

# Formalise to automate: deployment of a safe and cost-efficient process for avionics software

Abderrahmane Brahmi,  
David Delmas,  
and Mohamed Habib Essoussi  
Airbus Operations S.A.S.  
316 route de Bayonne  
31060 Toulouse Cedex 9, France  
First.Last@airbus.com

Famantanantsoa Randimbivololona  
CEPRESY Informatics  
La Nauzère  
32500 Castelnau-d'Arbieu, France  
First.Last@orange.fr

Abdellatif Atki  
and Thomas Marie  
Ausy  
4 rue Pierre Vellas  
31300 Toulouse, France  
aatki@ausy-group.com  
tmarie@ausy-group.com

**Abstract**—For over a decade, Airbus have been introducing formal techniques into the verification processes of some of their avionics software products, to cope with the steady increase of the size and complexity of related avionics systems. These techniques have come of age for large-scale industrial deployment. All design and verification processes are currently being revised to take maximum advantage from them, i.e. improve industrial efficiency while maintaining the safety and reliability of avionics systems.

To achieve this goal, all human-engineered design artefacts are being formalised using languages with well-defined syntaxes and semantics, in order to allow for the automatic generation of all subsequent, computable design or verification artefacts, and the preparation of the input data for non computable activities.

To this aim, several domain-specific languages and related compilers have been developed internally, which cover all design activities, and bridge the gaps to integrate external tools into the overall development processes, e.g. sound, semantics-based, static analysis tools. For instance, the formalisation of detailed designs in the form of function contracts expressed in a first-order logic-based language allows for a hybrid approach to unit verification. Designs may be compiled down to ACSL [5] contracts, allowing for program proof with Frama-C [22], or they may be compiled down to test contracts, allowing for semi-automatic unit tests.

**Keywords**—*design, formalisation, domain-specific languages, compilation, static analysis, formal methods, avionics software, DO-178C, development process, industrial application*

## I. INTRODUCTION

Avionics software running on on-board computers are critical components of the systems of civil aircraft. They are thus subject to certification by Certification Authorities, and developed according to stringent rules imposed by the applicable DO-178/ED-12 [1] international standard.

Among the many processes described in DO-178, verification processes are responsible for more than half of the overall costs of avionics software development. Considering the steady increase in size and complexity of this kind of software, classical V&V processes, based on massive testing campaigns and complementary intellectual reviews and analyses, no longer scale up within reasonable costs.

On the other hand, some formal verification techniques have been shown to cope with real-world industrial software, while improving automation. For a decade, Airbus have therefore been introducing such techniques into the verification processes of some internal avionics software products [39], in order to replace or complement legacy methods. Significant effort is currently being invested into generalising this approach to most development and verification processes. Nonetheless, the current state of the industrial practice is that completely formal development methods are still not quite tractable for most avionics software products.

Therefore, a hybrid approach has been developed to integrate design activities with static analysis, program proof, and testing. The goal is to take maximum advantage from formal methods, i.e., improve industrial cost-efficiency while maintaining the safety and reliability of avionics systems. In this approach, all human-engineered design artefacts are being formalised using dedicated domain specific languages. Internally-developed compilers automate many subsequent development or verification activities, and generate all relevant data for non computable ones, such as verification of correctness. In particular, these compilers bridge the gap from designs to formal verification tools. This approach is made industrial reality thanks to the integration of a set of tightly-coupled tools into a new development workshop supporting advanced dependencies. This workshop, as well as the development process it supports, is currently being deployed on all new avionics software development projects.

This paper aims at describing this process, and supporting workshop. Section II presents the current industrial context, in particular legacy development processes. Section III introduces the new process, with associated industrial objectives. Section IV describes the languages, workshop and tools supporting the new process. Section VI addresses deployment and industrial results. Section V describes related work. Section VII describes future work and concludes. An extended version of this paper, including a running example based on a real-world avionics use case, can be found at <http://www.di.ens.fr/~delmas/erts18/>.

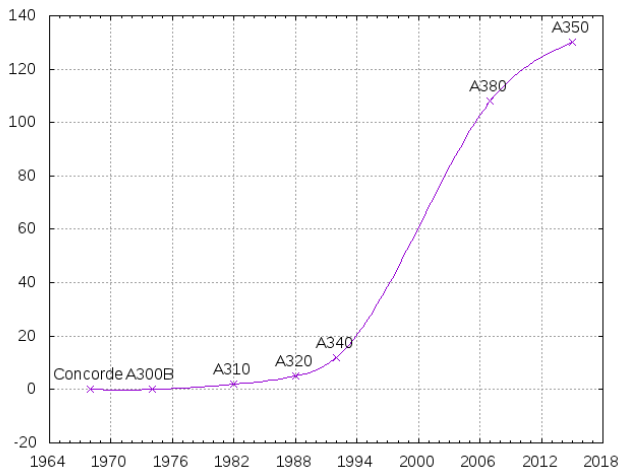


Fig. 1. Airbus avionics software (MB)

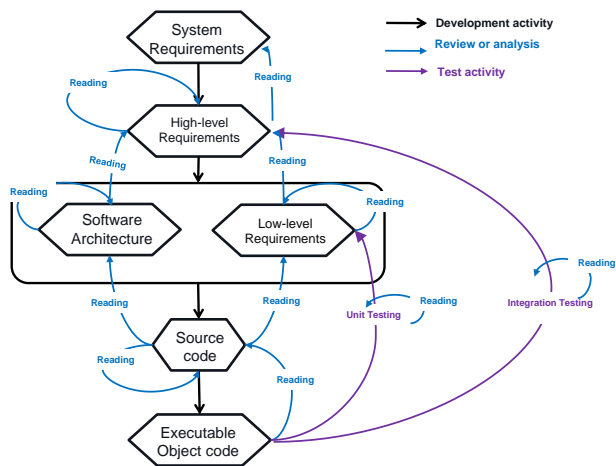


Fig. 2. Legacy process for DAL A

## II. INDUSTRIAL CONTEXT

The workshop presented in this paper is primarily designed to improve the industrial efficiency of software development processes for cockpit avionics systems, such as fly-by-wire, warning, communication, and maintenance. Such software is written in C and assembly, and encompass all DO-178 *Development Assurance Levels* (DAL): fly-by-wire, warning, or communication software are typically certified according to DAL A to C, while maintenance software usually enjoys DAL D or E.

