Applying Continuous Integration principles in safety-critical airborne software

André Barbieri Boralli[1], Ricardo Bedin França[1]

[1]EMBRAER S.A.
Rod. Presidente Dutra, km 134 – 12247-820 – São José dos Campos – SP – Brazil
{andre.boralli,ricardo.franca}@embraer.com.br

Abstract. *In this paper, we propose an approach to apply Continuous Integration principles in the development of airborne software subject to DO-178C considerations. We present the fundamentals of the DO-178C and continuous integration, then we explain the software development workflow used in our case study, which is a flight control software program. We provide possible applications of Continuous Integration in several stages of software development, according to our workflow and the needs of our clients (flight control law engineers). Finally, we present the results obtained by the practices evaluated in our case study.*

1. Introduction

1.1 Context

Given the potentially catastrophic consequences of an aircraft accident, it is natural that certain aviation embedded software programs figure among the most critical software products one can imagine. Hence, developers invest large amounts of time and resources to have sound products and processes, while following very stringent guidance, such as the DO-178C [SC-205 RTCA 2011]. On the other hand, in the middle of this rigorous environment, there are budgets and schedules that must be respected by any enterprise that hopes to be competitive. Therefore, it makes sense to look for development processes that are both sound and lean: while following the DO-178C considerations usually leads to solid software development processes and products, the same cannot be said about lean practices. Indeed, a naive use of the waterfall-like software development process as explained in the DO-178C (requirements, design, coding, integration, verification) would lead to the infamous and cumbersome "Big Requirements Up Front", which usually yield integration issues, missed deadlines and cost overruns.

Since the birth of the Agile Manifesto[1] in 2001, much attention has been devoted to the use of agile techniques in the development of all sorts of software products, including embedded, safety-critical ones. While there are many available agile techniques, with a broad scope of application, we shall focus our work in Continuous Integration, which is one of several practices that help in agile software development.

[1] agilemanifesto.org/

1.2 Objectives

In this work, we intend to present an approach of using Continuous Integration in several stages of software development, while meeting the DO-178C considerations. Specifically, we have the following goals:

- **Evaluate Continuous Integration practices and their capability of meeting, directly or indirectly, DO-178C objectives**: While we would like, ideally, to use Continuous Integration to help in full compliance with DO-178C objectives, it might be simpler to use it as a means to increase product maturity and reduce rework in the software product life cycle.
- **Assess the use of such practices in a DO-178C level A software development:** In order to keep the study in touch with actual needs and constraints of real-life projects, we chose a case study that fits in the most critical software considerations of the DO-178C.

1.3 Related Work

The use of agile methods in safety-critical software might have seemed odd at a first sight, as they could be regarded as undisciplined [Douglass and Ekas, 2012]. This prejudice has been dispelled and some interesting research has been done in this field.

A very relevant work with agile methods and airborne software is Chisholm's dissertation [Chisholm, 2007], which discussed the application of several agile practices in a DO-178B level C software process and evaluated if (and how) they could address the necessary process objectives for software products of that criticality level. Other works [VanderLeest and Buter, 2009] [Wils et al, 2006] [Chenu, 2009] follow similar approaches and present results that show the viability of many agile practices in the environment of airborne software. Agile practices can benefit other domains of safety-critical software, such as railway [Jonsson et al, 2012] and medical [Mc Hugh, 2013].

Our work has a different focus from the aforementioned ones: while they present insights over the whole software development process, using many agile techniques, we opt to focus on Continuous Integration and deepen our study to the level of implementation methods and tools. This emphasis is also found in Stolberg's work [Stolberg, 2009], although his work is not oriented to safety-critical software.

1.4 Structure of the paper

Firstly, we explain the fundamentals of the DO-178C in Section 2. Section 3 briefly discusses Continuous Integration practices, Section 4 presents the workflow of our case study and the use of Continuous Integration in our workflow is presented in Section 5.

The results obtained in our case study are discussed in Section 6 and our conclusions are presented in Section 7.

2. DO-178C and its supplements

RTCA's DO-178C is the *de facto* standard for the development of airborne software: while aircraft makers and suppliers are not forced to make use of it, certification

authorities usually acknowledge the DO-178C as a valid means to demonstrate the soundness and rigor of an airborne software product life cycle process.

