Toward Efficient Model-Based Development of Aerospace Applications

Isaac Amundson, Lyle Shipton, Anshuo Liu, and Michael Nowak
Eaton Corporation
LyleShipton@Eaton.com

***Abstract*— Developing embedded systems for the aviation industry presents numerous challenges, particularly with respect to design assurance and verification. To address these challenges, we work to continuously improve the tools, methods, and processes used to develop our products. In this paper, we present our experience and lessons learned in adopting a model-based development process for aviation component development. It is our hope that other product design teams considering model-based development can benefit from the initial knowledge we gained by undertaking this endeavor.**

I. Introduction

Developing embedded components for the aviation industry presents numerous challenges, particularly with respect to design assurance and verification. Developers must navigate vague or incomplete customer needs, ambiguous requirements, evolving development processes, and regulatory procedures, all while customers demand increasingly smarter, more complex systems. Ultimately, a dependable product – on time and on budget – is the desired outcome, and so it is up to the development team to understand and utilize the best available practices, tools, techniques, and methodologies to accomplish this.

Model-based development (MBD) has been in use in the aviation industry for several years now, and early on was primarily used for rapid design prototyping and simulation. More recent MBD efforts have focused on code generation and design assurance and verification using formal methods. The use of MBD in the product development process has been shown to reduce development time, risk, and cost by, among other things, identifying and addressing faults early in the development cycle [1], [2], [3]. This is illustrated in Figure 1, in which the effort in each development phase is significantly reduced when applying MBD. This is especially true for testing.

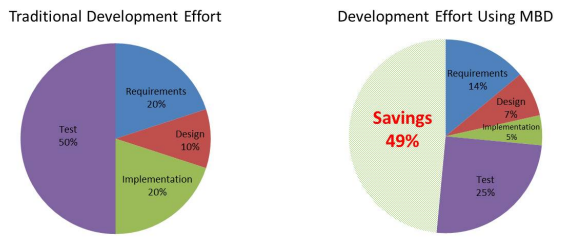

Fig. 1. Development Effort Comparison

Figure 2 shows typical relative costs of removing software faults based on the development phase in which they are detected [4]. The exponential cost increase as the product makes its way through development is expected because the longer a fault exists in the component under development, the greater the opportunity for its effects to propagate. It is therefore desirable to remove these systemic faults as close to their introduction as possible. MBD is a leading methodology for realizing such assurance in design.

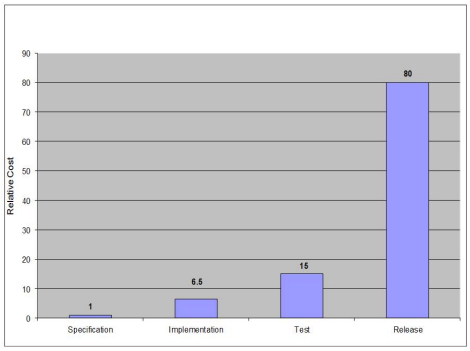

Fig. 2. Relative Cost of Removing a Fault during Product Development

Commercial tools are now available to enable even small development organizations to take advantage of MBD. However, adapting these tools to an established development process is not trivial. Systems development, in which large parts of the system may be unknown or undefined, complicates matters because requirements flow-down and interface definition often must undergo several iterations of refinement.

In this paper, we describe our experience of adopting an MBD methodology for developing software-integrated aircraft components. Because MBD scope varies based on project and organization, and because full MBD capability and maturity cannot be realized overnight, we describe here our initial steps on the path of achieving a comprehensive MBD process for new product development. We specifically make the following contributions to software-integrated component developers in the aircraft industry:

- We describe how we integrated modeling techniques into our standard development process. This includes all development phases starting with requirements and flowing through design, implementation and test.
- We present organizational lessons learned. These include observations regarding the adoption of MBD at an or-

ganizational level. In order for a new methodology to take hold and be successful, personnel at all levels of the organizational hierarchy from design and test engineers to senior management must understand it and see its value.
- We present MBD practices that we found added value. These include solutions to issues we encountered when integrating MBD into our development process that lacked a practical solution in the literature. We believe many of the issues we encountered are indeed common in the industry, and other groups looking to adopt MBD may benefit from our findings.
- We describe MBD pitfalls that we will avoid in the future.

