Generating Qualifiable Avionics Software: An Experience Report

Andreas Wölfl*, Norbert Siegmund*, Sven Apel*, Harald Kosch*, Johann Krautlager†, Guillermo Weber-Urbina†
*University of Passau, Germany
Email: {andreas.woelfl, norbert.siegmund, apel, harald.kosch}@uni-passau.de
†Airbus Helicopters S.A.S., Germany
Email: {johann.krautlager, guillermo.weber-urbina}@airbus.com

***Abstract*—We report on our experience with enhancing the data-management component in the avionics software of the NH90 helicopter at Airbus Helicopters. We describe challenges regarding the evolution of avionics software by means of real-world evolution scenarios that arise in industrial practice. A key role plays a legally-binding certification process, called *qualification*, which is responsible for most of the development effort and cost. To reduce effort and cost, we propose a novel generative approach to develop qualifiable avionics software by combining model-based and product-line technology. Using this approach, we have already generated code that is running on the NH90 helicopter and that is in the process of replacing the current system code. Based on an interview with two professional developers at Airbus and an analysis of the software repository of the NH90, we systematically compare our approach with established development approaches in the avionics domain, in terms of implementation and qualification effort.**

I. Introduction

Engineering in the avionics domain is driven by various legally mandated regulations, dedicated to safety and reliability. Prior to the first official flight in a commercial aircraft, a national certification authority has to inspect the airworthiness of all components, including airborne computers and software. This process is called avionics *qualification*. The acceptance is strictly related to compliance with international aerospace standards, such as DO-178: Software Considerations in Airborne Systems and Equipment Certification [1]. Together with white papers[1] from the Certification Authorities Software Team, these documents define process completion and functionality verification with software life-cycle objectives. Depending on the (safety) criticality of the target system, applicants for qualification have to meet a specific set of life-cycle objectives, determined by different Design Assurance Levels[2].

In industry, compliance with DO-178 adds about 75%–150% to the total development costs [2], caused by additional effort for developing or extending code and documentation for life-cycle objectives, called *qualification assets*. By over two third, the majority of these artifacts relates to software verification and validation (e.g., establishing bidirectional traceability evidence from system requirements to unit tests) [1]. Developing qualification assets also complicates programming of the software itself, because many assets must adhere to strict coding constraints, such as the prohibition of late binding or dynamic dispatch. The impact is most significant between Design Assurance Level D and C. Level C is assigned to *mission-critical* software and demands about 30% more budget and schedule than software at Level D [3].

In this paper, we report on our experience gained in a cooperation between Airbus Helicopters and the University of Passau. The key objective of the cooperation was to develop a highly configurable real-time database management system, intended for integration in core mission-control applications in the avionics software of the Nato Helicopter 90 (NH90). Analogous to product lines in other industry branches [4], customers in the aviation domain can choose among a large number of features and configuration options for the airborne equipment, which requires variability of the avionics software in the NH90, including the data-management component.

In a first step, we investigated state-of-the-art development methods used at Airbus Helicopters. There are two basic approaches to handle variability in qualified avionics software: The first approach is to consider a software variant as separate unit, such that for each variant, the source code must be implemented and qualified manually. The second approach is to apply specifically qualified development tools to automatically generate variant-specific software artifacts. Based on the findings of a semi-structured interview with two senior developers at Airbus Helicopters, we found that these approaches are economically suboptimal implementing data management in the avionics domain. While an approach of separately implementing and qualifying software variants does not scale with an ever increasing number of new requirements, introducing a qualified development tool causes huge initial costs for either purchasing a commercial tool or implementing and qualifying a suitable tool in-house.

Based on our analysis, we propose a novel generative development approach that addresses the challenges of variability and software evolution in the context of qualification demands in the avionics domain. Essentially, we combined the aforementioned approaches. In a nutshell, using model-based and product-line technology, we generate tailored system variants based on declarative specifications written in a domain-specific language. Furthermore, we ease software verification and validation by automatically generating qualification assets in this process. This way, we have a scalable approach in terms of the number of variants and we keep the initial effort feasible by supporting the qualification of the generated code instead of qualifying the tool itself. We integrated our approach into

[1] https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/

[2] Scale with five criticality levels, ranging from level A (most critical) to level E (least critical), assigned by the national certification authority.

the tool chain of the NH90 development environment. Our implementation is fully operational. The resulting software variants are executable on the target avionics hardware and currently prepared for qualification at Design Assurance Level C as part of the next release of the NH90 System Software.

To compare our approach to established development approaches, we systematically describe the evolution of avionics software by means of real-world scenarios regarding data management in the NH90. In detail, we discuss frequency and severity of functional and non-functional requirements by combining insights from a developer interview and data collected from the software repository of the data-management component of the NH90 System Software. The software repository contains the problem reports, software change notes, and engineering change requests of the past 18 years. In the long run, developers can use our results to assess maintenance effort in an avionics software project and determine a development strategy for concrete scenarios.

In summary, we make the following contributions:

- We analyze development approaches applied in the avionics domain regarding practicability and feasibility in terms of the challenges of data-management and qualification demands in the avionics domain.
- We report on a developer interview at Airbus Helicopters, providing insights into the implementation, maintenance, and evolution of avionics software.
- We present a novel generative development approach based on model-driven and product-line technology, aiming at the efficient realization of both implementation and qualification of avionics software.
- We implement our approach as part of the real-time database management system of the NH90 avionics software at Airbus Helicopters.
- We quantitatively and qualitatively discuss our approach based on real-world evolution scenarios combined with the analysis of the software repository of the NH90 System Software related to data management.

II. Avionics Software

In this section, we explain the target avionics software environment and evolution with a focus on data management. Furthermore, we characterize general challenges in developing avionics software and describe established development approaches with a focus on software qualification.

A. NH90 Avionics Software Evolution and Variability

The software division at Airbus Helicopters develops the on-board software for federated avionics in the NH90. The NH90 is a medium sized multi-role helicopter, manufactured by the NHIndustries consortium. Its operating system consists of an application framework, called *NH90 System Software*, and several application components, called *Operational Processing Functions*. The programming language of the NH90 System Software is Ada[3], which has been specifically designed for the application in safety-critical embedded and real-time systems. The first qualified version of the NH90 System Software was released in 1995, the year of the maiden flight of the NH90. From the first 2 variants for field- and maritime missions, economical success raised this number to 11 variants in the early 2000s. From here, obsolescence led to the introduction of additional hardware architectures, causing further variation in the software. Today, 45 variants are deployed on 3 different hardware platforms, shipped to customers in 14 nations [5].