While the DO-178C does not enforce a specific software life cycle process, it does mention a sequence of steps that is similar to "traditional" development cycles, such as the waterfall or the V-model. We chose the latter to briefly explain the DO-178C concepts, as shown in Figure 1.

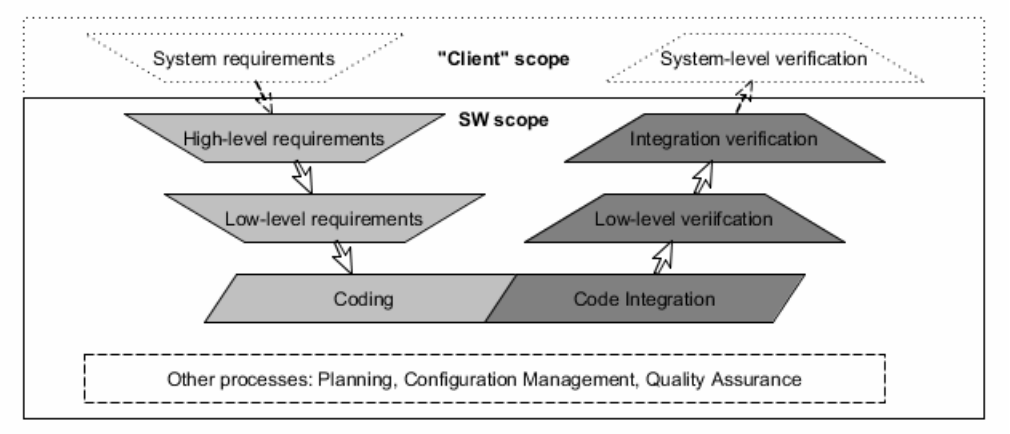

Figure 1- A V-cycle based in DO-178C

The product life cycle, according to the DO-178C, comprises software planning, development processes and integral processes - such as verification, configuration management and others. Two main keywords for the software product development are **traceability** and **compliance**: traceability shows concrete links between one development step (e.g. low-level requirements) and its adjacent ones (e.g. code) while compliance demonstrates that a development step implements correctly the previous one (e.g. low-level requirements detail correctly what was specified in the high-level ones).

Another important concept is **independence**. Independence exists whenever verification activities over a software item artifact is performed by people or team different than the ones producing the item [SC-205 RTCA 2011]. Hence, people that write the requirements are not the same that review them, and the developer of a given code module and the verifier of this same module are not the same person.

Given the different levels of criticality for airborne software (a failure in a flight control software and a bug in the passenger entertainment system do not necessarily have the same consequences on the flight itself), the DO-178C specifies five levels of criticality. In level E (the less critical), one does not need to follow any of its considerations, while in level A the most stringent guidance applies.

The DO-178C guidance includes many specific objectives for each life cycle activity; we shall discuss them in more detail in section 5. The scope of this paper comprises the development and verification activities: we judge that these activities are to get the most positive impact of Continuous Integration practices.

Besides the DO-178C guidance, there are other documents that detail specific aspects and technologies involved in airborne software development. In the scope of this paper, the most relevant is the DO-331 [SC-205 RTCA 2011(b)], which discusses the

use of models as artifacts in a software product development cycle and adapts considerations, methods and means to the model-based environment.

3. Agile practices and Continuous Integration

The twelve practices recommended by the Agile Manifesto gave birth to several software development and management methods. Some of the best-known are Crystal[2], Scrum and XP (eXtreme Programming). While Scrum focuses in management practices in software development [Gomes, 2013], both Crystal and XP emphasize technical aspects, including the concept of Continuous Integration - which is a term mainly used in XP but also present in Crystal.

Continuous Integration (or Frequent Integration) can be seen, essentially, as a set of code build and test activities that are performed frequently and as automatically as possible, so as to anticipate code integration issues. As mentioned in [Gomes, 2013], several activities - such as tests and automatic analyses - can be performed in the Continuous Integration framework.

4. Project workflow

In this section, we present the structure of the project used as case study, considering its application in the overall aircraft development program, the teams involved in its development, and some highlights of the processes followed by them.