The remainder of the paper is organized as follows. Section II describes our model-based development approach. We present our experiences, lessons learned and pitfalls in Section III. In Section IV, we provide our overall impression of MBD in our conclusion.

II. Model-Based Development Approach

In this section, we outline our approach to adopting model-based development, and describe aspects of how we aligned MBD with our current development process.

The decision to adopt MBD entails a certain amount of risk, and must be driven by organizational need [5]. There should be a clear understanding of the scope of MBD and the benefits of following MBD over other approaches. In other words, we do not want to model just to model. In our case, although we had identified activities in our development process that could benefit from the increased productivity inherent in MBD, our main motivation was customer-driven. We have had OEM agreements in which a model was requested as the deliverable, either an entire component design, or a black-box subsystem that could be integrated into an OEM's system model to facilitate their own MBD process (for example, enabling the OEM to start their own testing and integration work earlier). Such instances have not necessitated full-scale model-based development; however, they do help form a solid foundation from which to grow our modeling capability.

Figure 3 illustrates a roadmap for maturing our MBD process. Currently, we use modeling in a planned, but ad-hoc manner throughout the company. Different groups use modeling for different activities, but it varies greatly by project, and is not used at all on many projects. As we started to realize additional benefits of modeling (i.e., risk reduction, increased productivity, etc.) we became interested in moving toward a more defined process with a standard set of activities and tools, and the skill sets to use them. Ultimately, our vision is a development process that is highly optimized, utilizing the best practices, tools, and methodologies for the project. Model-based development appears to play a part in that process for years to come, but we understand that we cannot migrate from our current process overnight.

We must therefore continuously redefine the scope of MBD as we improve the way we develop products. For an organization that is just embarking on MBD, limiting the scope can greatly increase the chances of project success. On the other hand, models can often be used to satisfy multiple development objectives, thus increasing value. Modeling objectives

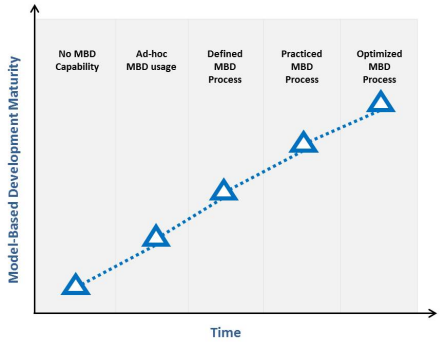

Fig. 3. MBD maturity roadmap

can typically be divided into three main categories: 1) analysis, 2) communication, and 3) synthesis [6]. Analysis activities are those that enable us to reason about the system design, and include, among others, characterization, optimization, verification, simulation, and in-the-loop testing. Models can also substantially simplify communication between parties involved in the design process. Because models are formal, unambiguous representations that are specified using a well-defined syntax and semantics, they are nicely suited for use in design reviews, and also as a source of documentation. Finally, high-fidelity models can be used to generate an implementation, such as c code or VHDL.

In our traditional software development process, high-level requirements were shared with the software development team, which was responsible for writing the corresponding software requirements. Software engineers would then write code intended to meet the software requirements, and a software review process was used to ensure the code satisfied the intent of the high-level requirements. With our MBD approach, rather than diving straight into the implementation by writing code, we create behavioral models in Simulink that satisfy the low-level requirements. This allows the development team to effectively simulate and test product functionality before spending significant time and resources on code development.

Although our development team has familiarity with modeling tools such as Simulink [7], modeling has not comprised a major part of system development to date. Early functional modeling endeavors entailed using Simulink to develop controller models for aircraft hydraulic power generation systems. Modeling enabled an iterative design-simulate cycle for assuring the correctness of the controller before implementation, but coding was still a manual process.

In subsequent projects, models were designed to reflect low-level controller requirements, which were traceable both to code and high-level system requirements and maintained in the Rational DOORS application [8]. Model traceability was established using the DOORS interface provided in Simulink. In some instances, the Simulink Verification & Validation Toolbox [9] was also used to aid in establishing traceability.

We are currently investigating the extent to which we can generate code from our models, having successfully done so on a small controller project. A key issue is whether or not a qualified code generator will be used. A qualified code

generator will produce code that does not need to be verified if formal verification testing is performed on the model. Unfortunately, qualified code generators are cost-prohibitive for organizations that have not reached an optimized level of MBD process maturity; it is likely less expensive to manually write the code. Therefore, our approach does not involve a qualified code generator. We mitigate any concern by manually merging the generated code with our baseline software, and perform review and verification testing on the code following the traditional process and complying with the DO-178B [10] guidelines.