The helicopter variants differ in their integral equipment configuration. Complex equipment provides optional features, from which customers can select arbitrary combinations. It is also possible that already integrated equipment from a delivered helicopter is replaced by a functional equivalent that is manufactured by another company. The effects on the associated avionics software are significant. The most frequently changed area is the combination and the implementation of Operational Processing Functions, which control or monitor subsets of the avionic equipment. Some functions rely on the same data or services, so the presence or absence of Operational Processing Functions can cause interactions between individual components, particular, affecting data management and data-bus scheduling. An amendment in the Operational Processing Functions regularly requires further adjustments in the functional capabilities of these components.

The constantly growing number of variants and the resulting complexity requires a re-development of major parts of the NH90 System Software. The primary goal of our endeavor was to improve extensibility and maintainability of the data-management component, while providing sufficient performance and resource consumption to deploy the software even on legacy hardware platforms. As part of this re-development endeavor, Airbus Helicopters initiated the cooperation with the University of Passau.

B. Data Management in the NH90 System Software

A central component that posed problems regarding the evolution of variability in the NH90 System Software is the data-management component. A single avionics computer contains about 10–15 Operational Processing Functions that use the data-management component to store and process various data, structured by about 30 entity types[4]. When we started the cooperation, data management was implemented as list-based storage, henceforth called *List Management* (LSTM). All data were kept as plain lists of data records. Complex data-management functionality was left to the individual Operational Processing Functions. With the increase of variability over time, the developers faced three major problems: First, relationships between entities were only implicitly present in the source code of the Operational Processing Functions. Second, there was no guarantee that referential integrity of the data is maintained consistently. Third, values of related entities had to be stored redundantly.

The re-development group at Airbus Helicopters concluded that the current data-management solution LSTM should be replaced by a data- and function-centric component. Since there is no commercial-of-the-shelf product available that can be qualified according to the *Plan for Software Aspects of Certification*[5] of the NH90, Airbus Helicopters decided to pursue an in-house development in cooperation with the

[3]http://www.adacore.com

[4]The complex data types to handle records in the data-management component (e.g., `Waypoint`).

[5]An agreement between the aircraft manufacturer and the certification authority concerning certification activities.

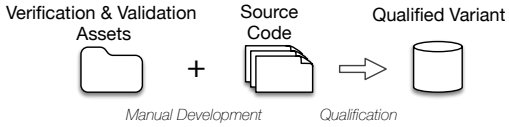

Fig. 1: Code-based development of qualified avionics software

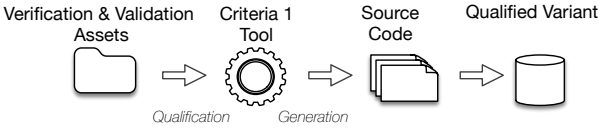

Fig. 2: Tool-based development of qualified avionics software

University of Passau, henceforth called *Embedded Database* (EDB). To accomplish this goal, we first had to discover the factors that complicate software development in the avionics domain.

C. Coding Constraints for Avionics Software

Based on accumulated knowledge gathered over years, Airbus Helicopters meets high standards with respect to software quality and safety, resulting in a large number of non-functional requirements. A subset of these requirements relates to qualification, showing compliance with DO-178 for civilian or DOD-STD-2167 [6] for military aviation. Conformity to these standards makes programming considerably more difficult and tedious. For reliability assurance in mission- and safety-critical avionics software, there must be a proof that program execution and resource consumption are entirely deterministic and predictable. Depending on the system criticality, heavy non-functional requirements prevail. In the case of the NH90 System Software, we list some representative examples of coding constraints, referring to mission-critical code, qualified at Design Assurance Level C:

- Dynamic programming mechanisms, such as late binding or dynamic dispatch are prohibited.
- To predict the memory footprint at compile time and to guarantee faultless resource consumption, memory must be allocated statically.
- Source code that is never executed in any configuration has to be completely removed.
- Each line of source code must traceably correspond to a low-level system requirement.

To get permission for flight, each development artifact must be approved by designated engineering representatives from the national certification authority. Next, we explain the current practice in qualifying avionics software.

D. Qualification of Avionics Software

Software verification and validation is the most integral part of the qualification process. In this context, DO-178 issues two methods: *formal methods* and *requirement-based testing*. Guidance for formal methods is documented in DO-333 [7]. It recommends model checking, theorem proving, and abstract interpretation as approved techniques for formal verification. By contrast, testing aims at validating the correctness, completeness, unambiguousness, and logical consistency of system requirements by running a comprehensive set of tests. In addition, bidirectional traceability between requirements, source code, and associated tests has to be established [8].

Independent of verification and validation, two software development approaches are commonly used in the avionics domain. In code-based development, both source code and qualification assets are implemented manually and submitted to the certification authority for individual qualification (Figure 1). By contrast, tool-based development relies on tool-generated software artifacts (Figure 2). A generator tool, whose output is part of the airborne software (in DO-178 as *Criteria 1* tool), must be qualified in compliance with the same software life-cycle objectives at the same Design Assurance Level as the resulting software itself. The benefit of this tool qualification is that the output is automatically considered as verified and validated [9], such that no additional approval is required for the integration of generated artifacts into the airborne software.

To properly determine the benefits and drawbacks of these approaches regarding the enhancement of data management in the NH90, we asked developers to report on their experience gained in the software development process of the currently operating component, which we describe next.

III. Developer Interview

In this section, we report on the setup, conduct, and results of an interview with two senior software developers employed at Airbus Helicopters. The goal was to collect information on the current data-management component of the NH90 avionics software and implications for our new solution. First, we list our research questions, followed by a description of the participants, questionnaire, and conduct. The remainder is structured by the research questions. The survey follows the guidelines provided by Jedlitschka and others [10].

A. Objectives

The subject of the interview was the development, maintenance, and evolution of the LSTM, the currently operating data-management component of the NH90 avionics software. We had the following research questions:

RQ_1 How much effort was required to implement and qualify the initial version of the LSTM?
RQ_2 How much effort requires deriving a new LSTM variant?
RQ_3 What situations and circumstances cause typical maintenance and evolution tasks for the LSTM?