Over the past four decades, most such aircraft functions have been transferred from hardware to software, with increasing requirements on process assurance. This trend, together with the growing effort to optimise passenger comfort, as well as system configurability and interoperability, has resulted in an exponential growth of the size and complexity of this kind of avionics software, as depicted on figure 1.

The state of the industrial practice, on the other hand, has evolved at a much slower rate during the same time frame. Most avionics software processes are still based on informal specifications and designs, pair reviews, and hand-written test procedures based on equivalence class partitioning.

As shown on figure 2, all artefacts produced by development processes, *i.e.*, in DO-178 terminology, *High-Level Requirements* (HLR), architecture, *Low-Level Requirements* (LLR), source and executable code, are verified several times. All of them are reviewed for compliance with artefacts from which they are derived on one hand, and for accuracy, consistency, hardware compatibility and conformance 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.

LLR provide very detailed specifications for every individual C function or assembly routine, down to the specification of every single procedure call or volatile access. These specifications are typically expressed in informal pseudo-code. In this context, significant effort is invested into unit testing, which roughly amounts to perform grey-box testing of every individual C function or assembly routine on the target hardware, to ensure that it implements the given algorithm correctly. Indeed test scenarios and expected values of outputs are derived completely by hand, from a intellectual reinterpretation of the informal LLR. Then, the correctness of this reinterpretation and the correct implementation of associate scenarios are verified in additional pair reviews.

Despite the automatic compilation of test scripts into target programs for part of the test procedures, the largest part of these heavy design and verification processes is essentially hand-crafted, and relies completely on human expertise. Unfortunately, such legacy processes do not scale up to current avionics software size and complexity within reasonable costs, hence cost issues in both development and maintenance phases. In particular, verification is liable for a steadily growing share of the overall development costs. The current status is about 70%.

To address this economical risk, Airbus have been introducing formal techniques into the verification processes of several internal avionics software products for a decade, in order to replace or complement legacy methods. Such techniques improve automation, while preserving safety. For instance, Some program proof [19] and static analysis [38], [15], [36], [17], [14] techniques have been introduced here and there on some avionics projects. However, their impact on industrial efficiency has been limited so far by a lack of formalisation of design processes, by an incompatibility of previous testing frameworks with formal methods, and by a lack of interoperation between tools, resulting in significant efforts dedicated to preparing inputs for formal verification processes.

## III. $\nu$ WoW: OBJECTIVES AND ORIENTATIONS

Airbus is currently investing significant effort into an internal initiative known as *New Ways of Working* ( $\nu$ WoW), which aims at improving the industrial efficiency of avionics software development processes, while maintaining the highest standards for safety. The major industrial goal of  $\nu$ WoW is to reduce the cost of design and unit verification processes by 75%.

The way to achieve this ambitious target is to move from hand-crafted legacy processes to an automated process. The  $\nu$ WoW approach to automation is language-based. Domain-specific languages with well-defined syntax and semantics

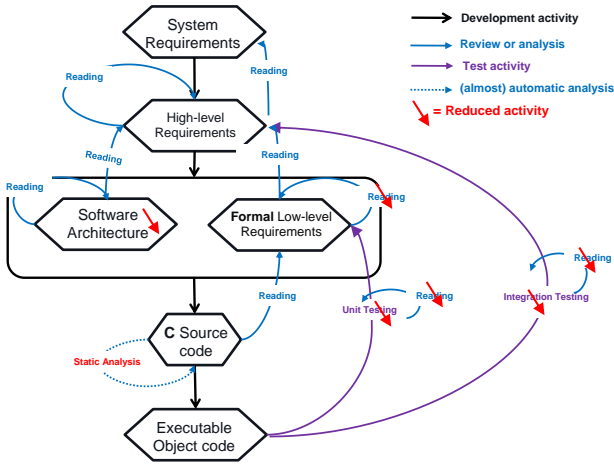


Fig. 3.  $\nu$ WoW process with unit test

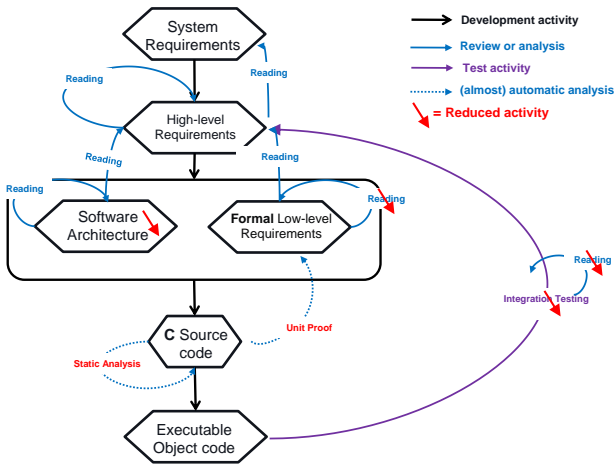


Fig. 4.  $\nu$ WoW process with unit proof

have been created, to enable the formalisation of all design artifacts. Dedicated compilers have been developed, so as to allow automatic, safe computations on explicit, unambiguous design data.

Figures 3 and 4 give an overview of the  $\nu$ WoW automated process. The design phase, presented in the central box, is deeply revisited. In legacy processes, this phase was mostly a preparation for the coding phase, producing informal documentation expressing LLR in the form of pseudo-code, and their traceability to HLR. In the  $\nu$ WoW approach, the design phase is extended to prepare all unit verification and static analysis activities, as well as part of integration activities:

- the Software Architecture is described in a dedicated domain-specific language, which will be presented in section IV-B. 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 ex-

actly 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 are expressed in a first-order logic based language, which will be presented in section IV-C. They describe the observable behaviours of unitary procedures, to be implemented in C or assembly, knowing the annotated interfaces of callees, but not their behaviours. This language is reminiscent of standard Behavioural Interface Description Languages (BISL). Functional requirements on procedure behaviours are formalised 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 formalised 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. Moreover, its scope is not limited to DO-178 compliant processes, as it allows for test-driven developments or re-engineering: see IV-C1.