For model verification, we developed an automated process using scripting and Simulink signal builders. For new product development efforts, we now must comply with DO-178*C* [11], along with its MBD guidance supplement DO-331 [12]. For certification of products developed under an MBD process, verification testing must be performed on the model. The verification can take multiple forms, such as review, testing, and simulation. Model coverage analysis is also required. Any tools used to perform this analysis must be qualified, according to DO-330 [13].

Looking to the future, we are very interested in development methods that enable us to drive down risk. Because we have our design formalized as a model, there is a lot of opportunity for early-phase analysis to help ensure our design is correct before moving on to the implementation. Formal methods can play a big role in such risk reduction. Formal methods are mathematical techniques for specification and verification of systems. There are several formal methods categories, but we are most interested in *model checking*. Because the design is formalized as a model, model checking tools can be employed to perform what is essentially an exhaustive execution of the state space to determine if there is any possible combination of input values that would cause the model to violate the requirements. Of course, these types of tools require a formalized version of the requirement as well.

Requirements are typically *textual*, written in a natural language (e.g. English) as a series of "shall" statements that must hold if the product is to perform its intended function. There are several drawbacks with the textual approach. A poorly written requirement will contain ambiguities, possibly resulting in an incorrect design. Even if requirement guidelines are closely followed, problems can still arise when multiple low-level requirements define a single complicated function. The interaction of these requirements could easily be lost to the reader. Additionally, English is not machine-readable, so there is no way to perform any software-assisted analysis on the requirements themselves.

There are several logical languages that can be used to formally state *unambiguous* requirements, such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). Another way to specify formal requirements is *graphically*, using a modeling language such as data flows (i.e., Simulink) or state charts (i.e., Stateflow [14]). In our work, we are exploring the use of textual requirements as our primary format, and creating formalized graphical requirements to use in parallel for design assurance. In this manner, traceability can still be established through the requirements interface provided in Simulink.

We piloted model checking on a project using the Reactis tool [15], and within minutes of running the tool we uncovered 35 model inconsistencies and three requirements violations on a pilot set of 12 requirements. In this case, the violations were also independently discovered by the development team, but only during a rigorous review. Such a fast and efficient means for validation provided an opportunity for system optimization much earlier in the design process.

At first glance it may seem that the additional steps of creating behavioral models and running model checking tools will increase development time and cost. However, MBD tools actually facilitate development in multiple ways. Models can accommodate early software, controller, and hardware-in-the-loop testing before final hardware or software design has been realized. A formalized design can then easily be translated into code using the code generation toolboxes that are available in the modeling environment. Automated code generation also eliminates fault introduction during the implementation phase and produces a consistent code base even if parts of the model were developed by multiple engineers. Tools such as Reactis can generate test suites that provide full coverage of metrics such as boundary and MCDC. Because testing often consumes at least 50% of development time, automated test case generation and execution provides a significant increase in productivity. The risk reduction inherent in MBD will also result in reduced development time and cost by reducing the number of failed tests and problem reports.

III. Lessons Learned

We categorize our lessons learned in the process of building our model-based development capability into four categories:

- **Development Process.** Our experience with adapting MBD tools and practices into our existing process. This was a major activity that required learning MBD tools and best practices, creating the modeling tool chain, and managing new process artifacts such as scripts, requirements, test cases, and documentation.
- **Organizational Lessons Learned.** A change in development process can have a substantial organizational impact, which needs to be managed appropriately.
- **Value-added Practices.** During the process of implementing our model-based framework, we compiled a list of practices and techniques that optimized our usage of MBD.
- **Pitfalls.** Similar to the value-added practices, we also compiled a list of pitfalls to avoid in the future.

A. Development Process Lessons Learned

As described in Section II, we performed modeling activities in product development as part of our traditional process. On the one hand, having experience modeling made it easier to move toward a formal MBD process. On the other hand, though, we experienced a sort of "model creep", where there was a tendency to expand usage of modeling, model analysis tools, and modeling synthesis tools, in ways that did not necessarily align with our development process. Although we

never deviated from our project development plans or the certification guidelines, we did experience cases where it was unclear how a particular tool or method fit into our process. It may also have contributed to development inefficiencies early on.

