Industrial use of a safe and efficient formal method based software engineering process in avionics.

Abderrahmane Brahmi*, Marie-Jo Carolus*, David Delmas*, Mohamed Habib Essoussi*, Pascal Lacabanne*, Victoria Moya Lamiel*, Famantanantsoa Randimbivololona** and Jean Souyris*.

*Airbus Operation S.A.S

**Cepresy

Abstract: Formal methods have reached industrial efficiency in avionics thanks to the development and deployment of an engineering process for software design and verification processes. It encompasses languages, compilers and formal verification tools in a highly automated workshop, together with adapted methods of use. This engineering process involves functional and non-functional formal verification techniques in a complementary way. It is being applied to the new avionics software products developed at Airbus. Currently, this means that tens of developers have been using the workshop daily since its initial deployment, three years ago. After presenting this engineering process, the main purpose of this paper is to report on its industrial use.

Keywords: Avionics software development, engineering process, formal methods, automatic workshop, industrial maturity.

1. Introduction

After a period of research, followed by a phase of industrialisation, Formal Methods now occupy an important place in the development process of avionics software products at Airbus. This component-based process is made of implementation and verification activities. The first ones are the specification, architecture, design and coding of the components. Verification activities apply to the artefacts resulting from the implementation activities.

The activities supported by formal methods relate to design, compilation, functional verification of source code against design and verification of non-functional properties of source and binary codes Formal languages are used for design and functional verification activities. The formal techniques employed are program proof and static analysis by Abstract Interpretation.

The engineering process at stake in this paper has been the object of [8]. By “engineering process” we mean a set of tool chains orchestrated by a build system that automates and coordinates activities of the development process, plus guidelines, methods and trainings.

The main objectives attached to this engineering process in [8] were the quality of the design, the soundness of the verifications, the suppression of time-consuming verification activities (like readings/reviews) and the substitution of costly activities by cheaper ones.

Therefore, the aim of this paper is to report about the actual use of this innovative engineering process and to make a first assessment of it.

The rest of the paper is organised as such: section 2 presents the development process, section 3 exposes the components of the engineering process in relation with the process shown in section 2, section 4 is the industrial feedback and gives some room for improvement, and section 5 concludes.

2. Avionics development process (NWOW)

Airbus is currently investing significant effort into an internal initiative known as New Ways of Working (NWOW), which aims at improving the industrial efficiency of the avionics software development processes, while maintaining the highest standards for safety. The way to achieve this ambitious target is to move from hand-crafted legacy processes to an automated process.

