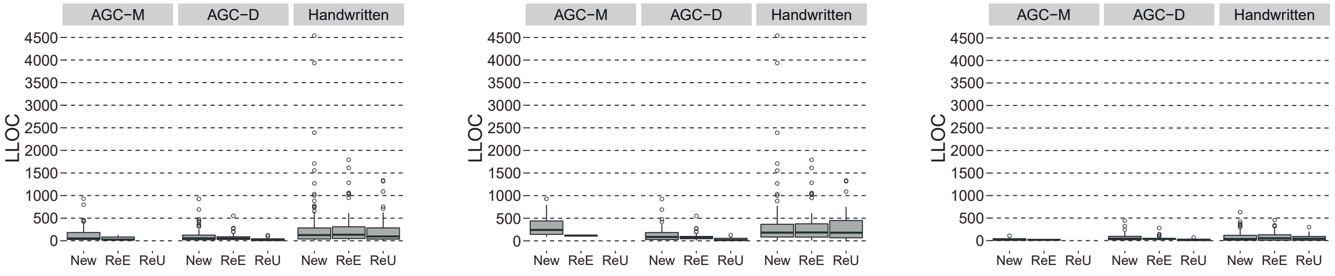
VI. Threats to Validity

Construct validity is concerned with whether we are measuring what we intend to measure. One threat to construct validity is the use of imprecise terminology, which makes comparisons across studies difficult. To address this threat, we provided the definitions of the terms faults, failures, and fixes. Other threats to construct validity may be due to data values and their interpretation. We worked closely with the NASA personnel to ensure that we thoroughly understood the data and did not misinterpret any values. To further address construct validity, we analyzed the fault proneness at both module and file level, and found that the analysis at file level was more relevant because (1) file sizes were more balanced than module sizes and (2) only two (out of 67) modules were fully auto-generated.

Internal validity threats are concerned with unknown influences that may affect independent variables and their interaction with dependent variables. Data quality is one of the major concerns to the internal validity. To ensure the data quality, we had multiple interactions with the NASA personnel, manually investigated the dataset used in this paper, and conducted sanity checks.

Conclusion validity threats impact the ability to draw correct conclusions. One threat to conclusion validity is related to data sample sizes. The work presented in this paper is based on a fairly large dataset consisting of 380 closed bug reports. Other threats to conclusion validity are related to the way the results are reported. We reported both the mean and median values, noting that for skewed datasets median values are more representative. Finally, it is important to mention that our work was done in close collaboration with NASA personnel. Developers' feedback on the initial results was obtained at in-person meetings. Subsequently, they provided feedback on the progress through regular e-mail communication. This close collaboration contributed to the quality of the research and ensured accurate interpretation of the results.

External validity is concerned with the ability to generalize the results. This work is based on one case study and we cannot claim that the results would be valid across other software systems. Generalizations are usually based on multiple empirical studies that have replicated the same phenomenon under different conditions [30], [32]. Whenever possible, we compared our results with prior works focused on software fault proneness. However, there is a lack of empirical works focused on specifics of software fault proneness for systems that were developed using MBSwE. Therefore, the external