B. Participants

Two senior developers of the software division at Airbus Helicopters participated in the interview. The first developer is a senior systems software analyst and developer. During 25 years of professional experience in the aviation domain, he worked with real-time SCADA systems, supervisory control software and data acquisition. With 19 years within the NH90 software team, he is one of the leading senior developers and assists younger engineers with training and knowledge transfer. He designed and implemented the LSTM and still maintains it. The second developer has been employed for more than 20 years in the avionics software domain. He has built consolidated expertise in the NH90 software team for more than 10 years. His responsibilities include all areas of

TABLE I: Topics of the semi-structured interview

Objective	Topic
RQ_1, RQ_2	Chronological sequence of the LSTM development
RQ_1	Application design
RQ_1, RQ_2	Functional and non-functional requirements
RQ_1, RQ_2	Stakeholders of the LSTM in the NH90 System Software
RQ_2	Proceedings to add, modify, and delete a list
RQ_2, RQ_3	Hardware platform and compiler-specific aspects
RQ_1	Development approach for unit tests
RQ_3	Software bugs
RQ_1, RQ_2	Procedure to verify and validate a new or modified list
RQ_1	Approach to establish traceability
RQ_3	Evolution of functional requirements
RQ_3	Evolution of non-functional requirements
RQ_3	Reasons of emergence of new requirements
RQ_3	Decision making for replacing the LSTM with the EDB

the software life cycle, mainly the development of control applications and the design of the NH90 operating system.

C. Questionnaire and Conduct

The interview was designed as a semi-structured interview. The topics are outlined in Table I. Besides the efforts for initial implementation and qualification, we wanted to collect information on what factors contribute to maintenance and evolution effort of complex multi-variant avionics software in industrial practice. We were particularly interested in reasons and frequency of changes in the software. Our goal was to get insights into long-term implications and severity of changes regarding the realization of new and unexpected requirements in the avionics domain. Therefore, we asked the interviewees about real-world challenges, solution approaches, and the time spent on resolving problems. The conversation was conducted face-to-face at the software division of Airbus Helicopters and lasted about 4 hours in total.

D. RQ_1: How much effort was required to implement and qualify the initial version of the LSTM?

The LSTM was implemented by hand (code-based development). The first qualified release was used for data-management in 7 NH90 variants. The programming of the initial version lasted about 18 person months.

> "[The requirements] were constantly moving to [...] a year and a half, really realistically. If you look at the code, it's not a huge many thousands of lines, but the concept, well you have to satisfy the constraints on the requirements."

For qualification, requirement-based testing was used. As part of the specified process in DO-178, a preliminary design was created before programming. This document comprises the initial software specification, including a mapping of low-level system requirements to procedures in the implementation. Traceability evidence as validation asset was built on this mapping. It is realized by adding annotations in the form of machine-readable comments to procedure headers. The completeness and integrity of the traces is then automatically proved by tools. The interviewees estimated the effort to create the combination of preliminary design and traceability evidence at 2.5 person months, which is 10% of the total development cost. The development of validation assets in form of functional tests and unit tests was extensive.

> "[There are] a lot of [tests], almost as much effort to develop the Operational Processing Function, [...], I would say easily 6 months effort, easily"

In summary, the development time for qualification assets of the initial version amounts to about 8.5 person months.

E. RQ_2: How much effort requires deriving a new LSTM variant?

The LSTM relies on generic software packages. In Ada, generic units are the instruments to safely implement parametric polymorphism. A generic package represents a parameterized template for a package whose parameters can be types, variables, subprograms, and other packages. Each instantiation conceptually creates a copy of the specification and body of the generic package, customized due to the actual parameters. All LSTM variants share the same operational functionality. They are distinguished by the data dictionary, which is defined by the absence or presence of certain Operational Processing Functions in a specific NH90 variant. Each contained entity type (e.g. `Waypoint`) is implemented as an independent list, based on a central generic package. It requires 4 type definitions and 9 subprograms to define the storage structure.

> "Accessing the data is not hard, that is the easy part. The difficult part is where it gets the data that you would like to modify. [...] Usually, the operator or the pilot is making changes to the graphical display, which is representing the data of one record. [...] This [interaction] is taking the developer three months, but the instantiation that is just one day."

The total cost of deriving a new LSTM variant is the effort required for adding or modifying the instantiation of lists, which is 8 person hours per entity type. On a single avionics computer, the NH90 System Software uses up to 30 different lists by 10–15 Operational Processing Functions.

F. RQ_3: What situations and circumstances cause typical maintenance and evolution tasks for the LSTM?

Maintenance effort arises from errors and flaws in the source code. The LSTM head developer reported from source-code changes due to bug fixes, but estimated the corresponding costs to be minor.

> "[...] if you consider 15 years of experience, your code keeps getting better. I mean, the software evolved of course, but not much of changes because of bugs."

The majority of maintenance and evolution effort for the LSTM is caused by engineering change requests, which are requests for system adjustments arising from new functional or non-functional requirements. The reasons are mainly new customer requests and changes in terms of on-board equipment. The new system requirements are allocated to software requirements, triggering development and maintenance activities for the corresponding components.

In addition to these general situations, we were interested in concrete scenarios that represent typical maintenance and evolution tasks in the context of the LSTM. The interviewees shared a number of actual scenarios, from which we present three selected examples for unexpected functional and non-functional requirements with differing severities. Each scenario

describes a real-world problem, the actual solution, and the time required to resolve it.

Hardware platform: In 2003, a new avionics hardware platform was introduced for two NH90 core computers. The new board differed in the number of processors and switched the processor architecture from CISC to RISC. The compiler required a version upgrade of the runtime kernel and an upgrade of the programming language from Ada83 to Ada95. The change in the architecture led to additional platform-specific customization options in the LSTM, further increasing the variability of the code significantly.

> "There were some minor complications, but they were relatively easy to fix. It worked on the host [simulation computer], but it didn't work on the target [avionics board], because the host compiler didn't change. The fix was done within a day, but if you start adding the initial analysis to locate the error and the tests on the target [avionics board], you can figure 2 weeks. I would say that is very fair."

The implementation was limited to the generic package for lists, which avoided further variant-specific modifications.

Memory optimization: Over time, more developers of *Operating Processing Functions* used the LSTM as central data-management component. The source code was constantly extended to support further entity types (i.e., new lists), which raised a problem related to resource consumption.

> "Memory was an issue on the first computers. They have 8MB and you have to get data and executable code in there. Actually the lists didn't take up that much code, but it was an issue [...] with all the communication lists and all the communication related Operational Processing Functions."

This issue has been addressed with a clone-and-own approach. Some of the lists belong to Operational Processing Functions that realize internal control applications, which handle only private data. These data are never printed on graphical displays or transferred to other devices. However, due to the implementation as generic package, each list allocates data structures for data-transfer functionality, regardless of whether used or not. Ultimately, a lesser memory footprint could be achieved by creating a slimmed-down copy of the original LSTMs code, in which data structures and functionality related to data transfer have been removed. The implementation required approximately 2 person months.