Obviously, a first advantage to such design formalisation is a significant reduction of reviews of design data for accuracy, consistency, or conformance to standards. Nonetheless, the key interest is that it allows for a hybrid approach to unit verification: either unit proof or unit test can be used, as shown on figures 4 and 3. 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 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;
- most unit verification support data are computed automatically from formalised designs. The only human-engineered inputs are:

for unit test: scenarios providing values for input variables. The set of input variables is precomputed for each behaviour from formal LLR. Also, test oracles are first order predicates directly extracted for LLR to be evaluated at run-time. This obviously decreases the necessary effort both in test procedure development and reviews, compared to legacy methods.

for unit proof, which will be presented in section IV-D:

- 1) loop invariants
- 2) proof tactics for verification conditions that underlying SMT-solvers fail to prove automatically.

Besides, integration testing is alleviated, as some properties that could only be verified in integration tests with legacy processes, *e.g.* volatile accesses, are verified at unit level, either by proof or by test on virtualized hardware.

Moreover, additional process optimisations are obtained in the case of C source code. Reviews for conformance of source code to LLR are eliminated where unit proof is applicable, thanks to the exhaustiveness of this verification technique. 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 conformance to standards are automated by syntax-based static analysis tools. Reviews for accuracy and consistency are automated by semantics-based static analysers by abstract interpretation. Note that the traceability analysis between source and compiled code, usually required for DAL A, is also avoided thanks to the use of a formally verified optimising C compiler: see section IV-G.

This process is the outcome of many years of industrial practice. it is made efficient by the tight integration of a number of automated techniques. This integration is orchestrated by a process management tool, which will be presented in section IV-A. For instance:

- the software architecture compiler generates code and documentation, and passes information on to the LLR compiler;
- the LLR compiler generates unit proof or test contracts and associate verification environments;
- the proof engine is used to check design completeness, even when the implementation is tested;
- the LLR compiler generates expected variables ranges and data-flow to be verified by static analysis tools;
- functional ranges guaranteed by static analysis enable to focus the scope of unit verification to reachable input ranges;
- some static analysers compute configuration files for other static analysers, *e.g.* a points-to analysis on source code resolves computed calls, which enables a stack analysis on machine code;
- static analysis tools validate assumptions or eliminate warnings from design or unit verification tools.

As a consequence, code and design reviews are mostly limited to tool warning analyses.

In addition, this tight integration allows for significant savings in tool qualification costs. Indeed, DO-178 compliant qualification strategies typically require the definition of a set of user contexts for which given tool is qualified. This context includes many user-specific tool parameterisations, but also requirements on the machine on which the tool may be run, *e.g.* processor, operating system, libraries, etc. In practice, this approach leads, in legacy processes, to a combinatorial explosion of the number of contexts to be documented and maintained. All qualification procedures, many of which are at least partially hand-crafted, are to be replayed in all candidate contexts. Strong requirement on host machine environments limits the set of machines on which tools may be run, and result either in the need to maintain or virtualize obsolete systems for legacy projects, or to invest in costly migrations. In the  $\nu$ WoW, tool

qualifications are also managed by the same process management tool as development and verification artefacts. As consequence, the  $\nu$ WoW process can be deployed over a cloud of heterogeneous machines. Whenever an operational process requires the use of qualified tool with a given set of options on a given machine, the process manager detects whether the tool is already qualified for this context, and launches qualification tests automatically for this context if it is not the case.

#### IV. $\nu$ WoW: IMPLEMENTATION

Figure 5 provides a high-level data-flow view on how the  $\nu$ WoW process is implemented on a given avionics project. The  $\nu$ WoW process has three tool-supported processes:

- 1) the design process;
- 2) the coding and static analysis process;
- 3) the unit verification process.

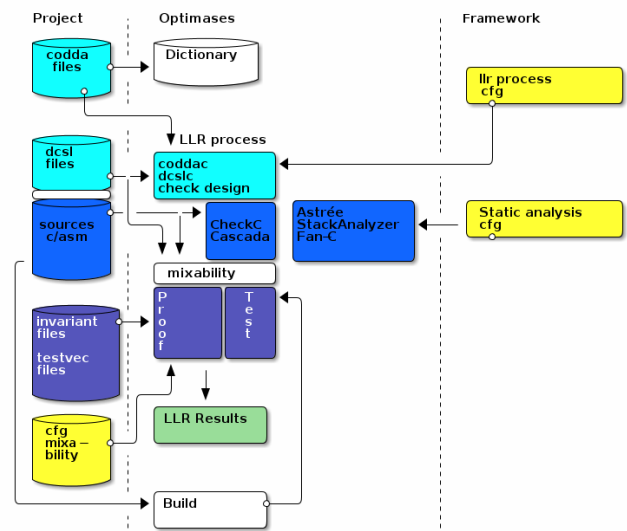


Fig. 5. Optimases for process management

In the design process (cyan boxes of figure 5), developers formalise the Software Architecture and associate LLR using the IDE of the CoDDA tool – see IV-B3. The output is a set of CoDDA files describing the Software Architecture, and a set of DCSL files describing LLR.

The CoDDA IDE interacts with companion design compilers *coddac* and *dcslc* through *Optimases*, the process automation tool. In particular, *coddac* computes a database from CoDDA files, aka dictionary, as depicted on figure 5. This dictionary describes all design objects, as well as relationships between them. This information is the primary input for *Optimases*, the process automation tool. Indeed the database contains all the dependency information that is necessary to instantiate the process for the target project. As a consequence, *Optimases* bootstraps itself to construct the whole project dependency tree from the design dictionary.

As part of the design process, *Optimases* interoperates *coddac* and *dcslc* to generate source code templates and headers, check for design rules dependant on both architecture and LLR, as well as to produce data flow specifications.

As part of the coding and static analysis process (blue boxes of figure 5), auto-generated code templates are filled in by programmers to produce C or assembly implementations, and compiled via a build process. **Optimases** triggers verification processes for coding standard conformance (**CheckC**), absence of run-time errors (**ASTRÉE**), correctness of data flow with respect to specifications (**Fan-C**), compliance with the Software Architecture (**Cascada**), and absence of stack overflow (**StackAnalyzer**).

As part of the unit verification process (violet boxes of figure 5), **Optimases** supports a hybrid approach to verification of implementations against LLR, either by proof or by test. Inputs of unit proofs are source files and hand-crafted loop invariants. Inputs of unit tests are compiled sources and test vectors. The coverage of verification is ensured by a (yellow) “mixability” configuration file: for every source file, every function or procedure is either proved correct, or tested according to the applicable structural coverage criterion.