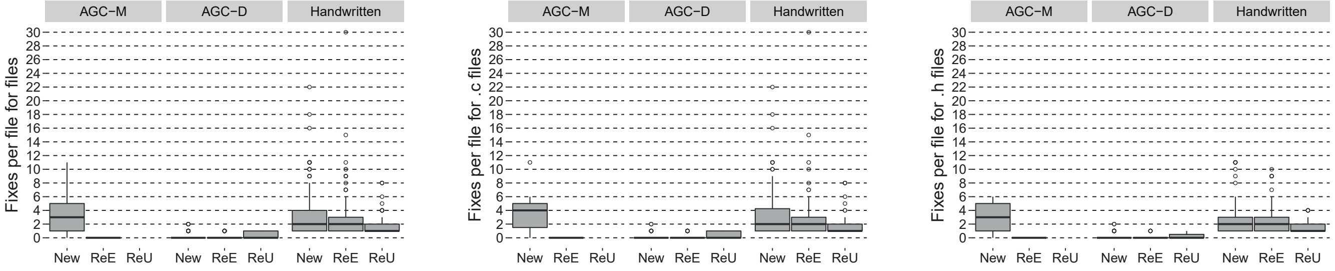


(a) .c and .h files together

(b) .c files

(c) .h files

Figure 5: Boxplots of file sizes

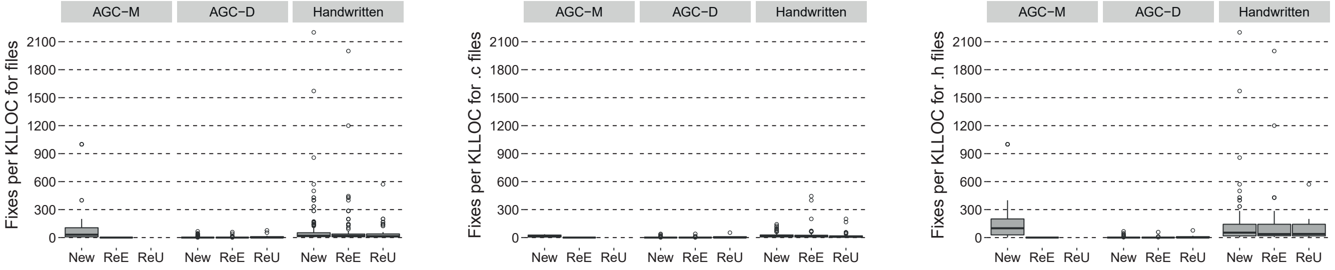


(a) .c and .h files together

(b) .c files

(c) .h files

Figure 6: Boxplots of fixes per file



(a) .c and .h files together

(b) .c files

(c) .h files

Figure 7: Boxplots of fixes per KLLOC

validity remains to be established by future studies that will use other software products as case studies.

VII. Conclusion

In this paper we presented a study of the approaches to MBSwE and AGC used by a NASA mission, the SWA practices used by the mission, and the empirical analysis of the software fault proneness. The mission's flight software is a combination of handwritten code and AGC developed by two different approaches, one using state charts and the other using specification dictionaries. The qualitative and quantitative analysis presented in this paper led to observation of some expected patterns, as well as to discovery of new findings. Several interesting findings and their potential implications are as follows.

Significant number of bug reports led to fixes that were spread not only across multiple files, but also across multiple modules. On average, individual bug reports were associated with 1.4 modules, that is, 3.4 files. These results show the generalizability of previous findings [3] and have several implications. Thus, they indicate the danger of injecting (i.e., seeding) only simple semantic faults, localized to a single module, which affects areas such as fault-based testing and mutation testing. Furthermore, the assumption that each failure is caused by a single component, which is a widely used in component-based software reliability models, appears to be an oversimplification of the real phenomenon.

MBSwE and AGC provide some benefits, but also impose challenges, such as inability of the tools to account for some specifics of flight software, extra effort needed to develop the models, low readability of the AGC, lack of off-nominal cases in the modeling phase, and difficulty of maintaining the AGC-M onboard due to limited hardware resources to run the whole suite of MBSwE tools. These challenges may be easily overlooked and/or underestimated when planning the software development based on MBSwE.