Navigation list: In 2012, the replacement of the data-management component was initiated. The most important argument emerged from one engineering change request:

> "There was one big effort when the navigation list came, because there we had to program all the links which are not in the list generic."

The mapping of the navigation data to the LSTM was difficult. It requires multiple entity types that relate to each other. For example, routes are basically sets of route points, which are in turn defined by different types of guidance points, such as airports or hospitals. The architectural design of the LSTM was never intended to support such complex data dictionaries. This caused considerable complications regarding data storage and access, since there is no mechanism for maintaining referential integrity. To solve this problem, a separate wrapper was added to the LSTM. It enriches the navigation lists by a proper access interface, which controls data manipulation in terms of consistency and integrity constraints.

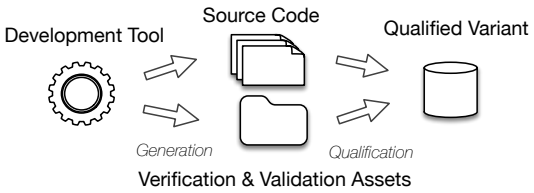

Fig. 3: Asset-based development of qualified avionics software

> "[...] and this is where an effort came with another year to implement. It is maybe the set of requirements that is difficult. You have to re-think your logic and you have to re-implement this relationship."

Due to the rigorous programming constraints for mission-critical avionics software, the code of the solution is strictly tailored to the specific use case. It cannot be re-used in different scenarios (i.e., for varying entity types).

G. Key Insights

- The development of qualification assets for the LSTM was expensive, adding about 8.5 person months to the total implementation time of 18 person months (45% increase).
- A major maintenance effort arises from the realization of new functional- and non-functional requirements.
- Long-term stability of system requirements is unlikely. Customer requests and new equipment hardware demand constant modifications.
- The severity of new system requirements varies considerably. The technical realization of severe requirements demands up to 12 person months of additional development time, which is an increase of 45% compared to the initial effort for implementation and qualification of 26.5 person months.

Based on these findings, we started working on an appropriate solution for data management in the NH90 avionics software.

IV. Design Decisions and Integration

Rigorous non-functional requirements in combination with strict coding constraints impose a challenge for a single qualifiable implementation of the EDB — the new data-management solution. These requirements state that the source code of a variant must be tailored to correspond exactly to the specific equipment configuration and the accessory hardware–software co-design of the target system. A *variation point* is a location in the software that differs in individual variants of the system. In the case of NH90's data management, variation points arise from various low-level functional requirements, such as the presence of a table or column with a specific size or data type. Accordingly, there are many possibilities to derive EDB variants. As a consequence, an appropriate technical realization requires a sophisticated re-use strategy.

Next, we compare the two development approaches established in the avionics domain regarding their applicability to multi-variant software systems. In addition, as a core contribution, we introduce *asset-based* development as a novel approach.

A. Assessment of Established Development Approaches

The practicability of a development approach depends on the combination of implementation, qualification, and maintenance effort. To decide whether to use code-based or tool-based development (see Section II), we compare their trade-offs regarding initial effort and effort per variant regarding these three development activities. We define the initial effort as the effort required for setup and development of the initial variant.

In *code-based* development, the initial effort boils down to the manual implementation and qualification of the first variant. The approach demands substantial additional budget and schedule to develop and maintain subsequent variants by individually adapting or re-implementing source code and qualification assets. Although the number of software variants is relatively small (currently 45), multi-variant software systems based on code-based development are hard to grow, since each individual variant must be qualified and maintained. There is no automatic process to obtain certification credit for unchanged parts of the another software variant. However, some qualification assets can be re-produced (e.g., the results of the unit tests of unmodified code).

Tool-based development relies on tool support. The underlying generative approach increases quality of the software and facilitates implementation by generating variants. As no further assets are required, the costs for deriving additional variants are reduced to the specification as input for the generator tool. The disadvantages are huge initial expenses for either purchasing a pre-qualified commercial tool or developing and qualifying an appropriate tool in-house.

From an economic point of view, it turned out in our interview that both code-based and tool-based development are impractical for the technical realization of the EDB at Airbus Helicopters. Regarding project parameters, such as the number of variants, degree of variability, and potential artifact size, the cost–benefit ratio is unsatisfactory. To overcome these problems, we propose asset-based development.

B. Asset-based Development

The key idea of asset-based development is to combine code-based and tool-based development, such that we keep the initial effort feasible, while reducing the effort of maintaining and qualifying a growing number of variants (Figure 3). To this end, we use model-based and product-line techniques to generate implementation and qualification assets that are used for the code-based qualification process. That is, instead of qualifying the code generator, as in the tool-based approach, we generate assets that substantially reduce the effort for qualifying the generated source code. Following a model-based approach, we create a declarative system specification, which is then automatically transformed by a non-qualified generator tool to software artifacts and supplementary qualification assets, such that they are amenable to requirement-based testing in terms of DO-178. This includes the automatic generation of functional tests, unit tests at 100% statement coverage, and traceability evidence to low-level system requirements. These qualification assets are intended to significantly facilitate qualification for each variant. This way, we obtain benefits from the generative approach, such as a reduced time-to-market and software quality, while avoiding the high initial costs for either purchasing or developing a fully qualified tool in-house. The asset-based approach does not violate aerospace standards and is fully compliant with DO-178 (civilian aviation) and DOD-STD-2167 (military aviation).

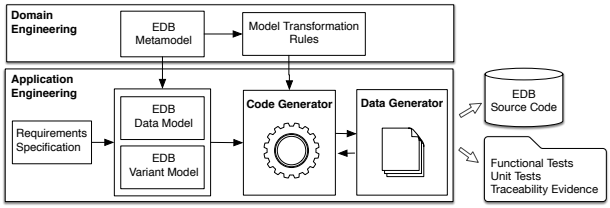

Fig. 4: Overview of the EDB development process

C. Implementation

As illustrated in Figure 4, our approach follows the classic product-line process [11], dividing the development into domain and application engineering. Since data management is used by developers from various backgrounds and divisions, we decided to use a specification and configuration mechanism that is expressive and easy to learn. To this end, we created a domain-specific language in the form of a set of SysML metamodels (domain engineering) [12]. It is used to define a descriptive system specification of a family of EDB variants, consisting of a data model to determine the storage structure and a variability model to define equipment-specific variations. During application engineering, the stakeholder creates appropriate SysML models according to her requirements (i.e., equipment and hardware specifications). Thereupon, a corresponding model transformation is invoked, and a code generator instantiates the model-transformation rules, based on source-code templates, with information extracted from the given SysML models. To compose the source code of the target EDB variant, the code generator supplements prefabricated database parts with glue code and the resulting code components of the model transformation.