More details on sub-processes and supporting tools will be provided in the next subsections.

### A. *Optimases*

**Optimases** is both an process management tool and a build system.

*A process management tool:* **Optimases** allows to configure complex processes easily. A process is fully defined in XML collections. It can be easily updated by adding/removing tool(s), fixing issue, or inserting new collection. Even if XML is not the most human readable format it allows to ensure the well formatted collection through XML scheme (*xsd* file) and offers the way to extract automatically some smart information about process (Tool dependency, collection’s process view, full avionic process view...) using *xslt* sheet.

The **Optimases** configuration is based on:

- **file’s type:** File are all typed using pattern definition.
- **tool definition:** Allow to link inputs to outputs (based on type definition).
- **variables use:** Variable are mainly required for genericity goals. They allow to manage several level of configuration (workshop, project).
- **template definition:** Templates provide the way to program complex process (with several step between user’s sources and result). The user has just to apply this template with their inputs regardless the intermediate state.
- **variants definition:** A variant allows to derivate main process in an other context (debug, coverage).

*A build system:* **Optimases** is inspired by **tup** [40], though constraints on the definition of build rules are less strict:

- use of a database to manage its dependency tree.
- explicit dependencies specified by user.
- implicit dependencies detection using file access detection (libfuse [32]).

In addition, we have added some useful features:

- the dependency arrows go both up and down (make uses down, tup uses up).

- File freshness detection based on SHA1. Decrease the build action by stopping the process as soon as a built action has no effect on this output.
- The **Optimases** build directory can be fully out-sourced, so **Optimases** can be seen as distributed process tool management.

*Optimases in <sup>v</sup>WoW:* **Optimases** (and this collections package) is the conductor of then <sup>v</sup>WoW. It makes a complex process easy to use for developers. Once the main process is defined, it allows to configure specific cases without effort. For instance, the unit test of functions calling inline functions requires automatic prototype adaptations. Also, the automaticity of unit proofs of functions accessing volatile locations is optimised dynamically, choosing the most efficient instrumentation mode from design information computed by the `deslc`.

**Optimases** provides also an alternative debug variant (derived from the main process). This variant is not relevant to certification, but critical for productivity (decrease of investigation time). It is fully automatised. If one **Optimases** step failed, the developer can replay it in debug mode and investigate. **Optimases** provides for instance:

- Object/binary compiled with debug information.
- Automatic debug script (break point on error) for unitary test.
- Launch debug GUI (Interactive Proof Editor or GDB client for test on a virtual platform).

At user level, **Optimases** allows to have the whole LLR process automatically computed and so already up to date, following the project’s design upgrade.

### B. *Codda*

1) *Introduction:* **CoDDA** (Compilable Design Description Assistant) is a framework implementing the method of software static design by abstract machines. This method is an adaptation of the HOOD [37] method to the avionics development context.

**CoDDA** provides a domain-specific language with a well-defined syntax and a set of functionalities, such as automatic generation of code skeletons and design documentation, design consistency verification.

2) *The CoDDA Language:* The **CoDDA** language (**CoDDAL**) enables designers to formalise the descriptions of so-called machines (a.k.a modules) featuring exported interfaces and hidden implementations. So-called non-terminal machines describe in fact only interfaces for other machines to use. Non-terminal machines are refined into children machines, until so-called terminal implementable machines are reached. Elementary design objects are named types, constants, resources and services. **CoDDAL** is supported by a dedicated, internally developed compiler.

3) *CoDDA IDE:* **CoDDAL** is supported by an IDE providing indentation, syntax highlighting, cross navigation between references and their declarations (**CoDDA**, **DCSL** and **C**), graphical interface to **CoDDA** functionalities (e.g. code skeletons generation), snippets and a relevant auto-completion. Auto-completion is implemented by using a database (the dictionary) in the back-end generated and updated by the **Optimases** processes.



4) *CoDDA Checker*: The CoDDA Checker is integrated into the development process to ensure that a set of rules is met by the static design of the software.

For instance, the Checker warns if a machine declares a service in its exported interface, but no other machine uses this service.

5) *CoDDA in the  $\nu$ WoW: Software Architecture*: Based on a well defined static design of a software, derived from the HLR, CoDDA provides major activities (or features) part of the  $\nu$ WoW process.

The CoDDA compiler ensures the correctness of the static design w.r.t. the CoDDAL syntax, and computes the Abstract Syntax Tree (AST). This latter is used by the CoDDA compiler in order to:

- generate the dictionary (a.k.a database) with all the objects of the static design. This dictionary has a very important place in the whole  $\nu$ WoW process and its activities. For example, Optimases uses the dictionary to manage the dependencies between services and their implementing modules. Which is relevant in order to enable DCSL contract compilation or unit proof.
- automatically generate header and include files used by DCSL compiler, unit verification and code compilation. For example, ranges of bitfields are defined in the static design and used in the DCSL. Header files generated by CoDDA make them visible to the DCSL compiler. Skeletons of source code files imported and filled in by the user at coding time are also generated and can be automatically pre-filled with elements from DCSL contracts. These skeletons can also be automatically merged with existing code.
- generate the traceability report from the static design: Specific keywords and rubric are used by the user in order to link design elements to the HLR. The report combined with the one extracted from DCSL contracts is used by dedicated tools part of the  $\nu$ WoW process to check if all the design elements and detailed design behaviors are all traced and covered by the High Level Requirements.
- verify the coherence of the code w.r.t. the design. Note that, only elements of the code which represent objects of the design are verified, such as resources, types, services signatures, etc.

6) *CoDDA interface*: The diagram in figure 6 gives an overview of the CoDDA interface.

The main inputs are:

- CoDDA files: a CoDDA file can represent either the main machine, a terminal machine, or a non terminal machine described in CoDDAL.
- pre-code files: designers may provide coding hints for a service, to be included into code skeletons.
- dcsl files: DCSL contracts for services;
- traceability keywords: the set of keywords (e.g. #Link\_To) to be used to identify traceability information in the design.
- checker rules: the list of rules to activate when calling the CoDDA Checker. See IV-B4.
- DCSL compilation reports: for tasks as the generation of traceability report or for the CoDDA Checker, CoDDA needs information from the detailed design (in DCSL, see IV-C). This information is given as input to CoDDA and is provided by the DCSL Compiler. The consistency of the dependencies between the different tools (CoDDA Compiler and DCSL Compiler in this case) is ensured by Optimases.
- user code files: previous source code to be merged with updated CoDDA generated skeletons.