Table X: Summary of main findings

	Topic	Findings	Section
RQ1	Approaches to MBSwE & AGC	Two MBSwE approaches: AGC-M from state chart models created in MagicDraw and AGC-D using specification dictionaries as input.	III
	Challenges: tools	Some specifics of the flight software could not be accounted for by the used MBSwE tools.	III
	Challenges: models	Extra effort was needed to develop the models; it was an open question when to stop including details in the model; models tended to obscure the lower level details in the AGC-M.	
	Challenges: AGC-M readability	Not readable because the code is not intuitive and lacks comments. Readability improved once the patterns used by MagicDraw were known and understood.	
	Perceived value	The development team found the AGC approach based on dictionaries (AGC-D) much more useful that the approach based on state charts (AGC-M).	
RQ2	SWA	Developers conducted design reviews using less detailed state charts, but did not find testing statechart models to be very useful. Both models and AGC-M were committed in the version control system; once committed, AGC-M was not changed manually. The models and AGC-M were always kept in-sync and AGC-M was tested using the same process as handwritten code.	III
	Challenges	The off-nominal cases were not available in the modeling phase. The mission's flight software is a combination of handwritten and AGC. AGC-M may be hard to maintain onboard while using a console on which the whole modeling suite, and thus the models, are not available.	
RQ3	Bug reports	Majority of bugs were found during FIT and SIT (i.e., 59% and 21%, respectively).	IV-B
	Fixes at module level	25% of bug reports led to changes in two or more modules. On average, 1.4 modules were fixed to address an individual bug report, but the distribution was skewed with some bug reports leading to as many as 9 or 11 modules being fixed.	IV-C
		55% of all fixes were associated with only 20% of modules	
		Among newly developed modules, AGC-M modules had significantly higher median number of fixes per module (i.e., 19.00) than AGC-D and handwritten modules (with 6.50 and 2.00, respectively). The same is true for the mean number of fixes per module. Re-engineered AGC-M modules had smaller mean and median number of fixes per module than AGC-D and handwritten modules. Reused handwritten modules had smaller mean and median number of fixes per module than the AGC-D modules.	
		Within each development approach the new modules had the highest mean and median fixes per module, followed by re-engineered and reused modules.	
	Fixes at file level	On average, 3.4 files were fixed to address an individual bug report. Most bug reports led to changes in more than one type of files. Most fixes required changes to handwritten source files, which was expected because they dominated the code base. The majority of changes to auto-generated source code files were made in conjunction to changes in either MagicDraw models (.mdxml) or XML files derived from dictionaries (.xml), which confirms that auto-generated code and models were kept in-sync.	IV-D.
		For newly developed files, the median number of fixes per file of AGC-M is slightly higher than for handwritten code (i.e., 3 vs. 2 fixes per file). The mean number of fixes per file of AGC-M and handwritten code were comparable (i.e., 3.21 vs. 3.24). The AGC-D files are the least fault prone, with zero median fixes per file and an order of magnitude smaller mean number of fixes per file. Reused AGC-D files had higher mean number of fixes per file than new and re-engineered files.	
		Both median and mean number of fixes per file are slightly higher for .c files than for .h files. Newly developed AGC-M .c and .h files have higher median number of fixes per file and similar mean number of fixes per file with newly developed handwritten files. AGC-D .c and .h files are the least fault prone, with zero median and an order of magnitude lower mean fixes per file.	

MBSwE does not mean that SWA should be done only on models. In case of this NASA mission, the AGC-M was tested using the same process as handwritten code and the models and AGC-M were always kept in-sync. Based on the lessons learned from model-based development of flight software in previous missions, this mission committed both the models and AGC-M in the version control system. Once committed, AGC-M was not changed manually.

MBSwE and AGC do not always lead to better software quality. Basically, not all AGC is equal. In our case study, for newly developed files, AGC-M and handwritten code were of similar quality (i.e., had comparable mean values of the number of fixes per file, while AGC-M files had slightly higher median values). These results may be partially due to the fact that some bugs, such as those related to requirements, are likely to affect the software development regardless of the development approach. Additional challenges, such as the use of new hardware and unavailability of the off-nominal cases during the modeling phase, could also explain these results. Unlike AGC-M files, AGC-D files were the least fault prone, which is likely due to the fact that AGC-D was mainly structural code, with no behavioral details.

It appears that learning more about the state-of-the-practice of MBSwE and AGC and the fault proneness trends of the software systems developed using these technologies is necessary to help understanding the benefits and challenges and further improve the state-of-the-art and practice of model-based software development.

Acknowledgments

This work was funded by the NASA Software Assurance Research Program (SARP). The authors thank the mission developers and IV&V analysts for their support of our work. Thomas Kyanko's contributions were made while he was a graduate student at West Virginia University.

References

[1] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, Aug 2000.

[2] M. Hamill and K. Goseva-Popstojanova, "Common trends in software fault and failure data," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 484–496, July 2009.

[3] ——, "Exploring the missing link: An empirical study of software fixes," *Software Testing, Verification and Reliability*, vol. 24, no. 8, pp. 684–705, 2014.

[4] ——, "Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system," *Software Quality Journal*, vol. 23, no. 2, pp. 229–265, 2015.

[5] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *40th IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, June 2010, pp. 447–456.

