Qualifying CompCert for Safety-Critical Avionics Software

Markus Pister, Daniel Kaestner, Christoph Cullmann, Michael Schmidt, Bernhard Schommer, Christian Ferdinand, Sonia Sieurac, Christophe Cucuron, Jean Souyris

Qualifying CompCert for Safety-Critical Avionics Software

Markus Pister[1], Daniel Kästner[2], Michael Schmidt[3], Bernhard Schommer[4], Christoph Cullmann[5], Sonia Sieurac[6], Christophe Cucuron[7], Jean Souyris[8], Christian Ferdinand[9]

[1,2,3,4,5,9] *AbsInt Angewandte Informatik GmbH, Germany*

[6,7,8] *Airbus Commercial Aircraft, France*

ABSTRACT

CompCert is the first commercially available optimizing compiler which has been formally verified. The executable code it produces is proved to behave exactly as specified by the semantics of the source C program. As a consequence, the risk of system malfunctions due to miscompilation bugs can be considered eliminated. The correctness proof of CompCert C including a posteriori translation validation for assembler and linker step guarantees that all safety properties verified on the source code automatically hold for the executable object code as well. In this article we will outline the qualification strategy which is used at Airbus Commercial Aircraft ("Airbus" in the rest of the document) to apply CompCert in critical avionics software in compliance to DO-178C, DO-333, and DO-330. We will describe the application context and illustrate the advantages compared to the traditional way of qualifying and certifying compilers that has been used in the past.

Keywords: DO-178C, DO-330, DO-330, formal compiler verification, qualification, certification, Coq

1. INTRODUCTION

As of today, the CompCert compiler is the first commercially available optimizing compiler which has been formally verified. A compiler translates the source code written in a given programming language into executable object code of the target processor. Due to the complexity of the code generation and optimization process compilers may contain bugs. In fact, studies like [18, 4] and [22] have found numerous bugs in all investigated open source and commercial compilers, including compiler crashes and miscompilation issues. Miscompilation means that the compiler silently generates incorrect machine code from a correct source program. In case of CompCert, the absence of miscompilation bugs has been verified using machine-assisted mathematical proofs: the executable code it produces is proved to behave exactly as specified by the semantics of the source C program.

The formal verification offers numerous benefits beyond the obvious advantage that the risk of system malfunctions due to miscompilation bugs has been strongly reduced. Safety standards like [19] require developers of safety-critical systems to monitor compiler bug lists, and investigate reported bugs whether or not they constitute a risk to the deployed system. With CompCert, the effort of this task is significantly reduced. The testing effort required to ascertain software properties at the binary executable level can be reduced since the correctness proof of CompCert C including a posteriori translation validation of the assmbler and linker steps (see Section 3.5)) guarantee that all safety properties verified on the source code automatically hold for the executable object code as well. In the past, compiler optimizations had to be completely disabled for highly critical applications; with CompCert, however, optimized code can be used, enabling significant performance improvements. None of this is possible with the traditional way to qualify and certify compilers that has been used in the past.

CompCert has been on the market since 2015, providing over a decade of practical experience with commercial, highly safety-critical systems. In [15] we have given an overview of the design and the proof concept of CompCert and have presented an evaluation of its performance on the well-known SPEC benchmarks. In [10] we described the tool qualification strategy of CompCert in the context of an IEC 60880-certified industry application [7] in the nuclear power domain and reported on practical experience with replacing a legacy compiler by CompCert. The current article focuses on applying CompCert in critical avionics software in compliance to DO-178C, DO-333, and DO-330, and describes the design of the CompCert Qualification Kit that complies with DO-330 requirements.

Qualification Kits for AbsInt verification tools, in particular the abstract interpretation-based static analyzers Astrée, aiT, and StackAnalyzer [13] have been successfully used in various industry domains for years. While for static analysis tools a qualification level of TQL-5 is usually sufficient, for CompCert additional evidence is needed.

The focus of this article is on the qualification strategy used at Airbus to apply CompCert in critical avionics software, in compliance to DO-178C, DO-333, and DO-330. The article is structured as follows: in Sec. 2 we present related work. Sec. 3 summarizes the design and proof concept of CompCert. Sec. 4 outlines the development context in which CompCert is used at Airbus. Sec. 5 describes the resulting certification credits with respect to DO-178C. Sec. 6 and Sec. 7 detail a DO-330-compliant qualification approach for CompCert. The difference to previous approaches to compiler qualification is summarized in Sec. 8 and Sec. 9 concludes.