The main outputs of CoDDA are:

- code skeletons
- design documentation (HTML or PDF)
- traceability report
- design database: persistence of all entities and elements of the static design. This is the dictionary mentioned in different sections of this paper.
- checker report: used in the derogation phase (the user can give justifications of warnings). After the verification phase, Optimases gives a synthesis of all justified and unjustified warnings.
- merged code: user code merged with freshly generated skeletons. Whether or not merged source code files are imported to replace previously developed code is left for the user to decide.

7) *The Application Binary Interface (ABI)*: In real-world industrial software, performance and hardware constraints may require developing software functions (a.k.a services) which do not meet standard ABIs. For example, contexts of callers may have to be managed in an environment which does not handle call stacks nor memory accesses. Other functions may require a change of privilege or specific calling conventions. For these reasons, as part of CoDDA specific ABIs can be defined in the static design in order to specify:

- registers used in the calling convention between the caller and the callee: which registers are used for which function parameters and how values are returned and retrieved.
- how formal parameters are allocated.
- the calling sequence of the caller and the returns sequence of the callee.

This allows the user to make abstraction of these particularities and homogenize the use of functions during the design or test activities.

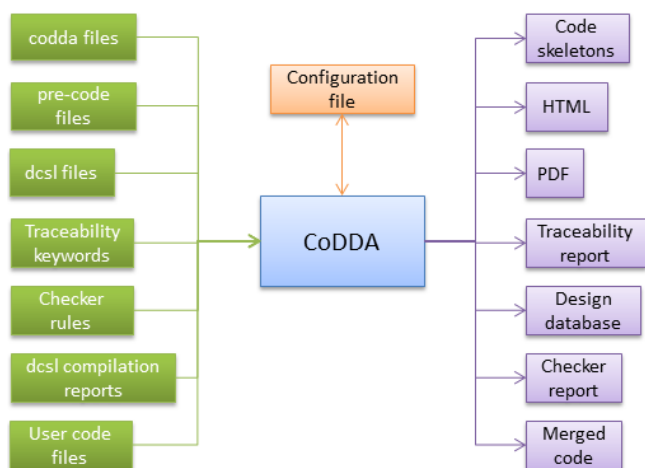


Fig. 6. Summary of CoDDA interface

### C. DCSL

1) *The language:* 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 [23] 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 [21], [27], [26], [35] approach shared by all code-level BISL's among whilst ACSL [6], JML [30], CodeContracts [33] or SPARK2014 [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:

- 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 E),
- 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,
- the engineering program may be a new development or a *WoW* rehosting of a mature (aka legacy) software.

A comprehensive presentation of the entire language is out of the scope of this paper. Instead, a running example available at <http://www.di.ens.fr/~delmas/erts18/> gives a concrete illustration of the solutions that have been chosen and implemented.

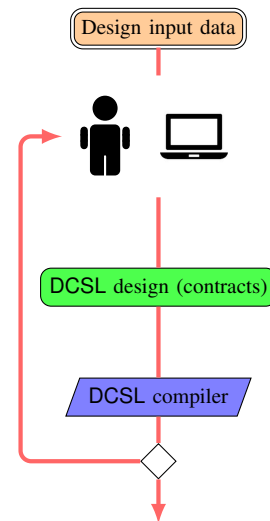
A DCSL specification program consists of two successive parts, the second of which is optional:

- 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, ...).
- 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 – essentially – 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

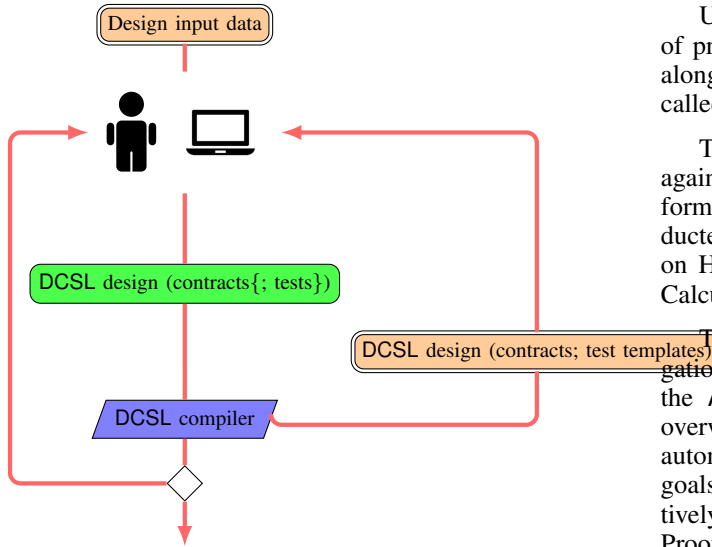
of program actions: calling a function, calling a function with specified values for parameters, reading a (volatile) variable, writing a (volatile) variable with a specified value. It is worth noting that formulas must be run-time evaluable. The quantifiers are restricted to finite integer intervals 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 contracts formulas. 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.

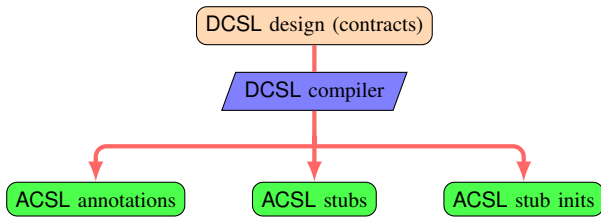
2) *Designing with DCSL and its compiler:* In the context of the *WoW*, the very first input when designing with DCSL is a CoDDA-pre-generated DCSL program which contains the design of the interfaces. Additional higher level specifications may be used by the designer to fully develop the formulas of the contracts. The DCSL compiler front-end 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.



