International Scholarly Research Network
ISRN Software Engineering
Volume 2012, Article ID 162305, 11 pages
doi:10.5402/2012/162305

Research Article

Lessons Learnt from Gauging Software Metrics of Cabin Software in a Commercial Airliner

Stefan Burger[1] and Oliver Hummel[2]

[1] *TCC4 System Integration—Communication, EADS Innovation Works, 81663 Munich, Germany*
[2] *Software Engineering Group, University of Mannheim, 68131 Mannheim, Germany*

Correspondence should be addressed to Stefan Burger, stefan.burger@eads.net
and Oliver Hummel, hummel@informatik.uni-mannheim.de

Received 14 June 2012; Accepted 4 September 2012

Academic Editors: C. Calero and R. J. Walker

In order to achieve high safety standards, avionic software needs to be developed with very high requirements regarding safety, reliability, and determinism as well as real-time constraints, which are often in conflict with the development of maintainable software systems. Nevertheless, the ability to change the software of an airplane is of increasing importance, since it consists of a multitude of partially hardware-specific subsystems which may need replacement during a plane's lifespan of typically numerous decades. Thus, as a first step towards optimizing maintainability of avionic software we have benchmarked the cabin software of a commercial airliner with common software metrics. Such a benchmarking of avionic software contributes valuable insights into the current practice of developing critical software and the application of software metrics in this context. We conclude from the obtained results that it is important to pay more attention to long-term maintainability of aviation software. Additionally we have derived some initial recommendations for the development of future avionic software systems.

1. Introduction

It is a (relaxing) fact that requirements for avionic software in terms of safety, reliability, and predictability need to adhere to very rigorous standards and hence software used within avionic systems must be certified before it is allowed to fly. The avionic industry accounts for these aspects by developing highly optimized, application specific solutions with an enormous amount of validation and verification effort [1, 2]. In order to allow easier certifiability, guidelines for the development of avionic software (such as DO-178B [3]) usually constrain the use of programming constructs and strive for manageable complexity. Due to the enormous recertification effort, avionic software systems in the past were normally rarely changed once they were certified and deployed in a certain airplane type. However, recent technology advances, especially in the field of entertainment and communication systems, have changed the needs and desires of airline customers considerably: a twenty-first century passenger demands functionalities such as video on demand, video games, and Internet access during the flight. Consequently, the demands for airlines and plane manufacturers are rapidly changing and require faster maintenance times and shorter product update cycles in order to quickly (in terms of avionic software development this may still mean years) react on the new capabilities of electronic consumer devices and their software.

Additionally, airplanes themselves keep changing as more and more electronic devices and sensors have found their way into modern aviation systems so that each new airplane generation is further increasing the amount of electronic equipment on board. As software is becoming an ever more important part of all electronic devices in an airplane, manufacturers are forced to deal with an increasing desire to change the software deployed on board, while in practice they have found that the effort for such an undertaking is currently in no healthy relation to its benefit [4]. Thus, on the one hand, avionic software solutions remain currently well suited for their specific purpose and meet the required high reliability and safety standards; on the other hand, however, despite (or because of) the tremendous effort invested,

aviation software and related systems seem to have become a real "maintenance challenge" for airplane manufacturers. As a consequence, the interest of the avionics industry in applying modern software engineering techniques for increasing maintenance has been constantly rising in recent years.

However, assessing the maintainability of existing software and deriving recommendations for increasing it is still an area of ongoing research: traditionally, software metrics [5] have been considered as helpful for identifying weakly maintainable (i.e., complex) system parts, but unfortunately, to date only few of the relations between software metrics and software quality have been actually understood (such as the relation between metrics and bugs, e.g., uncovered by Nagappan et al. [6] and Schröter et al. [7]). Only recently, the open source movement has made it easier for researchers to intensively gauge numerous real-life desktop software systems (such as Mozilla [8]) with common software metrics and to create at least a set of common values (i.e., initial benchmarks) for the applied software metrics. Nevertheless, there is still no clear consensus on the implications of this exercise for software maintenance and the situation for proprietary embedded and avionic software is even worse: comprehensive case studies are barely published and hence researchers and developers lack common baselines against which they could compare newly built systems in order to gain a better insight of the mechanisms at work here. In other words, at the time being, most statements on the actual maintainability of avionic software are based on "hearsay."

In order to shed some light on this important aspect, we have performed a systematic assessment of a recent cabin management and monitoring system (from hereon we will abbreviate it with CMMS) taken from a current Airbus plane type (which we are unfortunately not allowed to disclose). Due to the numerous tasks (such as passenger announcement, light, and climate control, etc.) it needs to manage, a CMMS is a complex piece [9] of software (in our case, comprising more than 1.5 MLOC, mainly written in the C programming language) to which numerous contractors need to contribute. Since the rigorous safety regulations for aviation software also apply for it, it is a particularly interesting subject for a maintainability study for which we have gauged it with various common software metrics such as those from Halstead [10] or McCabe [11]. In this paper we do not only present the results of our efforts, but also a comparison with recently published benchmarks from other domains and an analysis of correlations discovered between various metrics. Before we explain this related work in more detail in Section 3, we briefly introduce some of the idiosyncrasies influencing avionic software development in Section 2. Section 4 describes the results of our analysis and briefly presents the findings for our case study from a maintainability perspective. Section 5 draws some more advanced conclusions on correlations amongst metrics before we conclude our paper in Section 6.

1.1. Research Questions and Contributions. In order to give a more detailed overview of this work, we briefly summarize its main contributions in the following: the focus of this case study presented here was on analyzing maintainability and changeability of a recent avionic software system, with the focus of making suggestions about improvements for current development standards and designs. Our work extends the existing body of knowledge with the following long neglected contributions: first, it presents the results of measuring a large piece of embedded software with a comprehensive set of common software metrics. To our knowledge it is the largest data set presented in the avionics domain so far that makes concrete metric results available. Second, it investigates whether and how far current software metrics can actually predict software quality and maintainability. Finally, it analyzes and uncovers correlations between various well-known software metrics.