4.1. Structure of Case Study Project

4.1.1. Project

The project used as case study for this paper is a fly-by-wire software application, developed for flight control system of an aircraft in development. Fly-by-wire, hereafter FBW, is a system that controls flight command surfaces of airplanes through electronic signals transmitted to their actuators from embedded computers that interpret inputs of pilots' inceptors, like throttles, sticks, pedals and levers [Spitzer, 2001].

Besides to receive and interpret pilots commands to send them to flight control surfaces, FBW systems are projected to implement the Flight Control Laws (hereafter CLaws), which are complex calculations performed by Flight Control Computers (hereafter FCC) that use signals from other aircraft systems to provide control surface movements that conform to pilots' commands. The scope of our case study comprises FCC software that implements aircraft CLaws - given the criticality of the FBW system in an aircraft, the FCC software belongs to the DO-178C Software Level A.

As an inseparable part of the overall aircraft, the final project clients are the operators of the airplane. However, these operators are rarely accessible for software developers, and they do not even care with this level of detail of the systems. Therefore, for software development purposes, the clients are flight command system engineers and CLaws designers, who allocate FBW system requirements to software.

[2] http://alistair.cockburn.us/Crystal+light+methods

To make such a decision, engineers and designers take into account the advantages of software implementation of parts of the system, usually the ones that could be simulated and previously tested before integration with the aircraft, what is much more expensive. The software development environment also could be prepared to validate hardware, test benches, models used by designers, unit and integration test vectors, and early integration of the FCC and FBW system.

4.1.2. Teams

CLaws designers engineers write most of the High-Level Requirements (hereafter HLR), that are actually control laws themselves to be implemented into FCC. These requirements are validated and unfolded by the development engineers.

Development engineers create HLR for software modules that are not in the scope of system designers (e.g. the operational system), unfold HLR into Low-Level Requirements (hereafter LLR), develop the source code and integrate them in the target computer.

Verification engineers are responsible for verification and validation of all other artifacts regarding compliance and traceability as mentioned in section 2, as well as perform all test activities in the product.