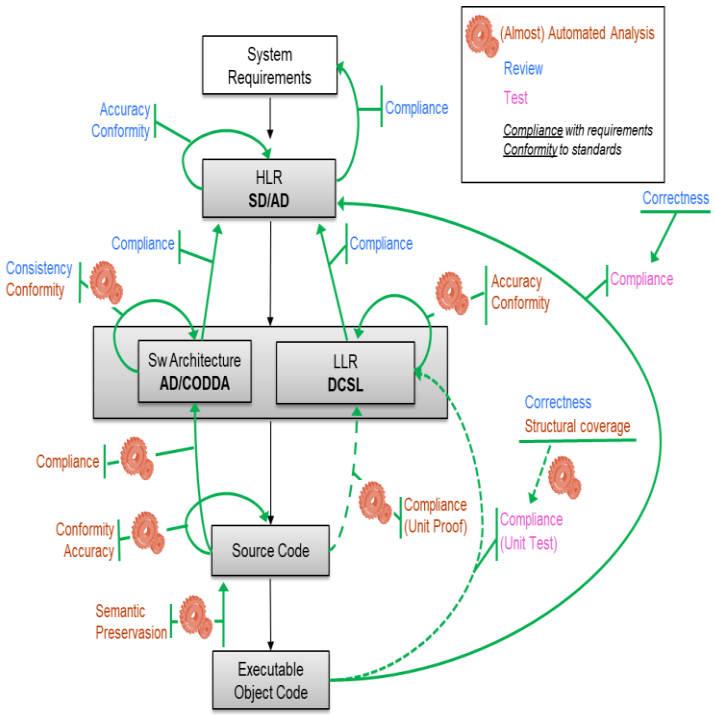


Figure 1: NWOW development process

All artefacts produced by development processes, i.e., in DO-178C [1] terminology, High-Level Requirements (HLR), Architecture, Low-Level Requirements (LLR), source and Executable Object Code are verified with different objectives. All of them are reviewed for compliance with artefacts from which they are derived on one hand, and for accuracy, consistency, hardware compatibility and conformity to standards on the other hand. In addition, executable code is verified against LLR and HLR by means of unit and integration testing. Test cases and procedures are then also subject to pair reviews.

Figure 1 gives an overview of the NWOW process.

HLR: HLRs are out of the scope of this paper since they are not defined formally.

Software Architecture and LLR: The design phase, presented in the central box, is deeply revisited. The NWOW approach to automation is language-based. As previously, the design is the specification of the code. Now, the formal contracts are the major inputs of the unit verification and static analysis. Domain specific languages with well-defined syntax and semantics have been created, to enable the formalization of all design artefacts. Dedicated compilers have been developed, so as to allow automatic, safe computations on explicit, unambiguous design data:

- The **Software Architecture** is described in a dedicated domain-specific language (CODDA). The software is decomposed hierarchically into a set of logical modules featuring both exported interfaces and hidden implementations. Relations between modules are expressed in terms of decomposition and dependency. All programming objects are then introduced and described in this language, and mapped to modules: types, constants, variables, procedures, code and data sections, non-memory mapped hardware registers, special processor instructions (e.g. memory barriers and cache management). This is exactly the set of objects to be constrained by LLR. Therefore, the level of description is not only sufficient to enable automatic extraction of source code skeletons to be filled in at coding time or merged with existing code, but also any data relevant to unit verification. For instance, procedure parameters are annotated with their direction (in, out, or both), their passing mode (by value, address or reference), and their class (e.g. arrays passed by reference are annotated with their actual length). Specific instructions expecting an argument (e.g. PowerPC mbar) are described with a function-like interface.

- The **LLR**s are expressed in a first-order logic based language (DCSL). They describe the observable behaviors of procedures, to be implemented in C or assembly, knowing the annotated interfaces of callees, but not their behaviors. This language is reminiscent of standard Behavioral Interface Description Languages (BISL) [5]. Functional requirements on procedure behaviors are formalized as first-order contracts, expressed in terms of pre and post-conditions over terms constructed on objects defined as part of the software architecture. In addition, non-functional requirements are formalized to automate verification. Indeed the LLR language supports formal descriptions for sequences of calls and volatile accesses, non-standard Application Binary Interfaces (ABI), decomposition of machine words into named fields, (e.g. to model accesses to register fields), and execution of special processor instructions.

Obviously, a first advantage to such design formalization is to replace by automated analysis a significant amount of reviews of the design data, for accuracy, consistency, and conformity to standards.

Source and Executable Object code: Additional process optimizations are obtained in the case of C source code.

- Reviews for conformance with the software architecture are automated using a mixture of syntax-based and semantics-based static analysis techniques, such as data flow analysis.
- Reviews for conformity to standards are automated by syntax-based static analysis tools.
- Reviews for accuracy and consistency are automated by abstract interpretation based static analyzers (CheckRTE, Anafloat).
- In addition, Unit Proof enables to verify that the source code complies and is robust with the LLR. Because of the preservation of semantics obtained by the use of the compiler CompCert the correctness of the translation of source to object code is verified. Therefore the formal analysis performed at the source code level against the LLR by Unit Proof is used to infer correctness of the Executable Object Code against the LLR, See [DO-333] item FM.6.7.f.(2).[2].