To facilitate qualification, unit tests, functional tests, and traceability evidence are derived from the declarative system specification on the basis of generic test cases, also implemented as source-code templates. The tests require a consistent and integral database instance of the target EDB variant to operate on. Therefore, the code generator is connected to a data generator, which receives a normalized relational database schema from the data model as input. Then, the data generator instantiates the schema with randomly generated values according to a predefined distribution with respect to foreign-key integrity and type-specific value ranges (e.g., -90.0 to +90.0 for the type `Latitude`). The resulting tuples are passed to the code generator, which is now aware of a consistent and integral database instance. Finally, the source code to fill a test instance as well as associated functional tests, unit tests, and traceability evidence are generated.

In terms of tooling, we decided to use open-source software only. It is required by law that the manufacturer of an aircraft is capable to maintain the software across the entire life cycle. Mainly due to its very long-term support, we used PolarSys [13], an open-source tool suite for embedded, model-based, mission- and safety-critical systems. We applied PolarSys to cover all tasks of domain engineering and application engineering. With the modeling component Papyrus, we specified the domain-specific language as well as an initial instance

of a data model and a variability model. We implemented the model-transformation rules using the component Acceleo, which requires OCL [14] as meta-language for the source-code templates. Ultimately, we created 24 templates for 12 logical components of the EDB, 9 templates for test cases, and 2 templates to generate test instances. This way, we generated fully qualifiable EDB variants in the programming languages Ada83 and Ada95. Overall, our approach relies on standard techniques, but combined in a novel way to face the data-management challenges in the avionics domain.

A generated variant of the EDB is already integrated as the new data-management component in the software of the *Mission Tactical Computer* of the NH90. Mainly, it handles complex data related to area navigation and radio communication.

V. Discussion and Perspectives

In this section, we compare code-based and asset-based development by means of the example of the LSTM (old data-management component) and the EDB (new data-management component). To this end, we systematically review the evolution using the real-world scenarios reported in the developer interview (see Section III). We categorize the scenarios according to their severities. For each of the categories, we describe the proceedings for the technical realization of the subject, once in context of the LSTM and once in context of the EDB. This way, we discuss weaknesses and strengths of code-based and asset-based development. Since there is no comparable data-management component implemented with tool-based development, an analysis of the tool-based approach remains subject to further work.

A. NH90 Maintenance Data

To learn about the frequency of maintenance and evolution tasks regarding data management, we collected maintenance data of the NH90 avionics software from two sources. First, we accessed the software repository of two core avionics computers of the NH90. Second, we analyzed the source-code files of the data-management component of the same two avionics computers. The software repository contains entries of over 18 years, from 1996 until 2014. For each entry, there is a short description of the issue, the current status, the date of creation, and the date of the last modification. The data set is divided into three subsets:

- *Problem Report* (PR): describes a situation in which a software component is not working as intended. PRs are initiated by engineers, managers, or software developers to inform project members (including customers) about problems in the source code. Typically, PRs are raised if a problem is of inter-divisional interest, for example, if it occurs on the software test bench or at test flights.
- *Engineering Change Request* (ECR): describes a request to adjust the software. ECRs arise from new system requirements, feature requests from customers or developers, and changes in the aerospace standards.
- *Software Change Note* (SCN): documents a software modification at a technical level. SCNs are linked to the causing PRs or ECRs in a many-to-many relationship.

As a second source, we examined procedure headers of the source-code files of the same two avionics core computers. The headers are auto-generated by the version-control system and include a full revision history of the target procedure. We parsed 10,768 source-code files, containing 1,812,289 procedure headers with 24,907,787 revisions in total. In 42.7% (10,647,287) of the annotations, we were able to extract a reference to an SCN, which relates these revisions to maintenance and evolution. The remaining revisions are related to progress in the development process. We transformed the data in both repositories into a unified format, stored them in a database, and joined them on the SCN identifier. This way, we enriched the data in the software repository with file-level information. By means of the folder structure of the source-code files, we were able to assign entries in the software repository to logical software components. Ultimately, we identified PRs, ECRs, and SCNs related to data management. We start our discussion with a comparison of initial effort of LSTM and EDB.

B. Comparison of Initial Effort

From the developer interview, we learned that the initial effort of implementing and qualifying the LSTM amounts to 26 person months, which is about 4,160 person hours. For the EDB, we estimate 4,848 person hours in total: 3,736 person hours development time and 1,112 person hours for adaptation, deployment, and integration at Airbus Helicopters. The majority of effort required to develop the EDB was spend by the first author on the implementation of model-transformation rules (i.e., source-code templates). At first, we experienced a significant overhead in programming in OCL. This overhead decreased rapidly with growing experience, such that we could implement templates in OCL as fast as source code in Ada. In this context, asset-based development involved a 17% higher initial effort due to the integration and familiarization of development tools.

In the next four subsections, we describe exemplary events and examine their effects on the evolution of the data-management component of the NH90 from the perspective of both systems. The discussion aims at establishing a descriptive and explorable model to assess and identify factors that influence maintenance and evolution in the avionics domain. All figures are meant as an illustration of the events and their corresponding effects, not for quantitative prediction.

C. Deriving Software System Variants

To derive a new variant of the LSTM, the set of supported entity types has to be tailored manually by removing, changing, or adding lists in the source code. As reported in the developer interview, one extension demands about 8 person hours of development time. By contrast, the EDB does not demand any modifications in source code. Each extension is performed in the declarative system specification (i.e., in the SysML models). To remove, change, or add an entity type, corresponding model elements have to be added to the data model and applied with stereotypes configuring variant-specific properties. As we first created a new data model similar to the entities of the LSTM, it required 2 person hours of development time. Naturally, this value depends on the size and complexity of the data schema and may vary in different scenarios. The model transformation to obtain the source code of the target variant is performed within seconds.