2. Related Work

Necula's translation validation approach [17] verifies the correctness of each individual compilation instance rather than the compiler itself. This is achieved by translating the program both before and after compilation into an intermediate language (IL) and performing equivalence checking on these representations. While conceptually attractive, equivalence checking is undecidable in general and may therefore lead to false alarms, which were reported for "about 10 % of the compiled functions" [17], despite the generated code being correct. In addition, the method incurs significant runtime overhead (a "factor of four" [17]) and requires the compiler to emit additional meta-information (so-called *simulation relations*) to establish equivalence between program fragments. In a safety-critical context, the correctness of the translation into IL, the computation of simulation relations, and the equivalence checking procedure itself would all need to be demonstrated.

CompCert follows the opposite strategy: it provides machine-checked proofs for all compilation passes, ensuring global correctness for all programs compiled by a given release. This correct-by-construction architecture eliminates false positives and aligns more directly with DO-330 qualification activities.

To the best of our knowledge, there is no previous publicly documented case of a C-compiler being fully qualified in accordance with DO-330. Existing compiler qualifications are typically limited to fit-for-purpose approaches, primarily relying on independently developed compiler test suites. While such test suites provide confidence in functional behavior for specific use cases, they are not derived from the compiler implementation itself. Consequently, they generally do not demonstrate the level of structural coverage of the compiler implementation required by DO-330 for higher Tool Qualification Levels.

3. CompCert – Design and Proof

Details about the design and proof concept of CompCert can be found in [15]. In the following we will give a brief overview to lay the foundations for the qualification concept described in Sec. 6. The input to the compilation process is a set of C source and header files. CompCert itself focuses on the task of compilation and includes neither preprocessor, assembler, nor linker. Therefore it has to be used in combination with a legacy compiler tool chain. Since preprocessing, assembling and linking are well-established stages there are no particular compatibility constraints.

CompCert reads the set of preprocessed C files emitted by the legacy preprocessor, performs a series of code generation and optimization steps and emits a set of assembly files enhanced by debug information. Debug information for functions and variables is generated in DWARF format, including information about their type, size, alignment and location. This also includes local variables so that the values of all variables can be inspected during program execution in a debugger. To this end CompCert introduces a dedicated pass which computes the live ranges of local variables and their locations throughout the live range.

The generated assembly code can contain formal CompCert annotations which can be inserted at the C code level and are carried throughout the code generation process. This way, traceability information, or semantic information to be passed to other tools can be transported to the machine code level. Since they are fully covered by the CompCert proof the information is reliable and provides proven links between the machine code and the source code level.

After assembling and linking by the legacy tool chain the final executable code is produced. To increase confidence in the assembling and linking stages AbsInt provides a tool for translation validation, called Valex, which performs equivalence checks between assembly and executable code [11].

3.1. Design Overview

CompCert is structured as a pipeline of 20 compilation passes that bridge the gap between C source files and object code, going through 11 intermediate languages. The passes can be grouped in 4 successive phases:

Parsing Phase 1 performs preprocessing (using an off-the-shelf preprocessor such as that of GCC), tokenization and parsing into an ambiguous abstract syntax tree (AST), and type-checking and scope resolution, obtaining a precise, unambiguous AST and producing error and warning messages as appropriate. The LR(1) parser is automatically generated from the grammar of the C language by the Menhir parser generator, along with a Coq proof of correctness of the parser [9].

C front-end compiler The second phase first re-checks the types inferred for expressions, then determines an evaluation order among the several permitted by the C standard. Implicit type conversions, operator overloading, address computations, and other type-dependent behaviors are made explicit; loops are simplified. The front-end phase outputs Cminor code. Cminor is a simple, untyped intermediate language featuring both structured (`if`/`else`, loops) and unstructured control (`goto`).

Back-end compiler This third phase comprises 12 of the passes of CompCert, including all optimizations and most dependencies on the target architecture. The most important optimization performed is register allocation, which uses the sophisticated Iterated Register Coalescing algorithm [6]. Other optimizations include function inlining, instruction selection, constant propagation, common subexpression elimination (CSE), and redundancy elimination. These optimizations implement several strategies to eliminate computations that are useless or redundant, or to turn them into equivalent but cheaper instruction sequences. Loop optimizations and instruction scheduling optimizations are not implemented yet.