Unit Verification: The key interest of NWOW is to propose an approach for unit verification with two alternatives: either unit proof or unit test. Unit proof is available for C source code, and extremely cost-efficient provided some minimal well-typedness and complexity constraints are satisfied: in this case the proof is automatic (Unit Proof tool chain). Unit test is available for assembly or less standard C code.

Key features for industrial efficiency are:

- all procedures of a given module need not be verified with the same technique;
- no change in design data (architecture or LLR) is needed to switch from one technique to another;

In addition, most unit verification data are computed automatically from formalized designs. The only human engineered inputs are:

- For unit proof: loop invariants and proof tactics for verification conditions that are not proved automatically.
- For unit test: scenarios providing values for input variables. The set of input variables is precomputed for each behavior from formal LLR. Also, test oracles are directly extracted for LLR to be evaluated at run-time. This obviously decreases the necessary effort both in test procedure development and reviews.

Process management tool: This process is made efficient by the tight integration of a number of automated techniques. This integration is orchestrated by a process management tool (Optimases).

In the design process, developers formalize the Software Architecture. The output is a set of CoDDA files describing the Software Architecture, and a set of DCSL files describing LLRs.

A dictionary describing all design objects, as well as relationships between them is the primary input for Optimases, the process automation tool which enables it to construct the whole project dependency tree.

As part of the coding process, Optimases coordinates the activation of CoDDA and DCSL compilers to generate source code templates and headers. Templates are filled in by programmers to produce C or assembly implementations, and compiled via a build process.

Optimases triggers verification processes for the conformity to design and coding standards, unit verification (UnitProof or UnitTest toolchains), absence of Run-Time Error and floating point accuracy errors (CheckRTE and AnaFloat toolchains), correctness of data flow with respect to specifications (CheckFlow).

3. The New Way Of Working Workhop

CoDDA (Compilable Design Description Assistant) is a framework implementing the method of software static design by abstract machines (adaptation of the Hood Method). It provides a specific language allowing designers to formalize the descriptions of so-called machines featuring exported interface and hidden implementation with their constants, types, resources, services. Moreover, CoDDA provides some micro-services like a checker of design (correctness of the design), a generator of skeleton of code, of documentation (html or pdf), of traceability information and information for unit verification (proof or test).

While CoDDA targets architectural designs, down to the prototypes of individual C functions or assembly routines, we defined another language to describe the expected functional behaviours of every function.

Design-Contract-Specification- Language, DCSL, is the notation that has been coined to formally support the detailed design of embedded-software units. It is essentially a code-level Behavioral Interface Specification Language (BISL) [5], where the functional behaviors are expressed in terms of precondition/postcondition in the language of a many-sorted first-order logic. It is a well-established approach shared by all code-level BISL, such as ACSL [4], JML [6], Code Contracts [7] or SPARK Ada [3].

However, DCSL has native extensions, additions and restricted language features, so as to provide an orthogonal support for the detailed design of diverse embedded software products:

- the programming language may be C or assembly,
- the software assurance levels (DAL) range from life-critical (DO178 level A) down to weakly-safety-related (DO178 level C),
- software types range from low-level software running on bare-hardware to application software running on top of a multi-application multi-threaded embedded-OS platform,
- the unit verification may use formal proof, test, or a mixed proof-and-test,
- the development may use the classical-V, specification-driven process, or may adopt the emerging test-driven process,
- it might be a new development or the rehosting of a mature (*a.k.a.* legacy) software product.

A DCSL specification program consists of two successive parts:

1. The design of the interfaces. The specification of the interfaces (types, variables, functions) uses C99 as its core language. In addition, native DCSL constructs are provided and may be used, as required, for the specification of specific hardware resources (e.g. non-memory-mapped registers), for the specification of specific attributes of hardware resources (read-to-clear, address constraints, …), for the specification of design-level attributes (mode and direction of a formal parameter, noreturn function, range of values, …), for the specification of complex sorts (list, string, …), for the specification of user-defined predicates.