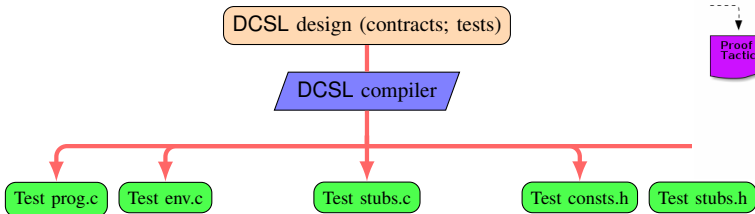
The design of the tests requires two DCSL design steps. The first step is the design of the contracts as formerly described. From this correct input, the compiler is run in a mode which will generate a DCSL program extended with templated tests. The identification of the formulas and the free variables that contribute to the tests is an intrinsic part of the generation. It make use of a dedicated semantic analysis guided by methodological rules. Thus the design of tests is the design of the models of the formulas, that is the design of the set of values to be substituted to the template placeholders, thus yielding the DCSL design with tests.



3) *Compile for proof*: The downstream unit proof process expects a set of ACSL formal annotations for its inputs. In legacy processes already including some proof techniques, such annotations were completely hand-crafted. In the context of the  $\nu$ WoW, they are produced automatically by the DCSL compiler from a semantically correct DCSL program. No further action of the designer is needed for the unit proof.



4) *Compile for test*: The downstream unit test process expects a set of C programs and declarations for its inputs. In legacy processes, such inputs were completely hand-crafted. In the context of the  $\nu$ WoW, they will be produced automatically by the DCSL compiler from a semantically correct DCSL program. No further action of the designer is needed for the unit test.



#### D. Unit proof

Since a decade, program proof techniques have been successfully introduced to replace unit testing on some software subsets, where the *Caveat* tool is used for certification credit [19]. This obsolete tool is currently being replaced by the WP plugin of the Frama-C [22] platform.

Unit proof as part of the  $\nu$ WoW stands for the restriction of program proof to the scope of an individual C function, along with a dedicated environment featuring stubs for callees and instrumentation of volatile accesses.

The implementation of a given C function is verified against a formal ACSL [5] contract, computed from its formal LLR by the DCSL compiler. The proof is conducted semi-automatically within a deductive system based on Hoare Logic [25] and Dijkstra's Weakest Precondition Calculus [18].

Throughout a proof campaign, bundles of proof obligations are generated by Frama-C/WP, and submitted to the Alt-Ergo [4] SMT-solver [12] (c.f. Figure 7). In the overwhelming majority of cases, proof is conducted fully automatically. Targeted rate exceeds 95% of entirety of goals. Nonetheless, proof engineers may terminate interactively some proof obligations by means of the Interactive Proof Editor. Complex goals generated by WP may then be decomposed into smaller pieces by applying several tactics available through the user interface. These tactics are saved and put to use automatically in the following proof campaigns. Proof engineers provide also loop invariants.

In addition, the proof engine checks the completeness of the LLR, in that it ensures every possible pre-state or the function is captured by at least one ACSL (hence DCSL) behaviour.

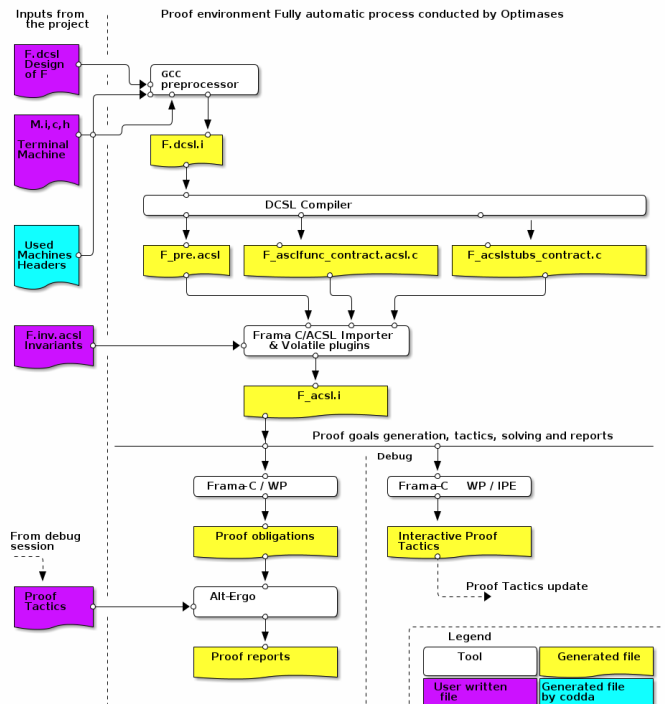


Fig. 7. Unit proof environment

The Figure 7 depicts, for a terminal machine  $M$  defining a C function  $F$ , the two automatic consecutive steps which are needed to carry out the unit proof of  $F$ :

- setup a proof environment conducted by Optimases which isolates the terminal machine implementing this function, by generating



F\_asclstubs\_contract.c: stubs for callees, and instrumentation for volatile accesses, if any;  
 F\_pre.acsl: initial values for instrumentation variables, in the form of ACSL preconditions;  
 F\_asclfunc\_contract.acsl.c: the ACSL contract, as translated from DCSL LLR.

This step is carried out mainly by the Airbus internal toolset. It also relies on Frama-C's ACSL importer plugin and GCC pre-processor.

- proof of generated ACSL postconditions by means of Frama-C/WP and Alt-Ergo.

The reader may refer to a long version of this paper at <http://www.di.ens.fr/~delmas/erts18/> for details on the automatically generated proof environment, in particular stubs for callees, and instrumentation of volatile accesses. It contains also information on the semi-automatic management of loop invariants.

### E. Unit test

1) *Tests design vs implementation:* The use of tools-supported formal methods all along with the automatization of the processes has enabled to separate the design of the tests from the implementation of the tests. The implementation of the tests with the tricky and error-prone activities is now totally tools-supported. The tester is focused solely on the design of the tests: design data sets which are models of the logical formulas at the prestate of the unit-under-test. He has no longer to develop test programs, simulated environments, test oracles, specific interfaces for assembly programs nor to manage test runs on specific-hardware test-platforms. All of these error-prone, full of tricky technical details activities are now totally handled by tools (DCSL compiler, Optimases, SIMUGENE).

2) *Process:* Figure 8 introduces the unit test process supported by Optimases. Testers fill in test templates computed by the DCSL compiler with input vectors. the rest of the process is automatic.