> Lesson: A defined MBD process and process migration strategy will greatly ease adoption of MBD.

As part of the MBD process definition, it is important to have a well-defined scope of what processes, methods, and tools are included in MBD activities. We found that practically every engineer and manager had a slightly (and on occasion drastically) different idea of what MBD was, and it is therefore vitally important to make sure the precise definition of MBD is known and accepted from the start. The definition of MBD is likely to change throughout process migration. It is tempting to use a superset definition that includes all existing MBD activities, but that causes confusion when claiming to follow an MBD process. It is therefore important to explicitly state the scope of MBD in the context of the current development process.

Setting the scope can be challenging. We want to derive the maximum value from MBD, but we do not want to bite off more than we can chew. A scope that is manageable but provides no value could cause MBD to be rejected by the development organization. Likewise, a scope that provides value but is not manageable could also cause MBD to fail. We must make sure we have the tools, competencies, process definition, and organizational support in place, or a plan for doing so.

> Lesson: Have a well-defined MBD scope that is manageable and can provide value.

When establishing MBD, it is unreasonable to completely discard the existing development process. Even if the majority of traditional procedures will eventually be replaced, this change cannot happen overnight without considerable disruption to project schedule and cost. The migration strategy should therefore be expressed in a manner that aligns MBD with the current process as much as possible, and gradually shifts at the speed of the organization's capability.

> Lesson: The migration strategy should align MBD to the current development process. Do not attempt to make major process changes on a single project.

It is difficult to discuss development process independently of certification guidelines. Although guidelines such as ARP4754A [16] and DO-178C intentionally offer a lot of freedom with respect to choosing how objectives are satisfied, they also constrain the process by including additional objectives if certain tools or methodologies are included. For example, in DO-178B, there was no guidance for modeling, so it was acceptable to create system models from which low-level software models were derived without having to perform explicit verification on the system models. That same approach would not be possible under DO-178C, and would require additional verification of the system model, which would add time and cost to the project schedule.

> Lesson: Do not invest in a specific modeling technology or methodology without first understanding the certification guidelines around it.

Consideration should be made as to the appropriate level of testing required. DO-331 mandates verification of models; however, there are various means that can be used to satisfy verification objectives. Automated test case generation tools can create test suites for the model that provide requirements coverage, as well as coverage for other metrics such as MCDC. In cases where there is a one-to-one mapping between the model elements and the code, or when the code is auto-generated, the model test cases can often be reused directly on the software. In our case, developing model-based test cases prior to implementation both expedited testing later in development and was necessary for work-share agreements with outside parties that were involved in the code generation process.

> Lesson: Creating test cases from the model that can be reused on the code can reduce overall test effort. It is important for the integrated product team to agree on the appropriate level of model testing early in the development process as testing can require significant time and resource allocation. Care must be taken to ensure that any tools used to generate the test cases are qualified.

In addition to planning test activities and creating test cases early in the development life cycle, test efficiency can also be achieved by eliminating the costly rework that results when tests fail. A failed test triggers a series of activities that involves finding root cause, designing and implementing a fix, regression testing, and re-running the test case that originally triggered the failure. If a small fraction of that effort was expended during the specification phase to assure the correctness of the design, the test would have passed, and no rework effort would be necessary.

Determining the best design assurance tools and methods is not trivial, because it is never clear how many faults we will be preventing from propagating into the implementation. We have been investigating the use of formal methods tools for gaining confidence in our design and driving down risk. Because we are not using these tools to satisfy verification objectives, DO-333 [17] does not apply.

> Lesson: In many cases early-phase design assurance methods reduce overall development time and cost by detecting and removing faults early in the development life cycle, eliminating costly late-phase rework effort.

B. Organizational Lessons Learned

Because every company has a different structure to their development organization, there is no single unified procedure for implementing an MBD process. When evaluating a new MBD technology, a clear up-front understanding is needed of where it fits in the MBD process, where the input life cycle data is produced and who the consumer of the output life cycle data is. Our development organization consists of several groups located around the world. Special attention is

needed to coordinate these groups with respect to day-to-day communication, configuration management, and tool licensing. These aspects need to be considered when migrating to MBD.

> Lesson: Understand how the organizational structure maps to the MBD process. A clear understanding of the development flow, with well-defined interfaces between development groups and roles and responsibilities is important.