4.1.3. Software Development Workflow

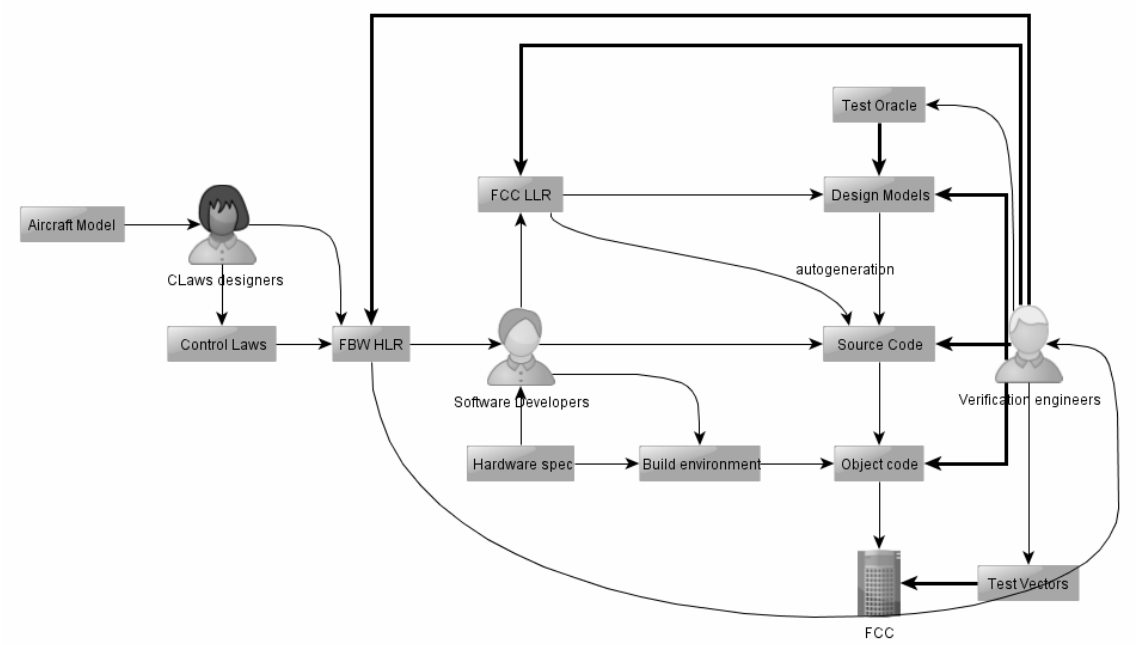

Figure 2 - FCC Software Development Workflow

Figure 2 presents a workflow that produces embedded FCC software from CLaws. The thin arrows represent artifact production flow, as well their entry points. Bold arrows, which come from verification engineers, represent validation activities executed over the artifacts to check compliance and traceability, as well as all kind of tests.

CLaws designers write HLR, describing flight control algorithms to be implemented into FCC. Developers unfold HLR into LLR, textual or models as well. In this case, models could be used to generate source code automatically. Textual LLR are used as requirements to develop code manually.

Considering hardware characteristics and constraints, developers configure the build environment to compile and link source code and build software image to fit in FCC. This configuration takes into account characteristics of processors, size and type of memory, and other hardware features to be considered.

Verification engineers execute activities that validate all steps of this workflow. They generate cases and procedures for software functional verification, and they also perform reviews and analyses on software development artifacts.

In this workflow, we need close cooperation among the teams to ensure (when changes are necessary) precise impact analysis and minimize re-verification effort.

4.2 Benchmark for evaluation criteria

Since we did not intend to develop two parallel case studies to evaluate the gains obtained with Continuous Integration, we based our "non-agile" time and cost estimates on the results of the first development iterations (when some activities and processes were not yet mature enough to be suitable to Continuous Integration). When concrete results are not available - either because we never executed the non-agile approach or because the gains are not tangible, we avoid a "worst-case benchmark" and, instead, assume a benchmark that misses one or two key concepts (e.g. periodical activities that are executed in a semi-automated way, or a process that does not use model-based development as intensively as ours).

5. Continuous Integration in a DO-178C-oriented workflow

In order to verify the applicability of Continuous Integration in our case study and similar development cycles, we follow Chisholm's [Chisholm, 2007] approach and use DO-178C (or DO-331, when applicable) tables as a reference for the process and product objectives. Though such tables "should not be used as a checklist" [SC-205 RTCA 2011], they synthesize most essential aspects that shall be taken into account when developing airborne software products.

We do not necessarily intend to use Continuous Integration to directly fulfill the DO-178C objectives. Instead, we prioritize its use to help product maturity and reduce rework when the official "run-for-score" development activities take place.

Since the Continuous Integration activities are highly focused on the program implementation, we prioritize the development processes (requirements, coding, integration) and their verification. We also consider that configuration management is in the scope of our work: while the DO-178C presents detailed guidance on configuration management practices, the only working hypothesis that we need is that there is a centralized server, accessible to all development engineers, where the development

artifacts are available for access and modification. There are many tools that may be used for this purpose, such as GNU Bazaar[3], Rational ClearCase[4], Git[5] and Subversion[6].

Another essential item, which is not necessarily linked to a DO-178C objective, is a Continuous Integration server to automatically run integration and test activities. The server usually runs tools such as AnthillPro[7], CruiseControl[8] and Jenkins[9].

5.1 Traceability between development steps

As mentioned in section 2, traceability is a key concept in the DO-178C. The following traceability evidence is required:

- Between system requirements and high-level requirements,
- Between high-level requirements and low-level requirements,
- Between low-level requirements and source code,
- Between source code and object code (for level A),
- Between verification cases and requirements.

The last item is not explicit in the DO-178C tables but we find it essential to accomplish the requirement-based test strategy as explained in the DO-178C.

The analysis of traceability between source code and object code is a rather manual task, as it requires semantic understanding of two languages to be performed. On the other hand, all other traceability evidence is produced in essentially the same way: as requirements and test cases have unique identifiers, traceability matrices and tables can be constructed to associate them - as for the source code, low-level requirement tags can be used at an arbitrary code granularity level.