To guide our study beyond the mere measuring of metrics, we have formulated the following research hypotheses under the very general assumption that higher metrics values usually indicate a lower maintainability.

Hyphothesis 1. Due to the (presumed) enormous domain complexity of the system under study, we expect the metrics values measured for it to be significantly higher than for the systems in the previous similar studies. This includes the average values (H1a) as well as the maximum metrics values (H1b).

Hyphothesis 2. We expect all metrics that have been proposed for predictor models in the literature to be independent from each other, that is, there is no correlation between them detectable.

2. Software Development in the Avionic Industry

Since the requirements and the environment for developing software in avionic industry differ considerably from widely known "mainstream" software development, for example, in a business context, we briefly describe some of its important fundamentals in this section. First and foremost, avionics systems have high requirements on reliability since a malfunction can result in injuries or in the worst case even in the loss of lives. This is obviously not only the case for cockpit and flight control software, but with cabin software as well. Consider, for example, if the passenger announcement system failed the ability of the flight attendants for addressing the potentially around 800 passengers of an Airbus A380 with potentially safety-relevant information would be lost.

In order to avoid or at least minimize the effects of such malfunctions in avionic systems, every airplane needs to adhere to guidelines issued by the Federal Avionic Administration in the US (FAA, cf., e.g., Chilenski and Miller [12]) and the European Avionic Safety Agency (EASA), respectively. Both governmental agencies maintain several guidelines, such as DO-178B (software [3]) and DO-254 (hardware [13]) describing the mandatory steps for developing and testing hardware and software in airplanes in order to get them certified. In case of software, amongst other regulations, the former standard demands requirements traceability and special testing procedures depending on the

criticality of a piece of code, which is classified into five safety levels. Software on the highest level, for instance, requires full modified condition/decision test coverage (MC/DC [12]). A further requirement of DO-178B regulates that tests on high safety levels have to be independently assessed and audited. Furthermore, different safety levels need to be isolated from each other and critical system parts need to be implemented redundantly, that is, hardware devices may need backup power supplies, additional communication lines, or even completely redundant "stand-by" backup systems.

3. Background and Related Work

When it comes to software maintainability, the ISO/IEC Standard 9126 [14] is still widely cited in the literature as it provides a simple model dividing maintainability into four central subcharactericstics. In order to provide optimal comparability with previous works (discussed in Section 3.3) we have also used this model and not its successor ISO 25010 [15] that has recently been finalized with revised maintainability subcharacteristics. ISO 9126 lists a number of quality characteristics [16] (such as usability and reliability) that are supposed to predict the quality of software as a product and also defines that "*Maintainability is the capability of software to be modified.*" Maintainability itself is then subdivided into four further characteristics: the first is *analyzability*, which constitutes the effort required to understand and analyze the program code (i.e., its complexity). The second is *changeability*, which describes the time needed to adapt the program to external change requests. The third maintainability characteristic is *stability*: it describes the probability that changes or fixes to a program might introduce new faults. The fourth is *testability*, characterizing the ease of testing and hence certifying a program.

As we explain in the next subsection, literature (e.g., Spinellis et al. [17]) attributes various software metrics to each of these characteristics, in order to define a commonly usable maintainability model. Even some commercially successful tools (such as IBM Rational Logiscope (http://www-01.ibm.com/software/awdtools/logiscope/, accessed April 2012)) define such a model, but nevertheless, the empirical and theoretical underpinning is thin. Consequently, it is hard to answer concrete questions when the maintainability of a system is to be investigated, which also limits the applicability of structured measurement approaches such as GQM [18] in this context. The few published examples that actually attempted such an approach (such as [17]) must hence be criticized for simply mirroring the categories of ISO 9126 [14] without breaking them up into independently and concretely measurable subgoals as usually required by GQM. In order to avoid such a mistake, we simply benchmarked our case study with common metrics, as explained in the following subsection. We are convinced that it is important to have such benchmarks available from various domains in order to allow detailed analysis of the underlying metrics before actual prediction models can be derived.

3.1. Software Metrics. Over the years, a large variety of software metrics has been published in the literature including "traditional" metrics for imperative programming languages (the metrics of Halstead [10] and McCabe [11] are probably best known in this context) and object-oriented metrics (e.g., by Chidamber and Kemerer [19]). However, these metrics alone are only of limited value since they are not correlated with the above-mentioned quality characteristics of a software system. Nevertheless, in recent years, numerous studies have attempted to establish such a correlation and have certainly been most successful in the area of predicting faults [6]. In terms of maintainability, comparable results are scarce, however, and hard to find due to the difficulties in establishing the actual maintainability of a system without expensive empirical experiments [20]. In order to nonetheless contribute a useful piece of work that might instigate other researchers to carry out and publish similar studies, we have gauged several well-known software metrics for our system in order to compare them with the previously published results from other domains.

Table 1 illustrates the metrics we have selected and presents an allocation to the maintainability characteristics mainly adopted from ISO 9126 [14]. As described there, one metric can be linked to more than one characteristic, for example, the maximal nesting is linked to analyzability and changeability.