2. The design of the behaviors. The specification of the dynamic behaviors uses native DCSL constructs. It consists of two parts, none of which is mandatory.

   a. Contracts. Contracts are made-up of precondition/postcondition properties. The properties are logical formulas. The formulas are many-sorted first-order logic whose sorts and base terms are the types and variables of the program. But some properties of interest, data and control flows, hardware resources handlings, are expressed as formulas in a native DCSL construct for finite temporal sequences, whose events are the observable occurrences of program actions: calling a function, calling a function with specified values for parameters, reading a (volatile) variable, writing a (volatile) variable with a specified value. It is worth noting that formulas must be run-time evaluable. The quantifiers are restricted to finite integer intervals for the first-order formulas. And the temporal operators are restricted to finite event-counts (no “eventually”, or “always” operators) for the finite temporal sequences formulas.

   b. Tests. Tests are made-up of test cases. Test cases are formally models (valuations of the free variables) of the formulas of the contracts. The set of the test cases is accepted as a “sufficient” characterisation of the program unit (DAL-sensitive consensus approved by all actors). In that sense, when both contracts and tests are present, they are specifications of the program unit.

The DCSL compiler.

The language is fully supported by a compiler, dcslc. In the context of the nWoW, the very first input when designing with DCSL is a CoDDA-pre-generated DCSL program which contains the design of the interfaces. Additional higher-level specifications may be used by the designer to fully develop the formulas of the contracts.

The front-end of the compiler parses and analyses the DCSL program. It detects any compile-time semantic errors and warns about some possible run-time semantic errors. If no semantic error is detected, the back-end generators can be activated. No generation action is run on an incorrect DCSL program.

The back-end may generate:

For proof.

For the downstream unit proof process, a set of ACSL formal annotations for its inputs as well as additional information about the Service Under Verification (SUV) as for example volatile or function pointer instrumentation or additional NUPW options to be used during the importation and proof stages (see section Unit Proof below)

For test.

For the downstream unit test process, a set of C programs and declarations for its input datasets, predicate evaluation in C language, callee stubs or Simugene platform options (see section Unit Test below).

For static analysis.

For the downstream static analysis processes:

- Flow annotations for the verification of the control flow by Fan-C.
- Value range annotations for the verification of the global invariants by ASTRÉE, to validate used DCSL preconditions during Unit Verifications.

Up to the advent of the DCSL-supported nWoW, all these verification input artefacts were totally crafted by the designers. These activities were time-consuming and error-prone and they induced some supplementary checks, such as re-readings. They are now produced automatically by the compiler from a semantically correct DCSL program. The designers are fully focused on the detailed specifications.

They no longer have to intervene – except for the rare cases of interactive proof termination – in downstream preparation activities, or execution of verification processes that are fully supported by automatic tools.

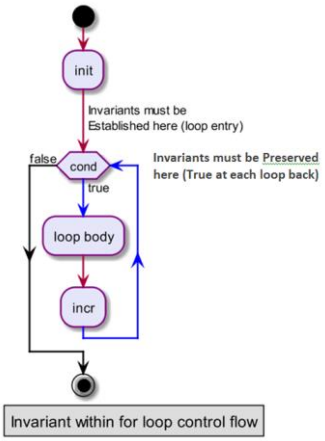
Unit proof as part of the NWOW stands for the restriction of program proof to the scope of an individual C function, along with a dedicated environment featuring stubs for callees (because of the unit verification approach). This includes different kinds of instrumentation for volatile accesses, function pointer calls and special prototyping for inline functions.

The implementation of a given C function is verified against a formal ACSL contract, computed from its formal LLR by the DCSL compiler. The proof is conducted most of time automatically within a deductive system based on Hoare Logic and Dijkstra's Weakest Precondition Calculus. The LLR compiler (DCSLC) generates unit proof contracts as well as a correct instrumentation for function calls, inline functions (other than the service under verification), pointer function calls as well as volatile read and write accesses. It generates contracts to verify the completeness of the design too.