While such matrices must be analyzed manually (due to the necessary semantic understanding of the evidence), their automatic and frequent generation is useful to find more obvious errors, such as the absence of code for a given low-level requirement.

5.2. High-level requirements development and verification

Traditionally, one thinks of requirements as text-based [Marques et al, 2012], although they could be expressed by models or any other formalism.

If textual high-level requirements are used, there is little automation that can be done, but some syntactic checks are possible (spell checks, parenthesis/brackets count, "shall" word count). Such checks can provide partial help to the compliance-related DO-

---

[3] http://bazaar.canonical.com/en/

[4] http://www-03.ibm.com/software/products/en/clearcase

[5] http://git-scm.com/

[6] http://subversion.apache.org/

[7] https://developer.ibm.com/urbancode/

[8] http://cruisecontrol.sourceforge.net/

[9] http://jenkins-ci.org/

178C objectives. If the high-level requirements are more formally defined, it is possible to devise stronger automatic syntactic and semantic verifiers.

In our case study, we opted to use text-based high-level requirements and prioritized Continuous Integration in other project steps, where the need for agility seemed most urgent and with more tangible gains.

5.3 Low-level requirements development and verification

While low-level requirements can also be associated with text, the advent of tools that generate code from models boosted the viability of model-based design. Besides the perceived ease in graphical modeling (instead of coding), model simulation tools are extremely useful to perform software verification - either as a official means or as a dry-run environment to increase product maturity. Another advantage of model-based design, when using suitable modeling languages, is the improved requirement consistency attained thanks to unambiguous models. Developers can devise consistency checks - for example, semantic checks in connections between design blocks.

One can argue that there are software modules that cannot or should not be developed using model-based design. For our case study, this is indeed true, as the embedded operating system that runs our flight control software requires much low-level programming, but we consider that the control application itself is much more voluminous and subject to change than the operating system - therefore, we can safely assume that the application is more relevant than the operational system for our study.

When using models as low-level requirements, we consider that the test oracle should also be a model. In this way, not only each model component would be easily testable, but more elaborated simulations (often called Software-In-The-Loop[10]) would become possible early in the software development cycle, helping in the validation of the implemented control algorithms and software interfaces. There is room for even more advanced verification, such as overflow detection [Honda and Vieira Dias, 2013] which is usually done more expensively at later development stages.

In any design approach (text, models or other), it is still possible to automate at least part of the design standard verification activities, although this is easier in design models than in text. Aspects such as naming conventions, diagram size, number of entities per model and many others can be verified automatically and frequently.

We summarize our proposed Continuous Integration means to tackle the DO-178C objectives related to low-level requirements in Table 1.

Table 1 - Continuous Integration activities for model-based design

Objective	Compliance Method	Attained objective coverage
Compliance with HLR	Frequent requirement-based verification (tests and/or proofs), software-in-the-loop simulations	Partial (product maturity with no process impact), can be full at a larger initial cost (integrate tests and processes with CI software)
Accuracy and consistency	Automatic semantic checks (e.g. interfaces, numeric overflow)	Partial (some degree of review is usually necessary)

[10] http://www.acm-sigsim-mskr.org/MSAreas/InTheLoop/softwareInTheLoop.htm

Conformance to standards	Automatic checks	Partial (not all checks can be automated)

5.4 Software Architecture

In the scope of the DO-178C, software architecture is treated as a design artifact. Nevertheless, its construction is heavily influenced by both high-level requirements and feedback from integration and verification activities - hence, constructing it in a pure waterfall-like fashion is a nightmarish activity. Thus, we recommend a more automated architecture development approach - at least for software scheduling, which can be very time-consuming - such as in Masini's work [Masini et al, 2014]: in order to keep the architecture as optimized as possible, it is important to perform "downstream" activities, such as timing and memory profiling, as soon as possible. Complex hardware-related aspects would still be dealt with manually but the automated scheduling eases the overall task of generating and maintaining a suitable software architecture.

Since the architecture document may also be subject to standards, it is once again possible to automate some syntactic and semantic checks, although these depend heavily on the architecture representation chosen by each specific project. Table 2 summarizes our recommendations related to software architecture and Continuous Integration.