Various of the presented metrics have already been proposed in the 1970s when Halstead [10] has defined his widely known set of so-called software science metrics (which in turn has also been under considerable discussion since then). All Halstead metrics are derived from the distinct number of operators (such as <, &, |, etc.) and operands (variable names, literals, etc.) and the total number of used operators and operands (we have used Scientific Tool Inc.'s *Understand* for measuring the metrics). The *program length* metric is the sum of the total number of operators and operands; thus, it is defined completely independently from the code formatting. The so-called program *Vocabulary* is the sum of the number of unique operators and operands in a piece of code. Furthermore, Halstead has defined the *difficulty* (D) of a program, which is supposed to represent the effort a developer needs to understand code. Probably the best-known Halstead metric is the so-called *Volume*. It describes the information content of a program in mathematical bits. The *effort* (E) for a program is supposed to represent the time and work, that are needed to implement or understand a program. Halstead's *Vocabulary frequency* indicates the hardness of a software code to be maintained [10].

Another software metric, called *the average size of statements*, is intended to give an indication for the complexity of code on the level of statements. The *Fan-In/Fan-Out* metric was defined by Henry and Kafura [21] and measures all incoming or outgoing dependencies (i.e., the functions called and the global variables set in case of the used tool) for a function or a module. Clearly, the higher the Fan-In of a specific module, the greater is the dependency with other modules (low cohesion). The lack of cohesion of functions measures the occurrence of global variables that is supposed to decrease the cohesion of functions and modules.

Table 1: Relationship between maintainability characteristics and software metrics (adapted from [6]).

Maintainability characteristic	Commonly related software metrics
Analyzability	Vocabulary size, difficulty, volume, effort, average size of statements, fan-out, cyclomatic complexity, maximal nesting
Changeability	Vocabulary frequency, unconditional jumps, maximal nesting, lack of cohesion, coupling
Stability	Number of entry points, number of exit points, fan-in
Testability	Test coverage

One of the most common metrics, *Cyclomatic Complexity*, was also used in our study. According to McCabe [11], it is defined as the number of predicate nodes (i.e., branch statements) in the flow graph of a piece of code (i.e., usually a function) plus one. McCabe defines a cyclomatic complexity greater than 10 as being too high, other authors suggest a threshold of 20 [17]. Furthermore, we measured the *maximum nesting level*, which describes the depth of nested predicate nodes (i.e. if, while, for, and switch). The deeper the nesting within the code, the higher is the presumable level of complexity. *Coupling* [22] is another metric for counting dependencies between modules on the level of files, that is, if a file uses two distinct files from the same module, this is counted as coupling of two. Another indication for the difficulty to implement change requests is the *number of GOTOs*, since "GOTO is considered harmful" [23] for structured programming and maintainability.

3.2. Maintainability Prediction Models. As discussed before, it is still not clear which concrete metrics form good predictors for which quality characteristic. Even worse, the applicability of those metrics varies with the programming paradigm and their expressiveness is sometimes at least questionable [24] and has been issue of an ongoing debate in the software engineering community. Hence, to our knowledge the literature contains no commonly accepted prediction model that would allow assessing the maintainability of a software system, since the models exemplarily discussed in the following are usually neither very well understood nor evaluated.

A prime example is the so-called maintainability index proposed by Coleman et al. [25] in the 1990s, which has never been systematically validated and is hence subject to a lot of criticism. Spinellis et al. [17] presented one of the rare general software quality models (cf. Table 3) that also aimed at including maintainability and recommends a number of object-oriented metrics (such as weighted methods per class [19]) to be measured for this purpose. However, beyond its unclear validity it is not applicable in the context of our case study that has been developed in a procedural language.

Heitlager et al. [26] have defined another practical model for predicting maintainability based on software metrics based on a set of about one hundred smaller programs from several industrial applications. Their model is based on a minimal set of metrics that is supposed to predict software maintainability characteristics as defined by ISO 9126 [14]. Examples for the metrics used include lines of code, code clones, cyclomatic complexity, or the unit test coverage. In order to create the model, they have analyzed the distribution of metrics values statistically and have derived four classes of values for each metric. These classes are measured for each function in the code base of a system and indicate low, moderate, high, or very high risk from a maintainability perspective. Depending on the risk distribution over the whole system, a star rating ranging from two to five star, indicating the overall maintainability of the system, can be calculated. Alves et al. [27] have recently also gauged a number of systems with software metrics and derived thresholds for the metrics (e.g., for McCabe's cyclomatic complexity) based on the gathered values. In a more recent work [28], the authors improved the method by using an N-point rating system; however, given the experience with other approaches, more evaluations are necessary to show the usefulness of their approach.

A systematic survey evaluating maintainability models has recently been published by Riaz et al. [29]. The authors concluded that many maintainability prediction methods are merely based only on one of the three attributes, program size, complexity, or coupling. Methods based on cross-validation, prediction models, or accuracy measures are only rarely proposed let alone evaluated or really used in practice so that the use of one of these models without great expertise and care can easily lead to wrong results.

3.3. Comparable Results. Interestingly, industry seems to have a more pragmatic attitude towards using software metrics since measuring tools such as IBM Rational Logiscope (that reports a significant user base [30]) often contain maintainability models that claim to give indications whether a piece of code is in acceptable shape or should be inspected or rewritten. Stamelos et al. [30] have also received notable attention for their study of various open source applications from the Linux ecosystem (comprising slightly over 600,000 LOC) and contributed one of the rare sets of metrics data that can be used to compare our case study to. Table 2 provides an overview of their results.

Ramos et al. [31] have recently performed another maintainability analysis for five Cobol-based information systems and have also contributed some concrete numbers for the systems they studied.

However, it is important to note that the systems just presented [17, 30] are considerably smaller than the case study we present in this paper and originate from a completely different domain.

Table 2: Comparable results by stamelos et al. [30].