A Frama-C plugin is used to compute the possible unrolling of loops when the number of iterations is constant and known at compilation time and when this number is reasonable. Otherwise, if the unrolling is not possible, Proof engineers must provide loop invariants. Loop Invariants are conditions written in ACSL holding necessary at the loop entry and before and immediately after each iteration of the loop. Loop invariants are crucial to prove correctness of any property related to the loop. They must be inductive and strong enough to prove post-conditions, their inference is not computable.

The generated contract in ACSL of the SUV, with its Invariants or loop unrolling directives, stubs and volatile instrumentation are submitted to the Frama-C Import and Volatile plugins, generating a unique file containing the preprocessed code with its contract and all the instrumentation and stubs necessary to launch the Frama-C WP.

There are range, functional and coupling invariants, a loop variant and loop assigns to be defined by the user. The main technique is the deductions of loop invariants from the function post-conditions when function or statement contract post-conditions have out operands depending directly from variables of the loop. The following figure shows the principle of establishment and preservation on a "for" loop:



Throughout a proof campaign, bundles of proof obligations are generated by Frama-C/WP, and submitted to the Alt-Ergo SMT-solver for discharging the Proof Obligations (PO) that QeD could not prove.

In the overwhelming majority of cases, proof is conducted fully automatically.

Nonetheless, proof engineers may terminate interactively some proof obligations by means of the Interactive Proof Editor on debug mode. In interactive mode, complex goals generated by WP may then be decomposed into smaller pieces by applying several tactics available through the user interface. These tactics can be saved in json files to be able to use them later automatically in the following proof campaigns.

Most of user preconditions are then verified with ASTRéE.

For a large overview of the Unit Proof Process, see Figure 2.

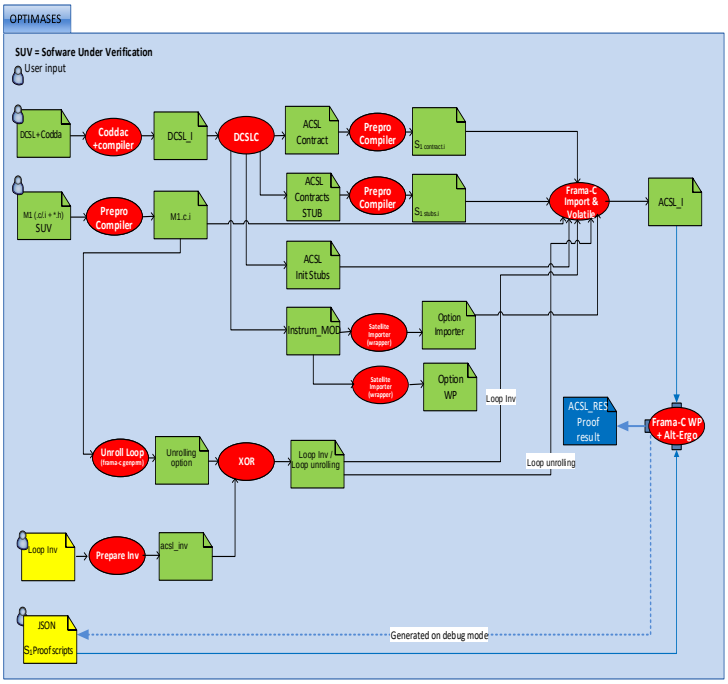


Figure 2: Unit Proof automatic process

Unit test as part of the NWOW allows to manage assembly or C code functions. It provides a specific language based on DCSL language allowing to tester to focus solely on the design of the tests: design data sets which are models of the logical formulas at the pre-state of the unit-under test.