Figure 5 shows the extrapolation of development time

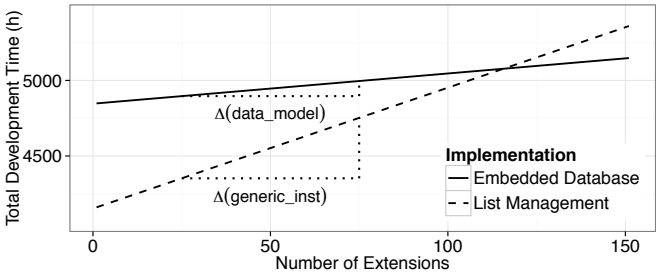

Fig. 5: Effort to modify the set of entity types

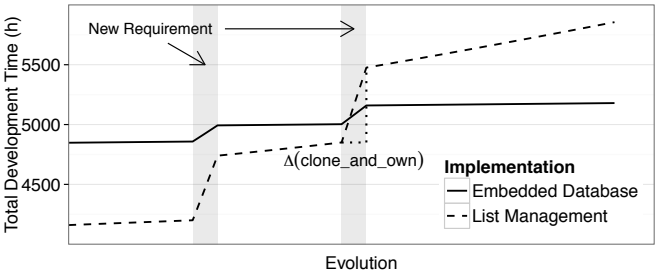

Fig. 6: Effort required to realize evolution tasks by clone and own

to derive a new variant of the LSTM and of the EDB. The y-axis depicts the total time for providing source code and qualification assets in person hours. The x-axis covers the number of extensions in the set of entity types. The values of both systems start with an offset for the initial effort and grow in relation to the required development time. The ratio is illustrated as $\Delta(\texttt{generic\_inst})$, in the case of the LSTM, and $\Delta(\texttt{data\_model})$, in the case of the EDB. In this scenario, code-based development outperforms asset-based development for a long time. We assume that the break-even point will not be reached until the end of the life cycle of the NH90. Up to now, the LSTM was extended to support 30 additional entity types.

D. Code Re-Use with Clone And Own

The developers at Airbus resolved two issues in the life cycle of the LSTM by applying *clone and own* as a re-use strategy. The scenario *memory optimization* from the developer interview (see Section III) falls into this category. The approach was used to deploy a solution to a subset of the instantiated lists. For this purpose, the generic package of the LSTM was copied and adapted to satisfy the requirement.

The clone-and-own approach has two major drawbacks, which we illustrate in Figure 6. The y-axis depicts the total development time for implementation and qualification, with an offset for initial effort. The x-axis represents progress in evolution. The grey areas illustrate the two phases where a new requirement was realized using clone and own.

As a first drawback, in addition to the effort required to perform a code change, copied source code must also be qualified, which involved about 320 person hours for each case in the LSTM. These efforts are illustrated with $\Delta(\texttt{clone\_and\_own})$ as sharp increase in development time. For the EDB, the effort is reduced to perform the actual code changes. Contrary to code-based development, asset-based development transforms model elements to source code. This way, we exploit the generative approach to add or remove functionality by integrating conditional units directly into the generic application logic (i.e., introducing new variation points), instead of adapting a code clone manually.

Regarding *memory optimization* as a what-if scenario in the context of the EDB, unnecessary data structures or procedures would be encapsulated as conditional units, which would be present in the target source code only if the functionality is included in the declarative system specification of a desired variant. Due to the automatic generation of unit tests and functional tests, adjustments in terms of qualification assets are reduced to the specification of generic test cases. Whenever functionality is removed, as in *memory optimization*, no additional effort for qualification is required. Overall, we approximate the total development time for the technical realization of this scenario for the EDB at about 80 person hours.

As a second negative effect of using a clone-and-own approach, we identified increased costs related to maintenance and evolution after cloning. This situation occurs if a PR or ECR concerns code clones. Let us assume, we found changes that were performed in original source-code files as well as in two clones A and B. Analyzing the software repository of the NH90 in this hindsight, we found 14 intersecting SCNs in the original system and clone A, 5 intersecting SCNs in the original system and clone B, and no intersecting SCNs in clones A and B. We identified such additional effort in 19 cases. We added up the time intervals of these intersecting SCNs and determined the ratio to the sum of the time interval of all SCNs in the original system (74 found). This way, we estimate the overhead in development time for the 19 intersecting SCNs related to the clone-and-own approach in the LSTM at 24%. We illustrate this effect as an increased slope after cloning in Figure 6.

In summary, asset-based development improves maintenance and evolution in terms of code re-use regarding two aspects: It reduces the effort required to realize changes with conditional units and code generation, and it entirely eliminates the drawbacks arising from using a clone-and-own approach.

E. Software Changes in Generic Parts vs. Specific Parts

So far, we have considered only non-severe requirement changes, which were realized in the LSTM by means of a modification of the generic package, as in the scenario *hardware platform* (see Section III). Severe requirement changes, as in the scenario *navigation list* (see Section III), however, require a specific solution. To explore the factors of maintenance and evolution effort related to changing generic and specific parts of the source code, we analyzed the implementation of the LSTM in this regard. As generic part, we considered each line of source code that is re-used either by parametric polymorphism in a generic package or by clone and own. We assume that the remaining lines of code constitute specific functionality, which consider the specific part.

Overall, we parsed 57,181 revisions in the generic part of the LSTM. 21,907 (38.3%) of the revisions contain an SCN and thus are considered as relevant for maintenance and evolution. In total, we identified 90 SCNs, which are related to 94 PRs and 10 ECRs. The effort for these tasks is basically the

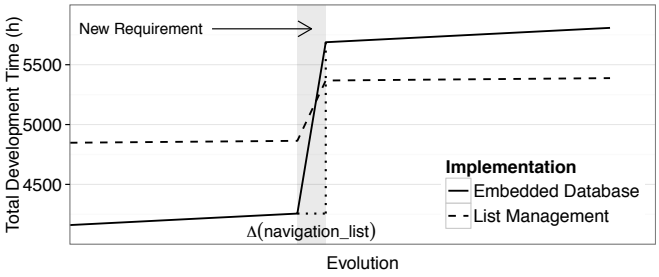

Fig. 7: Effort required to realize a severe evolution task

same for code-based and asset-based development, since the technical realization affects the generic part of the software and the qualification assets to equal shares (i.e., generic packages of the LSTM and source-code templates of the EDB).

Analogously, we analyzed the specific part of the LSTM. It contains 165,278 revisions in total, where 83,023 (50.2%) are relevant for maintenance and evolution. In total, we identified 536 SCNs, related to 474 PRs and 97 ECRs. With 6 times more SCNs targeting specific parts, our data indicate a considerable increase in occurrences of software changes related to maintenance and evolution than in the generic part. Regarding the number of revisions, which has increased by a factor of 3, the frequency of software changes in the specific part of the LSTM is doubled compared to the generic part. In particular, there is an exceptionally higher number of ECRs (almost factor 10). This shift shows the increased significance of realizing new requirements in specific variants of the considered software. Next, we discuss the scenario *navigation list* from the developer interview as representative example for the severity of changes of this kind.