	Min.	Max.	Mean	SD	Median
Number of statements	3.00	92.00	23.43	12.25	21.65
Cyclomatic complexity	1.00	35.00	7.70	4.28	5.58
Max. nesting	1.00	8.00	2.99	0.81	2.94
Number of paths	1.00	32,767	1266.34	3317.85	704.96
Unconditional jumps	0.00	1.96	0.14	0.30	0.00
Comment frequency	0.00	1.32	0.11	0.15	0.08
Vocabulary frequency	1.5	9.90	2.75	0.93	2.67
Program length	18.00	516.00	133.38	73.17	122.82
Average size	3.68	14.96	6.35	1.58	5.96
Number of in/out	2.00	6.03	2.92	0.77	2.80

Table 3: Comparable results by Ramos et al. [31].

	Range
Line of code per system	469 k–816 k
Line of code per module	3 k–43 k
Cyclomatic complexity	11.7–50.6
Comment frequency	20.3%–29.8%
Program effort	32.7–1407.4
Fan-Out	0.5–9.4

4. Assessing the CMMS

Due to its significant size, we have performed an automated static code analysis of the CMMS in order to collect concrete values for the metrics used for our system evaluation. We have used Scientific Tools Inc. Understand (http://www.scitools.com/ (retrieved 07/2011)) in version 2.6 that supports most of the metrics introduced before. The central motivation behind this "measuring exercise" was to identify weak spots within the current architecture from a maintainability point of view that can be improved in future cabin software systems.

4.1. Collected Data. As briefly mentioned before, the investigated CMMS comprises roughly 1.5 million lines of C code. Interestingly, their distribution on 71 modules is quite uneven as there exist two modules (numbers 5 and 71), which alone represent more than 21% of the overall code. The average number of statements per module is 7,901 (the standard deviation is 10,527). Furthermore, each module contains an average of 112 functions (standard deviation 207 yielding a total of 7,951 functions in the system) and 16 files (standard deviation 15.7). The maximum number of files is 81 in module 7 while the maximum number of functions (also found in module 7) is 1,678.

Table 4 summarizes the metrics collected for this study and is intended as a starting point for the discussion and interpretation of the results following afterwards. As long as not otherwise stated, all values are measured for each source file within a module and then summed, respectively, averaged per module.

The heterogeneity found in the code sizes per modules is also characteristic for most other metrics we have collected for the CMMS. While most average values are within the recommended ranges, there are also a number of modules that stick out significantly—for example, having more than 240 functions in one file is clearly very (if not too) high. As far as space permits, we discuss some mentionable results in more detail to give a better impression of the heterogeneity found for each maintainability characteristic.

4.1.1. Analyzability. The literature suggests various metrics, such as those proposed by Halstead [10] or McCabe [11], for understanding the analyzability of (or in other words difficulty to understand) a piece of code. In our CMMS the Halstead's program length clearly indicates modules 25 and 19 as being the most substantial ones. Module 25 has a length of over 70,000, which is more than six-times larger than the second-largest module number 19 with 12,930. Nevertheless, module 19 is still nearly two-times larger than the next largest module. Furthermore, we recognize a few modules that are remarkable as their Vocabulary size is close to or even higher than 1,500 different terms, which is a clear warning sign for a high complexity and hence for code that might be hard to understand and eventually hard to maintain.

Module 5 reaches the highest maximum difficulty with 2,324 within one of its files. Furthermore, module 19 has the highest average difficulty with 961. The average Halstead volume of files within our case study is 20,697 per file (standard deviation: 78,183) and thus far beyond the recommended limit of 8,000. Consequently, 53.5% of the modules are on average above the threshold volume.

The largest Halstead effort measured is 4.516 billion (for module 25), the second largest is 6 million for module 71, and the average over all modules is 69.503 million. The box whisker plots (BWP) in Figure 1 illustrate the effort for the five most substantial modules (the effort per module is the sum of all measured file efforts within the module). Consider the plot of module 25 as an example, it contains a minimal effort of 72 K, a lower (1.6 M) and upper quartile (509 M), the median of 67 M, and a maximal Effort of 1.6 B found in all files of the module.

A recommended maximum found in the literature for the average size of statements (S) is 7 [32]. Based on that

Table 4: Results overview.

Metric	Recommended	Worst	Mean	Std. Dev.	Median	Within Rec.	Remarkable modules
Lines of code	—	72,300	7,901	10,527	4,925	—	5, 71
Number of modules	—	71	16	15	10	—	5, 71, 67
Functions per module		1,113		111	58		
Halstead length	—	70,000	2,845	8,257	9,493	—	
Amount of comments	≥ 0.5	0.25	0.82	0.37	0.72	81%	1, 19, 46
Functions (per file)	7 ± 2	242	7	5,34	6	71%	5, 22, 48
Vocabulary size	—	2,245	445	354	358	—	—
Difficulty	<30	2,324	186.5	143	152	2.8%	5, 71, 61
Volume	100–8000	667 K	20,697	78,183	9,053	46%	25, 71
Effort	—	4,516 Billion	69.50	531 M	2.3 M	—	25, 71
Size of statements	≤7	46	2.99	7.92	0.90	95%	19, 27, 49, 50
Fan-In	≤7	1,476	196	215	136	0%	5, 66
Fan-out	≤7	3,701	195	443	105	0%	5, 66
Cyclomatic complexity	0–10	383	5.74	51	30	45%	5, 31, 33, ≥57
Nesting	0–5	12	7.2	2.4	7	59%	53, 6, 2
GOTOs	0	0	0	0	0	100%	—
Frequency	—	101	0.08	11.68	3.85	—	26
Coupling (out)	—	235	44	41	38	—	5, 67, 68
Coupling (in)	—	294	41	47	29	—	5, 67,68
Entry nodes	7,951	—	112	207	—	—	—
Exit nodes (per function)	5,618	—	3.34	—	—	—	—