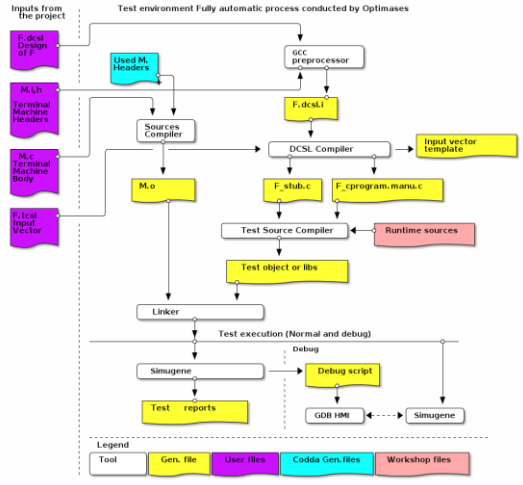


Figure 3: Unit Test automatic process

Figure 3 introduces the unit test process supported by Optimases. DCSL compiler analyses the design contract and generate a template of test vector which identifies the relevant variable to build the models. Then, Testers fill in test templates computed by the DCSL compiler with input vectors. The rest of the process is automatic: evaluation of properties (pre-state and post-state), instrumentation of called functions

In order to ease the management of lower applications (use of hardware registers, specific instructions …), and provide efficient debug, unit tests are run on the SIMUGENE representative virtual platform and associated with GDB.

Anafloat Toolchain

The AnaFloat toolchain using FLUCTUAT is being introduced in the verification process to evaluate the numerical accuracy of floating-point libraries. FLUCTUAT is an abstract interpretation based static analyser for characterizing the propagation of rounding errors in floating-point computations.

The aim of the toolchain is to prove round-off errors are small enough to rely on a real-valued semantics when proving C functions performing floating-point computations. It also proves the absence of unstable tests (1).

Furthermore, accuracy budget on some floating point computations can be validated as well as the “method precision” on well-known mathematical functions (e.g. sqrt).

The validation strategy is a semi-integrated one for the computation of global round-off errors on large calculus and unitary on method error validations or for the verification of more simple computations.

CheckRTE toolchain: the aim of the tool is to validate within ASTRéE most of the preconditions used by the unit proof (validity and separation of external pointers, absence of overflows in arithmetic’s or conversion) as well as user preconditions during the Unit Test or Unit Proof verification about variable ranges or other similar implicit assumptions. Those hypothesis, which can come from the user DCSL contract or implicitly known as been necessary to other verification steps, are translated in Astrée Annotation Language (AAL) to be used or verified, depending on their kind, by the tool.

ASTRéE is an abstract interpretation based static analyser used industrially at Airbus, prior to certification, to prove the absence of run-time errors on safety-critical synchronous control/command programs written in C. It has also be been shown to be mature for use on a large set of asynchronous, multithread avionics applications.

ASTRéE is implemented as an integrated verification on a whole program from a binary point of view.

CompCert [9] is an optimizing compiler for the C programming language. It is compliant with ISO C99 language, with few restrictions and extensions. Its backend can generate code for ARM, PowerPC, RISC-V (32 and 64-bits), and x86 (32 and 64-bits). The great originality of

CompCert is that it is formally developed and verified, using machine-assisted mathematical proofs, to be exempt from miscompilation issues. This means that the code it produces is proved to behave exactly as specified by the semantics of the source C program, thus contributing to meeting the highest levels of software assurance. In the scope of the NWOW, using the CompCert C compiler is a natural complement to applying formal verification techniques (static analysis, program proof) at C code level: the correctness proof guarantees that all functional and safety properties verified on the source code automatically hold for the generated executable as well.

Optimases is both a process management tool and a build system. Optimases allows to configure complex processes easily. A process is fully defined in XML collections. The Optimases configuration is based on

1) file types: files are all typed using pattern definition;

2) tool definition: this permits the link between inputs and outputs, based on file types;

3) the use of variables: they are mainly required for genericity; enabling the management of several level of configuration;

4) templates providing the way to implement complex processes, i.e. with several steps between user's sources and result; the user has just to apply this template with their inputs regardless the intermediate state;

5) variants: derivate a main process in another context (debug, coverage).

Tools qualification

Outputs from generation tools are verified by tool or reviewed, and all verification tools are qualified DO-178C TQL5 [1].