The interviewees explained that the navigation-list requirement was realized in the LSTM by implementing a wrapper around the instantiation of navigation lists with an additional interface for data access, to maintain referential integrity and consistency. For the EDB, this scenario required the adaption of the domain-specific language to specify a model element that relates entity types in the declarative system specification. To this end, we used instances of *SysML::Association* to link *SysML::Blocks* — the model elements that represent entity types in the data model. We realized this complex requirement by implementing conditional units that were integrated in the source-code template for query processing.

Regarding qualification, we defined a generic test set that covers the additional functionality. We integrated it into the same template to automatically generate suitable test cases, amenable for requirement-based testing. Finally, these test cases have been automatically instantiated with valid data from the data generator, to create actual unit tests and functional tests compliant to DO-178.

Figure 7 compare the development time required to realize the scenario *navigation list* in the context of the LSTM and the EDB. The axes are as in Figure 6. The grey area illustrates the phase in which a new requirement arose. The effort for realizing this severe requirement in the LSTM is illustrated with $\Delta(\mathsf{navigation\_list})$ as a stark increase in development time. The developers of the LSTM estimated the effort at about 12 person months in total, that is 1,432 person hours. The ratio of required effort associated to qualification is the same as for new developments, which was estimated in the interview at about 45%. In the case of the EDB, we estimate the development time to 504 person hours, where 24 person hours are spent on extending the domain-specific language, 400 person hours to develop model-transformation rules, and 80 person hours to define and implement the set of test cases.

In summary, asset-based development can reduce the effort and cost for implementation and qualification compared to code-based in a severe scenario, such as *navigation list*, by almost two third, which may compensate the higher initial effort that is required by asset-based development.

F. Lessons Learned

Answering the question of whether code-based, tool-based, or asset-based development is the most suitable approach for a specific subject matter depends on an estimation of the initial effort and a detailed assessment of potential tasks related to maintenance and evolution. In safety-critical application domains, where the software has to pass a qualification process, strict coding constraints often demand a tailored solution for a specific problem. Our analysis demonstrates the potentially high severity of maintenance and evolution tasks in these domains. Real-world scenarios considering severe new requirements have shown that the technical realization requires up to one third of the initial development cost. However, the efficiency of a development approach depends on the parameters of the target system. If the system requirements are stable or changes are very unlikely, code-based development may still outperform asset-based development. In cases where maintenance and evolution is likely, such as in the data-management of the NH90, asset-based development appears to be an efficient and practical development approach. Regarding tool-based development, further studies are required to explore its application in software systems, in which maintenance and evolution contribute to a majority of the effort.

VI. Threats To Validity

Our approach to assign source-code files to logical software components based on the folder structure might unintentionally exclude files, which threatens internal validity. It is possible that semantically related source-code files are located in different folders. The PRs, ECRs, and SCNs of these files might have been mismatched. However, through personal communication with a senior developer who is familiar with the system as a whole, we have been informed that the source files of the LSTM as subject matter of our study are well encapsulated and not scattered over multiple folders or packages.

A second threat to internal validity arises from our assumption that our data basis is free of defective entries. After manually reviewing the textual description of all affected entries (PRs, ECRs, and SCNs), we could not find any duplicates, but we cannot guarantee that the repository does not include invalid items (e.g., PRs that could not be replicated).

Regarding external validity, one could argue that insights gained from an interview of two developers cannot be used to justify general statements. Nonetheless, one of the interviewees is the head developer of the data-management component and has, much like the second interviewee, decades of experience in the field of avionics software engineering. Furthermore, their

approximation of the ratio of effort for implementation and qualification of the LSTM, as case example of a complex avionics software, agrees with the results of previous work [15] (coding makes up 16%, where unit tests, functional tests, and design reviews sum up to 9% of the total development costs of military avionics software).

VII. Related Work

As a first systematic report on achieving benefits from adopting product-line technology in complex avionics software, Sharp suggested a logical pattern-driven design to facilitate the integration of predefined components in the avionics software in the context of the Bold Stroke Initiative at the Boeing Company [16]. A similar approach was proposed by Ganesan et al. at the NASA Goddard Space Flight Center. The authors use a layered architecture to re-use modules, configurable for mission-specific needs in the Core Flight Software (software platform for NASA missions) [17]. Both approaches rely on product-line engineering to build hardware- and mission-specific variants of the avionics software. The authors focus on exploring domain and application design rather than facilitating the development process by addressing the challenges of qualification that prevail in the avionics domain.

Regarding the application of model-based methods in the aerospace industry, Batory et al. suggest GenVoca, a domain-independent model for hierarchical systems as compositions of reusable components, to create a reference architecture for avionics software synthesis [18]. Similar to the work of Sharp and Ganesan et al. the authors present lessons learned in applying model-based techniques to improve the application design, but do not target qualification. Hovsepyan et al. report on an enhancement of the development life cycle at Space Applications Services by means of a model-based development process to establish traceability of system requirements across the phases of the V model [19]. In the work of Dubois et al. at the Thales Group, model-based methods have been applied in the form of a domain-specific language in SysML to support domain engineering in context of product-line engineering in the CEASAR project [20]. Analogous to Hovsepyans work, Dubois et al. used models for maintaining traceability of system requirements to application components, but they do not target code generation or qualification. Delange et al. [21] propose model-based engineering to capture architecture requirements using the AADL modeling language. Their approach exploits AADL models to validate non-functional requirements, such as resource dimensioning through simulation, but they did not consider qualification obligations.

There are model-based and generative approaches to develop software systems based on the ARINC 653[6] specification [22]. Horvth et al. [23] discuss model-driven engineering as an approach to systematically develop configuration data. A similar strategy is described by Choi et al. [24]. They present a tool to generate XML configuration files for an ARINC 653 compliant operating system. In contrast to our approach, the authors apply code generation to derive configuration assets, not qualification assets.

With Matlab Simulink[7] and Esterel SCADE[8], there are two commercial products that can be considered as the de facto standard tool set for tool-based development. Both provide model-based design and code-generation capabilities. Esterel SCADE was successfully applied at Airbus to develop the software for the flight control computers of multiple aircrafts, such as the Eurocopter EC 135/155 and the Airbus A340/500 [25]. The share of automatically generated code was 70%, which reduced the change cycle time by a factor of 3 to 4 compared to manual coding. Both projects showed compliance to Design Assurance Level A. However, these tools aim at generating source code for the most critical components of a system and require a model-based specification at a very low level of abstraction, which is inappropriate to handle complex data-management functionality.

Bridges et al. and Dorodwsky et al. describe the successful application of a generative approach at Airbus Helicopters without applying commercial tools [5] [26]. They use tool-based development based on an in-house tool chain based on a fully qualified code generator. It produces software artifacts for low level components of the NH90 System Software, such as the real-time tasking model, processor allocation, and raw data I/O. The authors identified qualification as key factor, but did not provide any details on the process.