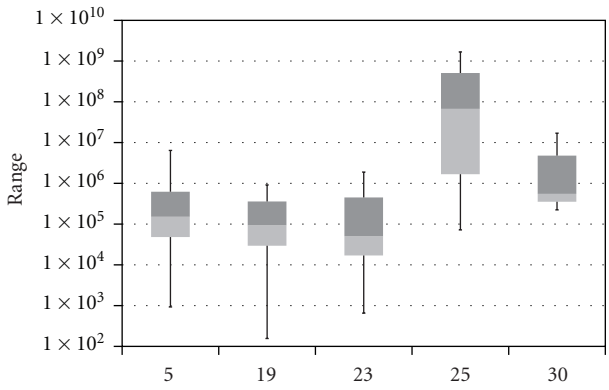

Figure 1: Halstead effort as Box-Whisker plot containing the most prominent modules.

prerequisite, most CMMS modules are again absolutely unremarkable since they have an average statement size of smaller than 5. However, four modules clearly stick out in this measure and hence "qualify" for a detailed inspection later, namely, modules 19 ($S = 46$), 25 ($S = 31$), 49 ($S = 11$), and 50 ($S = 40$). The average fan-out of all modules is 11 (standard deviation is 10). The highest measured Fan-Out is 62 in module 5, which is thus more than five-times greater than the average and far beyond good and evil regarding the recommendation. Furthermore, it underlines that module 5 is depending on 87% of all other modules in the system. Besides module 5, modules 68 (Fan-Out 52) and 67 (Fan-Out 43) have also recognizably high values. This can be interpreted as another clear warning sign that some modules may be incoherent and too complex.

The calculation of McCabe's cyclomatic complexity for the case study also delivered quite interesting results: the CMMS has an average of 5.7 (standard deviation: 2.29) per function and hence lies well within the recommended range [11]. Figure 2 gives an overview of average and maximum values per module.

Alarmingly, however, the highest measured cyclomatic complexity is 383 for a function in module 10, and furthermore, 74% of all modules have at least one function with complexity equal or greater than 20. As visible in Figure 2, we have even found five modules containing functions with Cyclomatic Complexities over 100.

The maximum nesting level measured for the CMMS is 12 (found in module 5). Furthermore, the modules 16 and 17 come close to this value with their nesting of 11 so that 76% of the modules in the CMMS exceed the recommended threshold of 5 at least in one function.

4.1.2. Changeability. Some common changeability metrics such as the average size of statements have already been discussed before. Others, such as Halstead's vocabulary, are commonly seen as a helpful indicator for estimating changeability. Again most modules seem to be unremarkable in this regard; only module 25 with a value of over 100 clearly stands out. Fortunately, the number of GOTOs in the CMMSs investigated code base is zero.

As the CMMS was developed under hard real-time constraints, we expected to find quite a number of global variables and were not disappointed since in total it contains

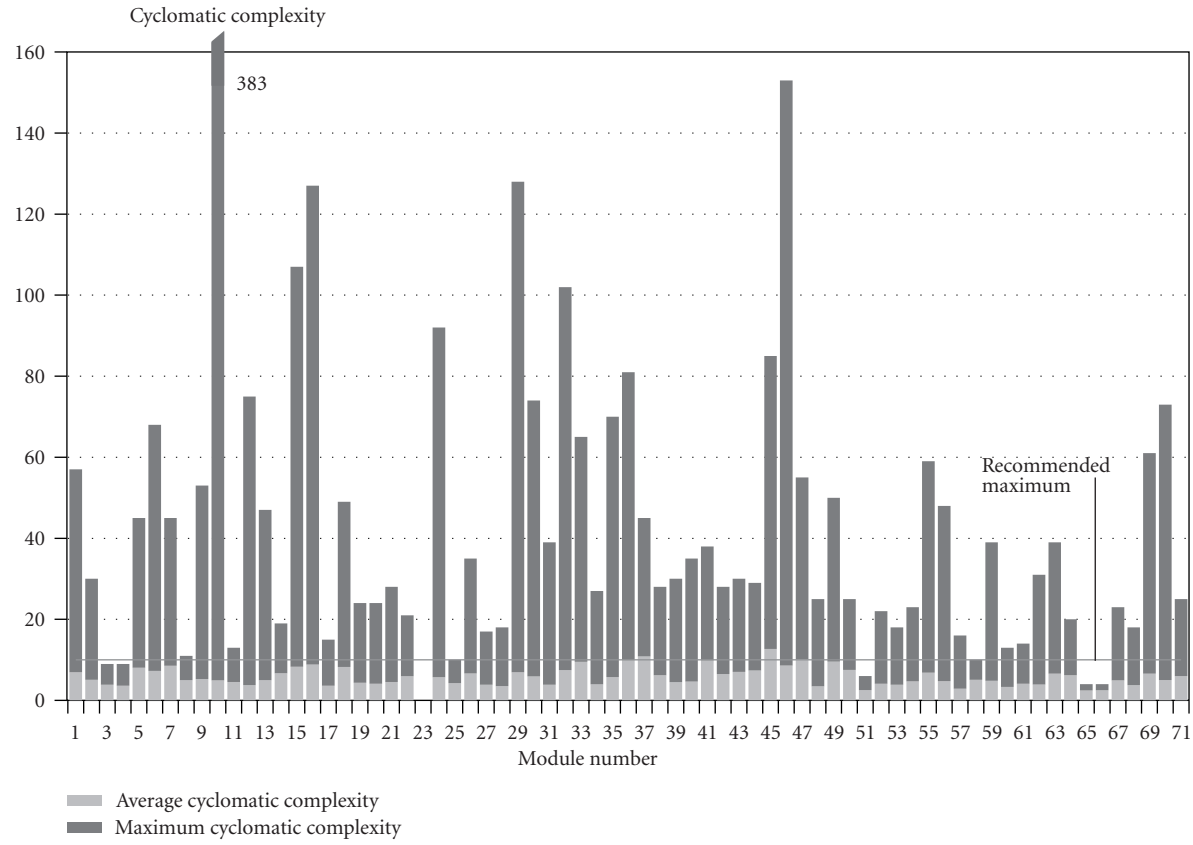

Figure 2: Cyclomatic complexity per module.

approximately 1,300 of them. On average this yields around 18 global variables per module, a number that is confirmed by a few manual samples showing that, for example, module 71 contains 8, module 12 contains 13, and module 68 contains 23 global variables. This will certainly influence changeability negatively since as soon as one of these variables needs to be changed the complete code-base needs to be checked for statements accessing it.