[6] J. Alonso, M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault repairs and mitigations in space mission system software," in *43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2013, pp. 1–8.

[7] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault triggers in open-source software: An experience report," in *24th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 178–187.

[8] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions of Software Engineering*, vol. 32, no. 11, pp. 849–867, Nov. 2006.

[9] X. Xia, X. Zhou, D. Lo, and X. Zhao, "An empirical study of bugs in software build systems," in *13th International Conference on Quality Software*, July 2013, pp. 200–203.

[10] I. Gashi, P. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 280–294, 2007.

[11] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile OSes: A case study with Android and Symbian," in *21st IEEE International Symposium on Software Reliability Engineering*, Nov 2010, pp. 249–258.

[12] F. S. Ocariza, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *22nd IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2011, pp. 100–109.

[13] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side JavaScript bugs," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 55–64.

[14] F. Frattini, R. Ghosh, M. Cinque, A. Rindos, and K. S. Trivedi, "Analysis of bugs in Apache Virtual Computing Lab," in *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2013, pp. 1–6.

[15] T. Weigert and F. Weil, "Practical experiences in using model-driven engineering to develop trustworthy computing systems," in *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, 2006, pp. 208–215.

[16] A. Nugroho and M. R. Chaudron, "Evaluating the impact of UML modeling on software quality: An industrial case study," in *12th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2009, pp. 181–195.

[17] ——, "The impact of UML modeling on defect density and defect resolution time in a proprietary system," *Empirical Softw. Engg.*, vol. 19, no. 4, pp. 926–954, Aug. 2014.

[18] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: An industrial survey on the state-of-practice," *Software System Modeling*, vol. 17, no. 1, pp. 91–113, Feb. 2018.

[19] E. Benowitz, K. Clark, and G. Watney, "Auto-coding UML statecharts for flight software," in *2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT 2006)*, 2006, pp. 413–417.

[20] R. Rogersten, H. Xu, N. Ozay, U. Topcu, and R. M.Murray, "An aircraft electric power testbed for validating automatically synthesized reactive control protocols," in *16th International Conference on Hybrid systems: Computation and Control (HSCC'13)*, 2013, p. 89.

[21] E. Denney and B. Fischer, "A verification-driven approach to traceability and documentation for auto-generated mathematical software," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2009, pp. 560–564.

[22] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Science of Computer Programming*, vol. 89, pp. 144–161, 2014, part B.

[23] K. Goseva-Popstojanova, T. Kahsai, M. Knudson, T. Kyanko, N. Nkwocha, and J. Schumann, "Survey on model-based software engineering and auto-generated code," NASA, Technical Report, Oct 2016.

[24] D. Akdur, V. Garousi, and O. Demirors, "A survey on modeling and model-driven engineering practices in the embedded software industry," *Journal of Systems Architecture*, vol. 91, pp. 62–82, 10 2018.

[25] P. Mohagheghi and V. Dehlen, "Where is the proof? A review of experiences from applying MDE in industry," in *Model Driven Architecture – Foundations and Applications*, I. Schieferdecker and A. Hartman, Eds. Springer Berlin Heidelberg, 2008, pp. 432–443.

[26] Y. Zhang and S. Patel, "Agile model-driven development in practice," *IEEE Software*, vol. 28, no. 2, pp. 84–91, 2011.

[27] D. Lucredio, E. Santana de Almeida, and R. Fortes, "An investigation on the impact of MDE on software reuse," in *6th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, 2012, pp. 101–110.

[28] *Systems and Software Engineering Vocabulary*, 2nd ed., 24765-2017 - ISO/IEC/IEEE International Standard, 2017.

[29] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[30] C. Andersson and P. Runeson, "A replicated quantitative analysis of fault distributions in complex software systems," *IEEE Transactions of Software Engineering*, vol. 33, no. 5, pp. 273–286, 2007.

[31] T. Devine, K. Goseva-Popstojanova, S. Krishnan, and R. R. Lutz, "Assessment and cross-product prediction of software product line quality: Accounting for reuse across products, over multiple releases," *Automated Software Engineering*, pp. 1–50, 2014.

[32] R. K. Yin, *Case Study Research: Design and Methods*. Thousand Oaks, CA: SAGE Publication Ltd, 2014.