Nevertheless, the certification credit taken from the use of CompCert makes it necessary to qualify it at higher level than TQL5. Indeed, CompCert ensures the preservation of semantics from source code to object code, then CompCert automates a verification process, see DO-333 item FM.A-7 objective FM 9 [2]. To give more confidence in CompCert than with a qualification at TQL-5 (verification tool), it has been decided with the Certification Authorities to perform more development and verification activities on CompCert to tend towards a TQL-3.

All Toolchain qualifications are performed including the process management tool Optimases involved in the toolchain.

In addition, whenever a software development process requires the use of a qualified tool with a given set of options on a given type of machine, the process manager (Optimases) checks if the tool is already qualified in this context. If it is not the case, it launches the qualification tests automatically.

4. Industrial deployment and feedback

The balance presented now is an *intermediate* "lesson learnt" after two years of exploitation of the NWOW.

The actual deployment of the NWOW development process could start only after the following was achieved:

- A sufficient level of maturity of the workshop, i.e. ensuring the capability of the languages and tools to satisfy all the foreseen user's needs in the targeted contexts of use;
- The availability of the necessary guidelines, methodological documents and training supports;
- The set-up of a user support and maintenance organization and infrastructure involving highly skilled people and a bug tracking system.

So far, about 60 people have been putting into practice the NWOW engineering process since its entry in service two years ago. 179 abstract machines have been designed in CoDDA. 3544 DCSL contracts have been written, leading to 3315 C functions and 230 assembly routines. 98.5 % of the C functions are Unit-proved, the other ones being Unit-tested. Obviously all assembly routines are Unit-tested since Unit Proof only applies to C functions. 336 C functions necessitated the writing of loop invariants.

The reason why 1.5% of the C functions could not be proved are:

- The use of C constructs that are not supported by Frama-C/WP, i.e. byte-wise memory copy/compare and linked-list handling;
- The verification of the temporal logic aspects of DCSL function contracts.

Among the 3315 C functions, 75 (2.3%) functions required the interactive termination of some of their proofs.

On the positive side:

Adequacy to the regulatory framework: the official review of the plans (mainly the Software Development Plan and Software Verification Plan) has been passed successfully in the frame of a DAL-A development process; this means that NWOW activities achieve the objectives assigned to them. This also implies the sufficient appropriateness of methodological documents.

The adequacy to the applicative context needs has been achieved either initially or during operation thanks to the close feedback loop between the developers and the support team.

Examples of "on the fly" improvements are:

- The temporal logic extension of DCSL and associated TCSL test generation and execution
- The Component Based Developed approach (CBD) made it necessary to implement build time variability in

preliminary design (extension of the CoDDA language) and detailed design (extension of the DCSL language); downstream NWOW activities, e.g. Unit Proof, being un-impacted since their inputs are generated from CoDDA and DCSL files.

- The handling of complex structures:
  - Character strings (in DCSL, Unit Proof and Unit Test)
  - Linked lists (in DCSL and Unit Test)
- "Semantic stubs" (Unit Proof), i.e. stubs with functional contract.

The skills for performing and completing the activities

First of all, let us emphasize the fact that formal methods are a lot less "exotic" for software engineers than twenty years ago when Airbus started working on this topic in research. Indeed, most students in computer science follow courses, or at least lectures, on these techniques during their studies.

The required knowledge for starting any activity of the NWOW have been transmitted to the developers via a 12-day training. The audience of these trainings was: Airbus employees, including Airbus India, and sub-contractors of both entities. People are trained on the whole process during 2 days; CODDA: 2 days; DCSL: 3 days; Unit Proof: 3 days; Unit Test: 2 days.

In operation, these skills are improved via the support requests that the users address to the NWOW specialist team. Example: an aspect of Unit Proof is the writing of invariants for loops that cannot be unrolled automatically. Although a big part of the training on Unit Proof teaches how to find loop invariants, it is a quite difficult art. When users have difficulties to complete the Unit Proof, e.g. a C function requiring loop invariants, they are encouraged to create support requests, so they get their problem solved and they improve their skills on the topic.