The highest in-bound and as well out-bound coupling has been measured for modules 5 in 294/out 235, 67 in 203/211, and 68 in 200/216. All other modules have less than 120 dependencies, respectively clients, which is still a very high value, though.

4.1.3. Stability. Additional metrics and analysis methods we used for measuring the stability of the CMMS are the number of entry and exit points, and the Fan-In per module. This is because of the fact that basically every C function has exactly one entry point (as long as no GOTOs are used), and thus the number of entry points per module can easily become large and is equal to the number of functions. Consequently, the system has 7,951 entry points, which makes an average of 112 per module (standard deviation: 207). Although this is per se not very remarkable for a C program, it bears a significant risk for maintainability as every function can be called from anywhere in the whole system. This means that the change of any function's signature may entail changes scattered all over the system.

Similar to the entry point metric, every C function has at least one exit point, namely, the end of the function. However, in contrast to entry points, a function can have additional exit nodes whenever a return statement is used within, for example, an "if" statement before the actual end. In total, we have counted 5,618 return statements which average at 3.34 returns per function and thus give us no cause for increased concern. The average fan-in in the CMMS is 195 (standard deviation 443). Module 5 again stands out as it has the highest fan-in with 3,700 calls coming in from various places and is thus used by almost half of the CMMS.

4.1.4. Testability. The final facet we investigated is testability, characterizing the ease of testing, which is one very important step in the development of every aviation system. Standards and guidelines, such as DO-178B [3], define the required steps and methods for testing airplane and related software. Hence, we have evaluated the size of test suites, the achieved test coverage, and metrics such as cyclomatic complexity, nesting, and unconditional jumps.

As explained earlier, every aviation system needs to be certified against the regulations of governmental agencies based upon DO-178B. Thus, code is classified into different design assurance levels (DALs) that need to fulfill different testing requirements depending on their criticality for safety.

Table 5: Comparison of maximum values found in data sets.

Metric	Ramos	Stamelos	CMMS	Rec. Limits
Overall LoC	3 M	606 K	1.5 M	—
LoC per module	3 k–43 k	4 k–40 k	172–72,300	—
Comment LoC	20.3%–29.8%	0–132%	20–242%	
Cyclomatic complexity	11.7–50.6	1–35	0–383	<10
Nesting	—	1–8	0–12	0–5
Fan-out	0.5–9.4	2.00–6.03	1–143	—
Frequency	—	1.5–9.9	0–2242	—
Length	—	18–516	0–70,173	—
Effort	32.7–1407.4	—	0–4.5 B	—

The most critical DAL A code needs to achieve full MC/DC [12] coverage, which automatically includes decision coverage and statement coverage. However, our CMMS case study does neither contain DAL-A nor DAL-E code. Functions on DAL B still need to prove full decision coverage, which in turn implies full statement coverage. 46% of the CMMS's source code is on DAL-B. On DAL-C an airplane manufacturer still has to demonstrate statement coverage of 100%. In the CMMS this is the case for 38% of the source code. Finally, DAL-D constitutes 16% of the code, which is, however, not subject to any special testing requirements.

5. Further Analysis and Discussion

The results presented so far in this paper are raw metrics values with a limited significance. All we can do on this basis is to speculate that modules with exceptionally high values should be inspected manually for maintainability problems. Nevertheless, we have formulated the hypothesis that we expect the metrics values of the CMMS to be higher than other numbers reported in similar studies ([17, 30, 31]) with less presumed domain complexity (H1).

In comparison to the other results, our maximum values are actually much higher which confirms hypothesis H1b. As example, the highest measured cyclomatic complexity in the CMMS is 383, while Stamelos et al. [30] has measured a maximal value of 35 and Ramos et al. [31] of 50. Furthermore, the Halstead effort of the CMMS is over 4 billion compared to merely 1,407 in the Ramos study. The results listed in Table 5 illustrate that we have found significantly larger values for almost all metrics measured.

As also visible in the table, the measured maximal values for nesting (CMMS 12/recommendation <6) and cyclomatic complexity (383/<10) are significantly higher than the thresholds recommended in the literature. Interestingly, the outcome for the average values (H1a) is somewhat different, as shown in the following Table 6.

Only the average values of the Halstead metrics are significantly higher for the CMMS. All other values are in a similar range, which clearly shows that the CMMS has higher peak values, but in average the values are relatively "normal," at least compared to the relatively small number of published similar studies. This may be an indicator for a higher inherent (domain) complexity, but is certainly subject to further investigation.

Table 6: Average results in comparison to other studies.