Assembling The final phase of CompCert takes the AST for assembly language produced by the back-end, prints it in concrete assembly syntax, adds DWARF debugging information coming from the parser, and calls into an off-the-shelf assembler and linker to produce object files and executable files. To improve confidence, CompCert provides an independent tool,

called *Valex* (cf. Sec. 3.5), that re-checks the ELF executable file produced by the linker against the assembly language AST produced by the back-end.

3.2. The CompCert Proof

The CompCert front-end and back-end compilation passes are all formally proved to be free of miscompilation errors; as a consequence, so is their composition. The property that is formally verified is *semantic preservation* between the input code and output code of every pass. To state this property with mathematical precision, we give formal semantics for every source, intermediate and target language, from C to assembly. These semantics associate to each program the set of all its possible behaviors. Behaviors indicate whether the program terminates (normally by exiting or abnormally by causing a runtime error such as dereferencing the null pointer) or runs forever. Behaviors also contain a trace of all observable input/output actions performed by the program, such as system calls and accesses to "volatile" memory areas that could correspond to a memory-mapped I/O device.

As a first approximation, a compiler preserves semantics if the generated code has exactly the same set of observable behaviors as the source code (same termination properties, same I/O actions). This first approximation fails to account for two important degrees of freedom left to the compiler. First, the source program can have several possible behaviors: this is the case for C, which permits several evaluation orders for expressions. A compiler is allowed to reduce this non-determinism by picking one specific evaluation order. Second, a C compiler can "optimize away" runtime errors present in the source code, replacing them by any behavior of its choice. (This is the essence of the notion of "undefined behavior" in the ISO C standards.) As an example consider an out-of-bounds array access:

```
int main(void) {
  int t[2];
  t[2] = 1;  // out of bounds
  return 0;
}
```

This is undefined behavior according to ISO C, and a runtime error according to the formal semantics of CompCert C. The generated assembly code does not check array bounds and therefore writes 1 in a stack location. This location can be padding, in which case the compiled program terminates normally, or can contain the return address for "main", smashing the stack and causing execution to continue at PC 1, with unpredictable effects. Finally, an optimizing compiler like CompCert can notice that the assignment to `t[2]` is useless (the `t` array is not used afterwards) and remove it from the generated code, causing the compiled program to terminate normally.

To address these two aspects, CompCert's formal verification uses the following definition of semantic preservation, viewed as a refinement over observable behaviors:

If the compiler produces compiled code C from source code S, without reporting compile-time errors, then every observable behavior of C is either identical to an allowed behavior of S, or improves over such an allowed behavior of S by replacing undefined behaviors with more defined behaviors.

The semantic preservation property is a corollary of a stronger property, called a simulation diagram that relates the transitions that C can make with those that S can make. First, the simulation diagrams are proved independently, one for each pass of the front-end and back-end compilers. Then, the diagrams are composed together, establishing semantic preservation for the whole compiler. The proofs are conducted using the Coq proof assistant[1]. Coq gives us means to write precise, unambiguous specifications; conduct proofs in interaction with the tool; and automatically re-check the proofs for soundness and completeness. We therefore achieve very high levels of confidence in the proof. At 100,000 lines of Coq and 6 person-years of effort, CompCert's proof is among the largest ever performed with a proof assistant.

3.3. Code Structure

The CompCert source code splits into a formally verified part which is implemented in Coq and hand-written code which is implemented in OCaml. The hand-written parts implement the above mentioned *parsing* and *assembling* phase whereas the formally proven part deals with all other phases.

In total and at the time of writing this paper, CompCert source code roughly consists of 145K lines of code. About 80 % of that is (formally proven) Coq code.

3.4. Freedom From Undefined Behaviors

The absence of undefined behaviors in the source code can be shown by abstract interpretation-based static analyzers [3], such as Astrée [16, 14]. Astrée reports program defects caused by unspecified and undefined behaviors according to the C norm (ISO/IEC 9899:1999 (E)) [8], program defects caused by invalid concurrent behavior, checks coding guidelines, and computes further program properties relevant for functional safety. In addition, a generic non-interference analysis supports data and control coupling analysis, demonstrating independence between input and output signals, and detecting flow-based cybersecurity vulnerabilities [12].