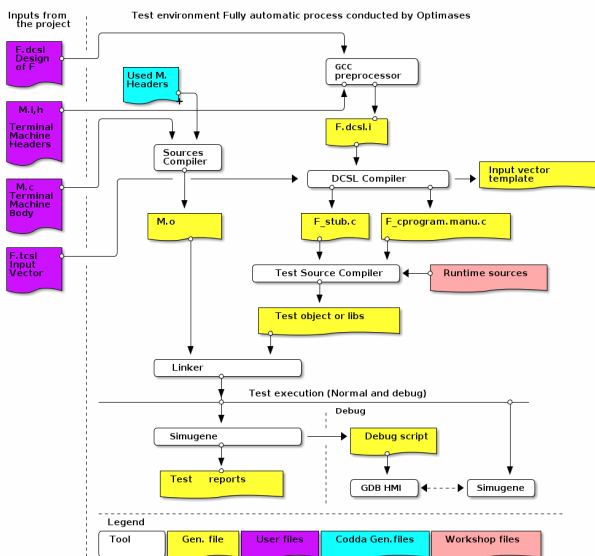


Fig. 8. Unit Test environment

3) *Practical implementation:* To reach this abstraction level for types of applications, in particular low-level applications, unit tests are run on the SIMUGENE representative virtual platform. The DCSL compiler generates calls to a dedicated run-time library, which cooperates with SIMUGENE's fault injection module [11], which handles:

- volatile variables The injection module is configured to both
  - 1) collect all information on write accesses to volatiles, e.g. written values, access numbers, etc.;
  - 2) inject user-defined values on read accesses to volatiles.

Hardware registers to prevent side-effects during the initialisation or capture of hardware registers, the injection module is used to ensure that registers may be safely used in the prologue of the procedure under test, and captured at epilogue, even in case of abrupt exit.

Management of non EABI routines In this case, a specific, non standard ABI is formalised to describe the call protocol of the routine – at entry and exit, the registers allocated to formal parameters and return codes, the volatile or non volatile nature of each register. The DCSL compiler generates calls to the fault injection module, so the tester does not need to do it. From the designer and test point of view, such non EABI routines are thus handled like simple C functions, with typed formal parameters and return value.

Special microprocessor instructions Some DCSL contracts described sequences of special processor instructions. The injection module allows the user to view them as standard functions, for which calls, inputs and outputs are to be monitored.

All these commandes and interactions are run without altering or instrumenting the software under test.

To improve the efficiency of the error detection and fix, a GDB script containing a breakpoint on every erroneous precondition and postcondition is generated automatically. Thus, a simple Optimases command lets the user relaunch a test scenario in debug mode. The test then stops on every error, and the GUI display the falsified predicate. the debug session may then continue either at DCSL contract level, or at the level of the C source files generated by the DCSL compiler, or at the level of the embedded software under test.

### F. Static analysis

The formal verification of non-functional properties requires semantics-based automatic analyses that scale up to very large programs. As a consequence, static analysis based on abstract interpretation [13] is currently used industrially for certification credit on many avionics software products developed at Airbus, to compute safe upper-bounds of stack consumption with AbsInt StackAnalyzer, or worst-case execution time [38], with AbsInt aiT WCET, or to verify data and control flows at unit level with the Fan-C [14] tool.

Moreover, the ASTRÉE [9] abstract interpretation based static analyser is being used industrially at Airbus, prior to certification, to prove the absence of run-time errors on safety-critical synchronous control/command programs [15] written in C. It has also been shown to be mature for use

on a large set of asynchronous, multithread avionics applications [36]. The FLUCTUAT static analyser has also been shown to be industrially usable to evaluate the numerical accuracy of floating-point libraries of control programs [17], and is in the process of being transferred to operational developers.

In addition to these semantics-based static analysers, an internal syntax-based rule checker [16] named CheckC is used to check C code for conformance with coding standards.

All these static analysers are integrated as part of the *WoW*, where they play a critical role.

Indeed

- 1) data flow specifications to be verified by Fan-C are computed by the DCSL compiler on formal LLR;
- 2) CoDDA, DCSL and Optimases generate inputs for the run-time error analysis, such as the ABI configuration, the set of inputs with their ranges, and expected ranges on outputs;
- 3) ASTRÉE validates strong assumptions used by Framac-C/WP to optimise proof automation (validity and separation of external pointers, absence of overflows in arithmetics or conversion), or similar implicit assumptions of unit tests;
- 4) ASTRÉE computes the set of functions reachable by any computed call, and generates the associate configuration file for StackAnalyzer;
- 5) FLUCTUAT proves round-off errors are small enough to rely on a real-valued semantics when proving C functions performing floating-point computations;
- 6) the coding rule checker limitates the subset of C that the programmer can use, which increases the automation of both semantics-based static analysis and unit proof.

### G. Certified compilation

After extensive experiments [8], [7], the CompCert [31], [10] formally verified compiler is currently being transferred to operational projects, to allow optimising compilation of DAL A software, while guaranteeing semantic preservation between source and compiled code. This guarantee enlarges the scope of sound source code analysers to also achieve verification objectives that traditionally require testing, or machine code analysis [2].

## V. RELATED WORK

Our approach to design by contracts is inspired by JML [29], and Microsoft research's CodeContracts [34]. However, we target C and assembly. Our hybrid approach to verification, combining testing, static analysis and program proof is similar to that of the Hi-Lite project [28], supported by AdaCore and CEA. We did not look into using SPARK, as Ada is not used on the avionics software products we develop. We could have considered e-ACSL, but we had already developed the internal TCSL solution on previous projects, so that it meets our needs exactly. Our use of domain-specific languages to improve productivity is related to Galois's endeavour [24], though we focus on automating the production of design and verification artefacts, rather the final product itself. Like in DARPA's HACMS project [20], we rely on domain-specific languages, formal verification, and certified compilation, though our main focus is correctness and safety rather than security.

## VI. DEPLOYMENT AND RESULTS

the *WoW* process and supporting development and verification workshop is currently deployed on two medium-sized development projects. No definite productivity measures have been established as of today, but there are strong indications of a significant improvement for the subprocess concerned with design and unit verification. Indeed, as demonstrated on a small (but real-life) running example available at <http://www.di.ens.fr/~delmas/erts18/>, about 80% of the design, code, and unit verification data is generated completely automatically by the design workshop. In addition, about 90% of the implementation can be verified by means of unit proof, and about 90% of the proofs are automatic (after loop invariant definition). In the case of unit testing, the level of automation is also tremendously improved versus legacy processes. Finally, for both verification techniques, the integration of advanced debug facilities to the development workshop yields key productivity enhancements.