Metric	CMMS	Stamelos	Ramos
Cyclomatic complexity	5.74	7.70	23.82
Maximum nesting	7.2	2.99	—
Unconditional jumps	0	0.14	—
Number of FAN/out	3.33	2.92	2.92
Effort	69 M	—	595.78
Number of statements	3,173.42	23.43	—

5.1. Metrics Dependencies. The expressiveness of software metrics from the perspective of software maintenance has been subject of an ongoing discussion for many years. Especially the Halstead metrics have been criticized repeatedly for not being very useful in this context since they are deemed being too closely related with each other. In order to obtain a better understanding in this context, we decided to analyze whether there is a correlation between the (module-based) metrics we have used, which was our second hypothesis (H2). Since our measurement results merely yielded ordinal-level data, the common Pearson correlation cannot be used. Thus, we decided to calculate the more general Spearman rank correlation coefficient [33] for the ranks of each module per metric. The outcome of this effort is presented in the correlation matrix illustrated in Table 7.

The matrix illustrates two clusters with high correlations, visualized with bold text, that all also have a P value smaller then 0.05. The first cluster, in the upper left corner, contains the Halstead metrics Vocabulary, volume, difficulty, and length. All these metrics are based on the "Halstead fundamentals" operators and operands and hence it is no surprise they are closely correlated with each other. The second cluster includes all lines of code-based metrics, such as Blank LOC, commented LOC, declarations, statements, and the overall LOC, and thus is also understandable.

Another interesting fact revealed by the matrix is that Halstead's difficulty and McCabe's cyclomatic complexity, both metrics that are intended to indicate complexity of code, are not correlating. On the other hand, Halstead's effort

Table 7: Matrix of Spearman rank correlations.

	Voc.	Vol.	Diff.	Eff.	Avg. CC	Max. CC	Nest.	BLOC	Decl	Stat.	CLOC	LOC
Len.	**0.95**	**1.00**	**0.95**	***0.71***	0.25	0.44	0.35	0.46	0.42	0.51	0.53	0.50
Voc.		**0.93**	**0.95**	0.59	*−0.08*	0.35	0.32	0.39	0.37	0.45	0.49	0.45
Vol.			**0.94**	***0.72***	0.28	0.46	0.35	0.47	0.43	0.51	0.53	0.51
Diff.				0.58	0.19	0.35	0.35	0.36	0.35	0.42	0.42	0.41
Eff.					0.39	***0.70***	0.41	***0.85***	***0.79***	***0.84***	***0.85***	***0.85***
Avg. CC						0.68	0.55	0.43	0.30	0.39	0.29	0.35
Max. CC							0.61	***0.77***	0.63	***0.74***	0.68	***0.72***
Nest.								0.45	0.33	0.47	0.40	0.42
BLOC									**0.91**	**0.95**	**0.94**	**0.96**
Decl.										**0.95**	**0.92**	**0.96**
State.											**0.95**	**0.99**
CLOC												**0.98**

seems to correlate significantly with the size of the code again, which seems logical since more code needs more effort to be written. The relatively high correlation of the maximum cyclomatic complexity with the LOC seems also reasonable as longer code is also likely to contain more branches.

5.1.1. Threads to Validity. Due to the size of our code base there was no other alternative to derive the desired metrics than to collect them automatically. We have used a proven state of practice tool and hence see no reason to expect large errors caused by the measurement approach. However, we cannot guarantee that another tool would have measured the exact same values since especially some fundamentals of the Halstead metrics (operators and operands) may still be interpreted slightly different from another tool. Since we performed all measurements with the same version of the tool and the same code base, we see no threads to the internal result validity. Clearly, the results in our data set are from a very specialized domain so that it is likely that they are only valid within the avionics domain for similar systems. Since this is the first comprehensive data collection of its kind we are aware of, its results should still be taken with a grain of salt or even better be complemented with benchmarks from similar systems in order to gain a better understanding of the factors influencing the maintainability of avionic software systems.

5.1.2. Lessons Learnt. Our case study revealed once again that measuring maintainability of a software system still remains a difficult challenge. As a consequence, it is not possible to label a large system as a whole or even its modules as being maintainable or not. Since comparable studies are scarce, we cannot make any reliable absolute statement in this regard, but only compare various modules of the case study amongst each other and identify those that raise suspicion due to extreme results for some metrics. Currently our only hope for resilient comparative values is that our case study motivates the future publication of similar studies that might deepen the understanding of maintainability metrics and perhaps even underpin their expressiveness with empirical studies in which actual change requests are implemented for the systems under investigation.

Thus, as previously indicated, one central result of our study is that the maintainability of our case study sometimes seems to vary even within one module. This fact is probably influenced by the relatively large number of independent project partners that have contributed to the system. Hence, one consequence of this result should be the creation and establishment of more strict coding guidelines that include measuring at least some of the metrics used in this study and ensuring that rather tight standards for them are met. Given the large size of the investigated CMMS it makes sense to automate this effort as far as possible with tools such as Checkstyle [34]. However, it clearly must be avoided that a system is only optimized for metrics values in the future.

In order to improve the volume and the length of program files, we suggest separating commonly used modules from modules with specific functions (i.e., enforce a stricter encapsulation of functionalities). In order to reduce the high coupling, we suggest developing an improved communication model, which defines clear interfaces between modules. Interestingly, the large number of contributors is somewhat contradicting the high coupling we have found between numerous modules. Although this is probably mostly owed to real-time constraints and the sometimes difficult integration of various hardware devices into the system, this is another issue that needs to be solved for future cabin software systems. In general, we see a clear need for a more sustainable system architecture that has a stricter separation of concerns and also defines standards (such as the use of layering) for the internal structure of individual modules. The recently completed revision of the DO-178 guidelines standard is certainly another necessary step for making progress in this direction, since many potentially interesting techniques (such as object-orientation) could not be certified for use on an airplane with the old DO-178B. The new version, which was finally accepted in January 2012 (http://www.rtca.org/comm/Committee.cfm?id=55), eventually allows object-orientation and model-driven development for all design assurance levels. Furthermore, the supplemental DO-332 [35] describes the usage of

object-oriented methods and garbage collection in safety-critical environments.