Table 2 - Continuous Integration activities for architecture development

Objective	Compliance Method	Attained objective coverage
Compliance with HLR	Automatic scheduling generation	Partial (reviews are usually necessary)
Accuracy and consistency	Automatic scheduling generation	Partial (reviews are usually necessary)
Conformance to standards	Automatic checks	Partial (reviews are usually necessary)

5.5 Coding and Integration

In our approach, most of the source code should come from an automatic code generator, thus not necessarily being subject to standards or to verifications that may be done at design level. However, the hand-coded software modules can be verified, at least partially, with respect to coding standards. In addition, there are many tools that perform static verification over the source code to prove specific properties (as mentioned in section 5.3) or detect run-time errors, such as Astrée[11] and Polyspace[12].

As in typical Continuous Integration applications, we also strive to ensure the integrity of a software build. First of all, frequent build attempts can quickly detect a broken build. Many activities can be performed to verify build integrity:

- Automatic analysis of the program map files helps evaluating memory usage.
- Timing and stack usage analysis can be performed either by static analysis of the executable object code or by tests on the target computer. If the latter alternative is chosen, the usage of test equipment is greatly optimized if the Continuous Integration server controls the jobs that are sent to the available target platforms.

[11] http://www.absint.com/astree/index.htm

[12] http://www.mathworks.com/products/polyspace/

- High-level and low-level tests can also be scheduled and run by the Continuous Integration server on the target platforms, once again optimizing their use.
- It is even possible to perform automated hardware-in-the-loop simulations if simulation models of the aircraft and test procedures are available.

  Not every test can be executed continuously: as the product evolves, more tests are needed and some have to be changed. Nevertheless, the examples above amount to a large part of all necessary integration verification activities.

6. Case Study Results

The adoption of Continuous Integration practices described in the section 5 allows us to execute our workflow without significant break in the FCC software development activities while the product evolves. Indeed, since the aircraft and all its systems are under development almost simultaneously, it is expected that CLaws (and their corresponding software HLR) evolve, too.

Considering the workflow described in the section 4.1.3, the minimum sequence of activities for any CLaws evolution step includes:

- Updates in requirements and models,
- Code auto-generation, coding and build,
- Model and code reviews and analyses,
- Requirement-based tests,
- Software- and Hardware-in-the-loop simulations with the client.

There are thousands of CLaws requirements to be allocated to FCC software, thus all this "minimum cycle" might take up to one hundred men-months to finish if the product is developed from scratch in a single step.

In order to increase maturity of the FCC software and anticipate integration problems within CLaws evolution cycle, we allocated the following tasks to a Continuous Integration server:

- Code auto-generation, and build,
- Execution of automatic model analysis scripts,
- Model testing with respect to the oracle model,
- Integration of software components in the client environment and software-in-the-loop simulation runs,

All activities are triggered by update of the software items into version control, in case of any failed task (e.g. a broken build), the software developer involved in the update is noticed to start corrective actions. Based on the time we took to keep build integrity and detect and fix errors before setting up our continuous build, analysis, simulation and test environment, we estimate that such automation alone may reduce the overall product lead time by 50%. These time savings are more pronounced in periods where the software product is being subject to frequent and numerous requirement changes, as these can be quickly verified and validated.

Besides time saving, Continuous Integration practices allowed us to balance workload throughout the software development cycle. Even if, for software approval purposes, the project is in a preliminary stage (e.g. HLR development and verification), it is possible for development and verification engineers to follow "downstream" with the build procedures and validate the planned development processes and improve the Continuous Integration activities.

Our Continuous Integration system also allowed the development of several verification and simulation platforms, many of them running at developer workstations, whose operation cost is relatively low. Since all software image and simulation environments are built immediately whenever FCC software items are updated in version control system, we can guarantee that all platforms are running consistent versions of our embedded software. We are able to detect problems that otherwise would be found out by using much more expensive means of test, like Iron Birds (ground-based test benches with flight control structures, actuators and controllers), or even in flight test. In particular, we have the following problem anticipation patterns:

- Desktop tests anticipate modeling/coding errors before on-target testing,
- Desktop simulations anticipate some on-target integration issues,
- Hardware-in-the-loop simulation can anticipate problems that would be found in Iron Birds and flight test campaigns.