Quality of the development artefacts and data. The design and especially the detailed design in DCSL are significantly more rigorous than non-formal legacy ones. This makes them closer to the spirit of DO-178C. This benefit is maximal when the sequence design, design review, coding, coding review and verification is actually performed in this order, as required.

Respect of the development schedules: the tight project schedules have been respected thanks to the high level of automation of the NWOW workshop and to the close feedback loop between the developers and the people in charge of the support, maintenance and adaptation of the workshop.

Room for improvement:

Quality of the development artefacts and data.

Excessive splitting. When the user struggles to write the LLR in DCSL or to terminate the proofs of some C function, he/she might decide to split it in two, each resulting one being simpler to design and prove. Most of the time this is beneficial to the design but sometimes it makes it locally too complex and leads to a degradation of the pertinence of the Unit verification. On a sample of 154 functions, 19 functions (12%) have been analysed as artificially split.

Code before contract. Although the DCSL contracts must be written and reviewed before coding, it happens that some developers do the other way round. The consequence is that the Low Level Requirements specification activity loses part of its interest and the verification of the C function against its LLRs is a lot less pertinent.

The above mentioned "bad practices" make it necessary to reinforce the checks of the actual application of the methods and process.

Adequacy to the applicative context needs.

DCSL:

- Enable the writer of contracts to define his/her own operators as logic functions
- Give the language new constructs for abstracting hardware resources.

Unit Proof:

- Give the user the capability to write invariants in DCSL instead of ACSL
- Combine various external provers (newer Alt-Ergo, Z3 and CVC4) for increasing the already high (97.7 %) automatic proof rate
- Improve the automaticity of the proof in presence of bit-wise operations
- Automatic search of WP built-in proof strategies to be applied

Unit-Test: develop heuristics for deducing the test cases from the DCSL contracts.

Skills for performing the activities:

Reinforce the developer's ability to master the writing of formal contracts from non-formal upstream artefacts (specification and architecture documents).

Note: skills in specification, design and coding and their relative levels of abstraction are strong prerequisites,

independently from the notion of formalisation. What is often observed is that issues in the formalization of function contracts result from shortcomings in the "art of specifying" in general, rather than in the difficulties inherent to the formal language.

5. Conclusion

The balance presented in section 4 above is globally very positive. The formal method based New Way Of Working is now a solid and important part of the standard Airbus development process of avionics software products. The lessons learnt so far are sufficient for being very confident in the result of the final and complete balance that will be made after certification of the two software products being developed.

References

[1] DO-178C: Software considerations in airborne systems and equipment certification, 2011.

[2] DO-333 formal methods supplement to do-178c and do-278a. Technical report, December 2011.

[3] AdaCore. SPARK 2014 Reference Manual. http://docs.adacore.com/spark2014-docs/html/lrm/, 2013.

[4] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language - Version 1.7. CEA/LIST & INRIA, 2013.

[5] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Muller, and Matthew Parkinson. Behavioral Interface Specification Languages. ACM Computing surveys, 44(03):16:2-16:58, 2012.

[6] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Muller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. JML Reference Manual. http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman.html#SEC_Top, 2013.

[7] Francesco Logozzo. Code Contract User Manual. https://github.com/Microsoft/CodeContracts/blob/master/Documentation/User%20Documentation/userdoc.pdf, 2013.

[8] Abderrahmane Brahmi, David Delmas, Mohamed Habib Essoussi, Famantanantsoa Randimbivololona, Abdellatif Atki, et al.. Formalise to automate: deployment of a safe and cost-efficient process for avionics software. 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Jan 2018, Toulouse, France.

[9] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert -- a formally verified optimizing compiler. In ERTS 2016: Embedded Real Time Software and Systems.

[10] A. Brahmi, F. Randimbivololona, P. Le Meur, T. Marie, R. Beseme, Final integration test of avionics software in full virtual platform, In ERTS2, Toulouse, France, 2014.