Before pursuing any MBD activities, buy-in is essential from all project stakeholders. Not only do the participating engineering groups need consensus, but management, other supporting functions, and even the customer in certain circumstances. We have had explicit requests from customers to *not* use MBD on certain development contracts due to perceived certification hurdles. Obtaining buy-in requires preparation. Without a strong value proposition, an attempt to change the process will be met with significant push-back. External data is obtainable to make the case for MBD, but often internal studies will also be necessary. As stated above, there is risk inherent with process change, and it makes sense for all stakeholders involved to understand the risks and benefits involved.

> Lesson: Before pursuing any MBD activities, buy-in is essential from all project stakeholders.

New skills may be required to perform MBD-related tasks. These will typically be skills that the current engineering team will be able to acquire, but training may be needed and there could be a learning curve, which could offset project schedules in the beginning. Even though our development group attended trainings and hired consultants to help us with various aspects of MBD, there was no substitute for practical project experience.

> Lesson: Account for the learning curve associated with new MBD process-related activities in project schedules.

C. Value-Added Practices

Model based development can be applied to virtually all aspects of a program, including system modeling, use cases, software and hardware behavior, and CAD. However, it is important to address the appropriate scope for MBD as part of a program. The scope should be based on program requirements and organizational maturity and capability. For example, on some of our projects we found that model based design provided a substantial advantage to activities such as verification and validation, requirements management, and document generation. However, other activities, such as activity diagram development, provided little added benefit for the level of effort required.

> Lesson: For each program, set a reasonable scope for MBD based on program requirements and organizational capability.

Logic subsystems can be reused in other controller designs and applications. For instance, in the aerospace fluid conveyance sector, there are a few types of line replaceable units (LRUs) that are the building blocks of a system, for example the Engine Driven Pump (EDP). The EDP state can be determined by monitoring the engine operating state, pressure switch state, and system pressure reading; if the engine is running but the pressure switch is low, then the logic will have determined that the pump has failed and not delivering pressure. The subsystem can then be saved and version-controlled in a model library. When future projects require monitoring of the EDP state, the readily available subsystem can be pulled from the library as a reference block.

In addition to reducing future development time, reusing logic blocks in this manner can also reduce testing. The verification test cases, as well as the test case results, can be reused to re-verify or claim similarity in future projects. Caution must be used when determining the reusability of a subsystem, however. The controls engineer must determine that the failure modes of the LRU are the same as those the logic was designed to monitor. Additionally, if a one-time certification credit is sought for the reusable component in order to reduce certification effort in the future, there are additional steps that must be taken, such as described in AC 20-148 [18].

> Lesson: Take advantage of reusability, but understand the certification guidelines first.

Establishing a set of modeling guidelines is key to any model based development process. Not only will it allow for uniformity between product design within the organization, but it is also required to obtain DO-178C certification for software designs developed using model based development. There are several established sets of guidelines for modeling in Simulink available for public review. The MathWorks Automotive Advisory Board (MAAB) guidelines are a good starting point. These rules can then be enhanced by adding organization-specific guidelines.

For example, when signal and block names are not explicitly specified, most modeling tools will generate a unique name using a set of random characters. This can lead to code that will be generated non-deterministically, will result in name mangling, and be difficult to maintain and merge with a baseline. If any form of code generation will be used, these signal and block names should be specified by the design engineer, even if these names will never be referenced. Adding a rule to our modeling guidelines to enforce the practice of naming all modeling elements will result in consistent models, as well as a consistent code base.

> Lesson: Establish a set of agreed-upon modeling guidelines and use static model analysis to enforce them.

D. Pitfalls

While multiple tools are available commercially to facilitate model based development, they typically address different aspects of model based development with varying effectiveness. Based on the scope and complexity of a given program's MBD efforts, it may be necessary to adopt multiple tools in order to accomplish the intended goals. It should be noted that some tools are developed by different vendors, and are not necessarily designed to interact with one another efficiently. Understanding the limitations of available tools

and their interactions with one another is key to developing an efficient model based development process. Furthermore, certain projects are not conducive to MBD at all. Project schedule and product quality can be significantly impacted by poor choice of development method. It may be tempting to use a particular tool on a project simply because that tool is available. However, the inevitable project inefficiencies that will result could lead to the tool's removal from all future development projects, including projects in which the tool would provide value.