While it is hard to estimate concrete cost savings from the above patterns, one can easily figure out that an embedded target computer is much more expensive than a desktop one and that any test means is less expensive than a prototype aircraft.

7. Conclusions and future work

Our main conclusion from this work is that widespread use of Continuous Integration techniques and tools through the development cycle of a DO-178C level A software is not only possible, but very beneficial. We obtained concrete gains in time (we can use less human effort than in non-agile approaches) and in verification costs, as we were able to detect and fix errors in earlier project phases, using less costly means.

Although Continuous Integration is not dependent on model-based techniques, we did notice that the adoption of frequent test and simulation activities was rendered much easier by the use of models in software design and test cases development.

We understand that there is still room for more Continuous Integration practices on our software development process. We intend to focus future work on automating more static review and analysis activities, which are unexciting and time-consuming. Another future work is the evaluation of static analyzers (such as Frama-C[13]) as early verification means, as the construction of test cases can sometimes be slower than the development of a model or a code functionality. Such analyzers have already been employed successfully in the aircraft industry [Souyris et al, 2009].

[13] http://frama-c.com/

References

Special Committee 205 (SC-205) of RTCA. (2011) “DO-178C, Software Considerations in Airborne Systems and Equipment Certification”.

Special Committee 205 (SC-205) of RTCA (2011) “DO-331, Model-Based Development and Verification Supplement to DO-178C and DO-278A”.

Douglass, Bruce P. and Ekas, Leslie. (2012) “Adopting agile methods for safety-critical systems development”, IBM Software Thought Leadership white paper.

Chisholm, Ronald A. (2007), "Agile Software Development Methods and DO-178B Certification", Master’s Thesis, Division of Graduate Studies of the Royal Military College of Canada.

Chenu, Emmanuel. (2009) "Agility and Lean for Avionics". Lean, Agile Approach to High-Integrity Software Conference, Paris, France.

Wils, Andrew; Van Baelen, Stefan; Holvoet, Tom and de Vlaminck, Karel. (2006) "Agility in the Avionics Software World". XP 2006, Oulu, Finland.

VanderLeest, Steven H. and Buter, Andrew. (2009) "Escape the waterfall: Agile for aerospace". 28th Digital Avionics Systems Conference, Orlando, USA.

Mc Hugh, Martin; Cawley, Oisin; McCaffery, Fergal, Richardson, Ita and Wang, Xiaofeng. (2013)"An Agile V-Model for Medical Device Software Development to Overcome the Challenges with Plan Driven SDLCs". The Software Engineering in Healthcare (SEHC) Workshop at the 35th ICSE, San Francisco, USA.

Jonsson, Henrik; Larsson, Stig and Punnekkat, Sasikumar (2012). "Agile Practices in Regulated Railway Software Development". 23rd IEEE International Symposium on Software Reliability Engineering, Dallas, USA.

Gomes, André F. (2013). "Agile: Desenvolvimento de software com entregas frequentes e foco no valor de negócio". Casa do Código.

Marques, Johnny C.; Yelisetty, Sarasuaty M. H.; Vieira Dias, Luiz A. and da Cunha, Adilson M. (2012). "Using Model-Based Development as Software Low-Level Requirements to Achieve Airborne Software Certification". ITNG, Las Vegas, USA.

Souyris, Jean; Wiels, Virginie; Delmas, David and Delseny, Hervé (2009). "Formal verification of avionics software products". FM 2009: Formal Methods, Lecture Notes in Computer Science Volume 5850.

Masini, Henrique F.; França, Ricardo B.; Bezerra, Juliana M. and Hirata, Celso M. (2014) "An approach to generate optimized cyclic scheduling from AADL specification ". 33rd Digital Avionics Systems Conference (to appear), Colorado Springs, USA.

Honda, Renato and Vieira Dias, Luiz A. (2013). "RangeAnalyzer: An Automatic Tool for Arithmetic Overflow Detection in Model-based Development". ITNG, Las Vegas, USA.

Spitzer, Cary R. (editor) (2001). "The Avionics Handbook". CRC Press