VIII. Conclusion

In our experience report, we described the enhancement of the data-management component of the NH90 helicopter at Airbus Helicopters. In a semi-structured interview with two professional developers, we gained deep insights into the industrial practice of developing qualified avionics software. We found that established development approaches are not feasible to cope with the challenges of realizing data management in such an highly sensitive environment.

To address problems regarding implementation and qualification effort, especially in the face of software evolution, we propose asset-based development as a novel generative approach to develop avionics software. Using model-based and product-line technology, we successfully implemented the *Embedded Database*, a software product line and a generator infrastructure to generate fully qualifiable variants of real-time embedded database management systems, which is already integrated in the NH90 development environment to replace the former system.

By a systematic comparison of the former data-management component and our solution, we identified significant improvements when using asset-based development, in terms of flexibility and efficiency of realizing tasks related to maintenance and evolution. Discussing a number of real-world evolution scenarios with data collected from the NH90 software repository and the developer interview, we demonstrated the likelihood and the potential severity of these tasks, and we identified the main cost drivers of evolution. Finally, our results are not restricted to data management in the avionics domain and may help developers to choose a suitable development strategy in domains where qualification plays a key role.

IX. Acknowledgment

This work has been supported by the German Research Foundation (AP 206/4, AP 206/5, AP 206/6).

[6] A standard for space and time partitioning in avionics operating systems

[7] http://de.mathworks.com/products/simulink/

[8] http://www.esterel-technologies.com/products/scade-suite/

REFERENCES

[1] *DO-178C/ED-12C: Software Considerations in Airborne Systems and Equipment Certification*, RTCA/EUROCAE Std., 2011.

[2] E. Thomas, "Certification Cost Estimates for Future Communication Radio Platforms," Rockwell Collins Inc., Tech. Rep., 2009.

[3] V. Hilderman, "DO-178B Costs versus Benefits," Consunova Inc., Tech. Rep., 2009.

[4] F. J. v. d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[5] R. Bridges, "NH90 Helicopter Avionics Systems from the 1990s to 2010 and beyond," in *Proc. Workshop on Software-Architekturen für Onboardsysteme in der Luft- und Raumfahrt*. Deutsche Gesellschaft für Luft- und Raumfahrt, 2007.

[6] *DOD-STD-2167A: Defense System Software Development*, Department of Defense Std., 1985.

[7] *DO-333/ED-216: Formal Methods*, RTCA/EUROCAE Std., 2011.

[8] W. Wong, A. Demel, V. Debroy, and M. Siok, "Safe Software: Does It Cost More to Develop?" in *Proc. Int'l Conf. Secure Software Integration and Reliability Improvement (SSIRI)*. IEEE CS, 2011, pp. 198–207.

[9] *DO-330/ED-215: Software Tool Qualification Considerations*, RTCA/EUROCAE Std., 2011.

[10] A. Jedlitschka and D. Pfahl, "Reporting Guidelines for Controlled Experiments in Software Engineering," in *Proc. Int'l Symposium on Empirical Software Engineering (ISESE)*. IEEE CS, 2005, pp. 95–104.

[11] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[12] *OMG Systems Modeling Language (OMG SysML), Version 1.3*, Object Management Group Std., 2012.

[13] G. Blondelle, P. Arbaret, A. Rossignol, B. Lundell, C. Labezin, R. Berrendonner, P. Gaufillet, R. Faudou, B. Langlois, L. Maisonobe, P. Moro, J. Rodriguez, J. M. Puerta Pena, E. Bonnafous, and R. Mueller, "PolarSys: Towards Long-Term Availability of Engineering Tools for Embedded Systems," in *Proc. European Congr. Embedded Real Time Software and Systems (ERTS)*. ERTS, 2012, pp. 1–8.

[14] *OMG Object Constraint Language (OCL), Version 2.3.1*, Object Management Group Std., 2012.

[15] C. Jones, "Software Cost Estimation in 2002," *CrossTalk: The Journal of Defense Software Engineering*, vol. 15, no. 6, pp. 4–8, 2002.

[16] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proc. Int'l Digital Avionics Systems Conf. (DASC)*. IEEE CS, 1998, pp. 1–8.

[17] D. Ganesan, M. Lindvall, C. Ackermann, D. McComas, and M. Bartholomew, "Verifying Architectural Design Rules of the Flight Software Product Line," in *Proc. Int'l Software Product Line Conference (SPLC)*. ACM Press, 2009, pp. 161–170.

[18] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, "Creating Reference Architectures: An Example from Avionics," in *Proc. Symposium on Software Reusability (SSR)*. ACM Press, 1995, pp. 27–37.

[19] A. Hovsepyan, D. V. Landuyt, S. Beeck, S. Michiels, W. Joosen, G. Rangel, J. F. Briones, and J. Depauw, "Model-Driven Software Development of Safety-Critical Avionics Systems: An Experience Report," in *Proc. Int'l Workshop on Model-Driven Development Processes and Practices*. CEUR Workshop Proceedings, 2014, pp. 28–37.

[20] H. Dubois, V. Ibanez, C. Lopez, J. Machrouh, N. Meledo, and A. Silva, "The Product-Line Engineering Approach in a Model-Driven Process," in *Proc. European Congr. Embedded Real Time Software and Systems (ERTS)*. ERTS, 2012, pp. 1–9.

[21] J. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon, "Validate, Simulate, and Implement ARINC653 Systems Using the AADL," *Ada Lett.*, vol. 29, no. 3, pp. 31–44, 2009.

[22] *Avionics Application Software Standard Interface - ARINC Specification 653 - part 1*, AEEC Std., 2006.

[23] A. Horvath, D. Varro, and T. Schoofs, "Model-Driven Development of ARINC 653 Configuration Tables," in *Proc. Digital Avionics Systems Conf. (DASC)*. IEEE CS, 2010, pp. 1–15.

[24] E.-T. Choi, O.-K. Ha, and Y.-K. Jun, "Configuration Tool for ARINC 653 Operating Systems," *Journal of Multimedia and Ubiquitous Engineering*, vol. 9, no. 4, pp. 73–84, 2014.

[25] J.-L. Camus and B. Dion, "Efficient Development of Airborne Software with SCADE Suite," Esterel Technologies S.A.S., Tech. Rep., 2003.

[26] F. Dordowsky, R. Bridges, and H. Tschope, "Implementing a Software Product Line for a Complex Avionics System," in *Proc. Int'l Software Product Line Conference (SPLC)*. ACM Press, 2011, pp. 241–250.