3.5. Translation Validation

Currently the verified part of the compilation tool chain ends at the generated assembly code. In order to bridge this gap we have developed a tool for automatic translation validation, called *Valex*, which validates the assembling and linking stages a posteriori.

Valex checks the correctness of the assembling and linking of a statically and fully linked executable file P_E against the internal abstract assembly representation P_A produced by CompCert from the source C program P_S. The internal abstract assembly as well as the linked executable are passed as arguments to the Valex tool. The main goal is to verify that every function defined in a C source file compiled by CompCert and not optimized away by it can be found in the linked executable and that its disassembled machine instructions match the abstract assembly code. To that end, after parsing the abstract assembly code Valex extracts the symbol table and all sections from the linked executable. Then the functions contained in the abstract assembly code are disassembled. Extraction and

[1] Coq has recently been renamed to Rocq, http://rocq-prover.org

disassembling is done by two invocations of exec2crl, the executable reader of aiT and StackAnalyzer [13]. Apart from matching the instructions in the abstract assembly code against the instructions contained in the linked executable Valex also checks whether symbols are used consistently, whether variable size and initialization data correspond and whether variables are placed in the right sections in the executable.

4. Usage Context at Airbus

Since 2008, Airbus has been collaborating with INRIA first, then with AbsInt, in order to get an industrially mature compiler that satisfies the requirements of its most critical avionics software products, which includes both performance and certification / qualification aspects [5, 2].

From the beginning, Airbus' interest for the formally verified compiler CompCert has been to benefit from its (proved) optimizations and, more generally, its formal demonstration of semantic preservation, in the development of highly critical avionics software products.

The maturation of CompCert mainly consisted in (1) making it capable of compiling a full scale safety-critical avionics software product, (2) evaluating the performance of the compiled code of that product and (3) defining its certification credits and qualification strategy. These three major activities were conducted mostly in parallel and were supported by the RNTL research project VERASCO [1]. Point (3) was achieved in collaboration with EASA software experts.

Based on this work, Airbus decided to choose CompCert as primary C compiler and to develop preliminary versions of most of the qualification artifacts, which AbsInt subsequently refined into the material and strategy presented in this paper.

The criticality of the Airbus' software products being compiled with CompCert ranges from DAL (Development Assurance Level) A, the highest one, to DAL D, according to DO-178C. This underlines that CompCert can be used for the strong guarantees it brings to critical software development, but also as an ordinary compiler.

The qualification of CompCert in the development of the next generation of an avionics software product is currently being conducted at Airbus, based on the AbsInt's material described in this paper.

5. DO-178C Certification Credits

DO-178C [19] constitutes the primary standard used by certification authorities, such as EASA or the FAA, to assess safety-critical airborne software. Its companion document DO-330 [20] defines the qualification of software tools and, although developed alongside DO-178C, is a stand-alone standard that can also be applied independently beyond the avionics domain. By contrast, DO-333 serves as a formal methods supplement to DO-178C, providing additional guidance on the use of formal techniques within its certification framework.

A compiler is needed to generate executable object code (EOC) from the source code written in a higher-language source program. Claiming certification credits from a compiler means it can be used to eliminate, reduce, or automate specific objectives defined in DO-178C [19].

CompCert allows us to achieve the following objectives.

5.1. Source to Object Code Traceability

DO-178C indirectly requires to bi-directionally map source to object code for test coverage objectives at the executable object code level (see §6.4 in [19]). Furthermore, compilers may generate so-called *additional code* that is not directly traceable to source code statements, e.g., due to optimizations or robustness array bound checks. Such additional code complicates demonstrating DO-178C structural coverage objectives and hence is subject to additional verification according to **DO-178C, A-7.9** "*Verification of additional code, that cannot be traced to source code is achieved*".

CompCert's semantic preservation property fully covers DO-178C A-7.9 because the generated object code is proven not to change the semantics of the input program (including any additional code).

5.2. Semantic Preservation

CompCert fully covers **DO-333, FM.A-7.FM9** "*Verification of property preservation between source and object code.*" as a direct consequence of the semantic preservation theorem proven in Coq. That means that properties verified at the source code level also hold at the executable object code level.