> Lesson: Do not model just to model. Make sure the tool being used for a development activity is appropriate and the interactions between tools are well understood.

In order to provide traceability between models and design requirements, Simulink has the ability to link model elements to external objects in a variety of applications via the Simulink Verification and Validation toolbox. This feature extends to IBM Rational DOORS, which we utilize for requirements management. Users can create links from Simulink to the corresponding requirements in DOORS and vice versa. Users can also automatically synchronize changes between the two programs and create customized traceability reports using Simulink Report Generator. We have experimented with utilizing Simulink Verification and Validation on a small scale. Prior to implementation, we performed a thorough review including testing the interaction between the application and the DOORS database. We found the feature enhances design traceability when maintained properly; however, significant upfront effort was required to ensure the application could be integrated into our pre-existing requirements management process without causing unintended consequences. Had we attempted to establish traceability using this method on an actual project without first coordinating with the DOORS database group, we would have faced a large schedule disruption while this work was carried out.

> Lesson: When adopting new tools and techniques on projects, a thorough understanding of their use and how exactly they fit into the development process in needed. Procedure review with all stakeholders is necessary to understand if any prerequisite work is needed, even for seemingly simple development activities.

IV. Conclusion

Perspectives from both management and engineering functions, along with the rapid adaptation of the model-based development methodology by OEMs, demonstrate the tangible benefit that can be harvested by organizations aiming to become turn-key system suppliers. The experience we have gathered through this endeavor could benefit other suppliers in reducing cost and development time by avoiding detour, enabling parallel-path development activities, and by avoiding costly pitfalls.

With well-developed tools and processes in place and organizational collaboration, model-based development can prove to be a powerful tool for expediting system design, mitigating risk early, and automating embedded system code generation.

Acknowledgments. We would like to thank Stefan Frischemeier, Rudy Rusali, Gerard Benenyan, Jake Stroud, Llonda Bins-Lohman, Martin Holland, Ankur Ganguli, and the Eaton Aerospace and Corporate leadership for their contributions, support, and feedback for this work.

References

[1] J. Lin, "Measuring return on investment of model-based design," *The MathWorks*, 2014.

[2] D. Cofer, M. Whalen, and S. Miller, "Software model checking for avionics systems," in *27th Digital Avionics Systems Conference (DASC 2008)*, 2009.

[3] M. Broy, S. Kirstan, H. Krcmar, and B. Schtz, "What is the benefit of a model-based design of embedded software systems in the car industry?" *Jrg Rech, Christian Bunse (eds.): Emerging Technologies for the Evolution and Maintenance of Software Models.*, 2011.

[4] R. Pressman, *Software Engineering, A Practitioner's Approach, 3rd Edition*. McGraw Hill, 1992.

[5] P. F. Smith, S. M. Prabhu, and J. H. Friedman, "Best practices for establishing a model-based design culture." The MathWorks, 2014.

[6] N. Törngren and O. Larses, "Characterization of model based development of embedded control systems from a mechatronic perspective; drivers, processes, technology and their maturity," in *Summer School on Model Driven Engineering for Embedded Systems*, 2004.

[7] The MathWorks, "Simulink," http://www.mathworks.com/products/simulink/, accessed: 2014-11-12.

[8] IBM Rational, "Doors," http://www.ibm.com/software/products/en/ratidoor, accessed: 2014-11-12.

[9] The MathWorks, "Simulink verification and validation toolbox," http://www.mathworks.com/products/simverification/, accessed: 2014-11-12.

[10] RTCA DO-178B, "Software considerations in airborne systems and equipment certification," 1992.

[11] RTCA DO-178C, "Software considerations in airborne systems and equipment certification," 2011.

[12] RTCA DO-331, "Model-based development and verification supplement to DO-178C and DO-278A," 2011.

[13] RTCA DO-330, "Software tool qualification considerations," 2011.

[14] The MathWorks, "Stateflow," http://www.mathworks.com/products/stateflow/, accessed: 2014-11-12.

[15] Reactive Systems, "Reactis," http://www.reactive-systems.com, accessed: 2014-11-12.

[16] SAE ARP4754A, "Guidelines for development of civil aircraft and systems," 2010.

[17] RTCA DO-333, "Formal methods supplement to DO-178C and DO-278A," 2011.

[18] FAA AC 20-148, "Reusable software components," 2004.