6. Conclusion and Future Work

In this article, we have presented an assessment of the maintainability of a large avionic software system. Since similar studies in the literature are sparse [17, 30] and from different domains, we believe that we have contributed a useful benchmark of metrics in a practically relevant case study. Over all, we were able to determine that large parts of our case study's code-base are well within the recommended limits found in the literature, although there are a few outliers somewhat blurring this good impression. It is also interesting to note that on average the investigated CMMS does not have higher metrics values than the few other—presumably less complex—systems presented so far in the literature. Only some maximum values are much higher so we have to assume that only a small part of the CMMS carries the core of its complexity. However, since the impact of concrete metrics on maintainability is generally still not understood in detail it is currently impossible to directly label modules as maintainable or not. Hence, the central benefit of this study can be seen in the identification of "suspicious" modules that perform poorly in various metrics and thus seem to be a good starting point for architectural considerations in future software versions.

After this analysis, we have to conclude that it is about time to learn how to apply modern software engineering techniques (such as objects or components) for the development of aviation software in order to keep track with the ever growing complexity in this field. This should also help to ensure a better long-term maintainability at the same time, but clearly requires to keep testing and certification standards high, which may also require developing new techniques for this purpose in the future. The recently published guidelines (DO-178C [36]) are important steps in this direction that open new possibilities for software design (as, e.g., already sketched in [37]) and development for aviation systems on the one hand. On the other hand, it will certainly require some time until the aviation industry and the certification bodies have developed the necessary procedures and guidelines to use these new possibilities successfully.

References

[1] Klocwork, *Streamlining DO-178B Efforts with Source Code Analysis*, Klocwork, Burlington, Canada, 2009.

[2] J. M. Voas and K. W. Miller, "Software testability: the new verification," *IEEE Software*, vol. 12, no. 3, pp. 17–28, 1995.

[3] RTCA, *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, DO-178B/ED-12B, RTCA/EUROCAE, 1992.

[4] D. C. Sharp, "Reducing avionics software cost through component based product line development," in *Proceedings of the 17th AIAA/IEEE/SAE Digital Avionics Systems Conference*, 1998.

[5] N. E. Fenton and S. L. Pfleeger, *Software Metrics*, Chapman & Hall, 1991.

[6] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pp. 452–461, May 2006.

[7] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proceedings of the 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISCE '06)*, pp. 18–27, ACM Press, New York, NY, USA, September 2006.

[8] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[9] S. Burger and O. Hummel, "Applying maintainability oriented software metrics to cabin software of a commercial airliner," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, Szeged, Hungary, 2012.

[10] M. H. Halstead, *Elements of Software Science*, Elsevier, 1977.

[11] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[12] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[13] DO254, *Design Assurance Guidance for Airborne Electronic Hardware, Research and Development for Alternate Antenna Designs is Needed to Solve the Antenna Crash Tolerance and Installation Problems*, DO-254 /ED-80, 2000.

[14] ISO, "International Standard ISO/IEC 9126," Information technology: Software product evaluation: Quality characteristics and guidelines for their use, 1991.

[15] ISO/IEC, *25010 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*, 2011.

[16] D. Stavrinoudis and M. Xenos, Comparing internal and external software quality.

[17] D. Spinellis, G. Gousios, V. Karakoidas et al., "Evaluating the quality of open source software," *Electronic Notes in Theoretical Computer Science*, vol. 233, pp. 5–28, 2009.

[18] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, pp. 728–738, 1984.

[19] S. R. Chidamber and C. F. Kemerer, "Metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[20] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," in *Proceedings of the IEEE Reliability, Maintainability Symposium*, 2002.

[21] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Transactions on Software Engineering*, no. 5, pp. 510–518, 1981.

[22] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, 1979.

[23] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[24] C. Jones, "Software metrics: good, bad, and missing," *IEEE Computer*, vol. 27, no. 9, pp. 98–100, 1994.

[25] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *IEEE Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[26] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability—a preliminary report," in *Proceedings of the 6th International Conference on the Quality*

of Information and Communications Technology (QUATIC '07), pp. 30–39, September 2007.

[27] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '10)*, September 2010.

[28] T. L. Alves, J. P. Correia, and J. Visser, "Benchmark-based aggregation of metrics to ratings," in *Proceedings of the 21st International Workshop and 6th International Conference on Software Process and Product Measurement (IWSM-MENSURA '11)*, 2011.

[29] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*, pp. 367–377, October 2009.

[30] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," *Information Systems Journal*, vol. 12, no. 1, pp. 43–60, 2002.

[31] C. S. Ramos, K. M. Oliveira, and N. Anquetil, "Legacy software evaluation model for outsourced maintainer," in *Proceedings of the European Conference on Software Maintainance and Reengineering (CSMR '04)*, pp. 48–57, March 2004.

[32] T. Weber, "Logiscope—Softwaremesstool Version 1.5.1," Universität Otto von Guericke Magdeburg, 2011, http://home.arcor.de/titusweber/logi/logi.html.

[33] P. Embrechts, F. Lindskog, and A. McNeil, "Modelling dependence with copulas and applications to risk management," *Handbook of Heavy Tailed Distributions in Finance*, vol. 8, no. 1, pp. 329–384, 2003.

[34] O. Burn, "Checkstyle homepage," 2012, http://checkstyle.sourceforge.net/.

[35] DO-332, *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A, Research and Development for Alternate Antenna Designs is Needed to Solve the Antenna Crash Tolerance and Installation Problems*, DO-332, 2012.

[36] RTCA, *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*, 2012.

[37] S. Burger and O. Hummel, "Towards automatic reconfiguration of aviation software systems," in *Proceedings of the IEEE International Workshop on Industrial Experience in Embedded System Design*, Munich, Germany, 2011.