Therefore, we are very confident to reach a productivity gain of 50% on the design and unit verification phases for these projects.

## VII. CONCLUSION AND FUTURE WORK

In this paper, we describe a development process integrating tightly formalised designs, static analysis, program proof, and testing. This process is supported by a workshop based on an internal process management tool maintaining dependency information between all development and verification artefacts, and allowing for contextual, automatic, dynamic adaptation of the verification process, and assisted transitions between development phases. This process, and supporting methods, tools and workshop, are intended to be deployed in all new projects. The next step is the deployment of the *WoW* on two large-size redesign projects: that of the Airbus Single-Aisle Flight Warning Computer, and that of ATR Multi-Function Computer. Some legacy projects are also considering migrating.

This hybrid process allows for the efficient cooperation of heterogeneous but complementary approaches to avionics software engineering. Up to now, it seems to keep its promise: allow for large improvements in cost-efficiency, while preserving the quality of avionics software.

However, the initial industrial target, which is to reduce the cost of design and unit verification processes by 75%, is not met yet. Extensions of the *WoW* are necessary to address residual shortcomings, and reach this target. For instance, unit testing methods could be reduced to functional ranges, provided sound static analysis guarantees no value is reachable outside these ranges. Limited extensions of the contents of design data could enable for the automatic generation of link scripts, and the automatic verification of memory mappings. Template loop invariants could be pre-generated to save time in the unit proof phase. Specific proof tactics could be integrated to further improve unit proof automation in a given context. Some preconditions on procedures still need to be validated in reviews. They could be validated by static analysis. Finally, static analysis could be used to check the consistency of test scenarios.

## 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] The alt-ergo smt solver. <https://alt-ergo.ocamlpro.com/>.
- [5] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA List, INRIA.
- [6] 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.
- [7] Ricardo Bedin Franca, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *Embedded Real Time Software and Systems (ERTS 2012)*, 2012.
- [8] Ricardo Bedin Franca, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards formally verified optimizing compilation in flight control software. In *PPES 2011*, volume 18 of *OASIScs*, pages 59–68, 2011.
- [9] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. *Foundations and Trends in Programming Languages*, 2(2-3):171–291, 2015.
- [10] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.
- [11] Abderrahmane Brahmi, Thomas Marie, Romain Beseme, and Faman-tantsoa Randimbivololona. Final integration test of avionics software in full virtual platform. 2014.
- [12] Sylvain Conchon, Évelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X): Semantical combination of congruence closure with solvable theories. In *Post-proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007)*, volume 198(2) of *Electronic Notes in Computer Science*, pages 51–69. Elsevier Science Publishers, 2008.
- [13] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, Jan. 1977.
- [14] Pascal Cuoq, David Delmas, Stéphane Duprat, and Victoria Moya Lamiel. Fan-C, a Frama-C plug-in for data flow verification. In *ERTS'12*. SIA, 2012.
- [15] D. Delmas and J. Souyris. Astrée: from research to industry. In *SAS'07*, volume 4634 of *LNCS*, pages 437–451. Springer, Aug. 2007.
- [16] David Delmas, Stéphane Duprat, Victoria Moya Lamiel, and Julien Signoles. Taster, a frama-c plug-in to enforce coding standards. In *ERTS 2010: Proceedings of Embedded Real Time Software and Systems*. SIA, 2010.
- [17] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *FMICS*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.
- [18] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
- [19] Stéphane Duprat, Denis Favre-Félix, and Jean Souyris. Formal verification workbench for airbus avionics software. In *ERTS 2008: Proceedings of Embedded Real Time Software*. SIA, 2006.
- [20] Kathleen Fisher. Using formal methods to enable more secure vehicles: Darpa's hacms program. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 1–1, New York, NY, USA, 2014. ACM.
- [21] Robert W. Floyd. Assigning Meaning to Programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.
- [22] The Frama-C framework for analysis of C code. <http://frama-c.com/>.
- [23] 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.
- [24] Patrick C. Hickey, Lee Pike, Trevor Elliott, James Bielman, and John Launchbury. Building embedded systems with embedded dsls. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 3–9, New York, NY, USA, 2014. ACM.
- [25] C. A. R. Hoare. Viewpoint - retrospective: an axiomatic basis for computer programming. *Commun. ACM*, 52(10):30–32, 2009.
- [26] Charles A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
- [27] Charles A.R. Hoare. An Axiomatic Basis for Computer Programming. *CACM*, 12(10):576–583, 1969.
- [28] Johannes Kanig, Edmond Schonberg, and Claire Dross. Hi-lite: the convergence of compiler technology and program verification. In Ben Brosgol, Jeff Boleng, and S. Tucker Taft, editors, *Proceedings of the 2012 ACM Conference on High Integrity Language Technology*, HILT '12, December 2-6, 2012, Boston, Massachusetts, USA, pages 27–34. ACM, 2012.
- [29] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.
- [30] 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](http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman.html#SEC_Top), 2013.
- [31] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [32] libfuse. <https://github.com/libfuse>.
- [33] Francesco Logozzo. Code Contract User Manual. <https://github.com/Microsoft/CodeContracts/blob/master/Documentation/User%20Documentation/userdoc.pdf>, 2013.
- [34] Francesco Logozzo. Practical specification and verification with code contracts. In Jeff Boleng and S. Tucker Taft, editors, *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology*, HILT 2013, Pittsburgh, Pennsylvania, USA, November 10-14, 2013, pages 7–8. ACM, 2013.
- [35] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.
- [36] A. Miné and D. Delmas. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In *Proc. of the 15th International Conference on Embedded Software (EMSOFT'15)*, pages 65–74. IEEE CS Press, Oct. 2015. <http://www-apr.lip6.fr/~mine/publi/article-mine-delmas-emsoft15.pdf>.
- [37] J.P. Rosen. *HOOD: An Industrial Approach for Software Design*. HOOD User's Group, 1997.
- [38] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, and Guillaume Borios. Computing the worst case execution time of an avionics program by abstract interpretation. In *In Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.
- [39] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546. Springer, 2009.
- [40] Tup. <http://gittup.org/tup>.