As a consequence of the credit in DO-333, FM.A-7.FM9, the use of CompCert also contributes to **DO-333, FM.A-6.3** "*Executable Object Code complies with low-level requirements.*" and **DO-333, FM.A-6.4** "*Executable Object Code is robust with low-level requirements.*".

Therefore, the tool user may use this argument to perform formal verification on source code instead of testing the object code to achieve the objectives DO-178C A-6.3 and A-6.4. This however has to be embedded into the certification strategy of the safety-critical system and thus has to be decided in a project-specific way and under the approval of the certification authority.

6. DO-330 Qualification Strategy

In the context of system certification according to domain-specific safety standards (such as DO-178C for airborne systems and equipment), any tool that eliminates, reduces, or automates a software life cycle process without subsequent verification of its output requires qualification.

Most domain-specific standards define criteria for such tool qualifications. For example, DO-178C (Section 12.2) specifies how to determine the so-called tool qualification level (TQL), based on the tool's intended use, its impact on software life cycle processes, and its classification according to defined criteria.

DO-330 [20] builds upon the tool qualification criteria introduced in DO-178C and provides detailed guidance for tool qualification. It defines objectives and corresponding activities for each software life cycle process, aiming to demonstrate that a tool performs correctly within its intended operational context. The objectives and activities vary depending on the assigned TQL. As DO-330 is a standalone standard for tool qualification, criteria from domain-specific standards must be

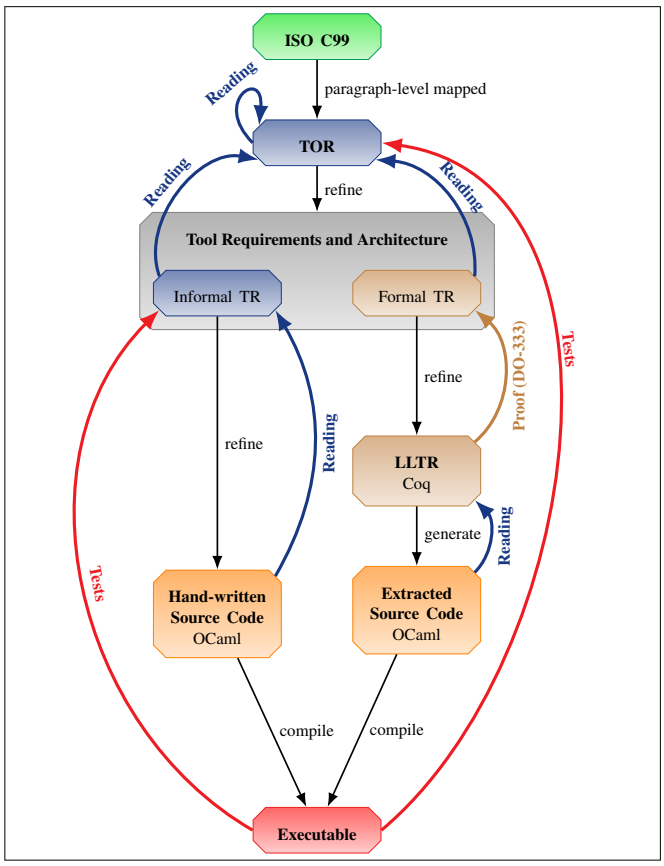

Figure 1. CompCert Qualification Structure

mapped to those defined in DO-330. For guidance, DO-330 repeats DO-178C's tool assessment criteria (Section 1.5.3.3.4, Annex D). DO-333 [21] extends and tailors the objectives defined in DO-178C, specifically regarding the use of formal methods.

The tool development of CompCert complies to DO-178C under the considerations of the formal methods' supplement DO-333. Therefore, CompCert can be qualified in accordance with DO-330. DO-333 specifies activities which can be performed using formal methods. Formal verification using Coq proofs is applied as a verification method to demonstrate the correctness of CompCert.

Based on Section 12.2 of DO-178C, a compiler could be considered a *Criteria 1* tool because its output directly affects the executable software. Accordingly, qualification would need to achieve TQL 1 to TQL 4, depending on the software level (Level A to Level D). However, compilers are typically not trusted and therefore not qualified. Instead, compliance with all safety properties is demonstrated at both the source and object code levels.

The certification credits provided by using CompCert are *verification* credits (see Sec. 5), which *reduce* the need for testing activities at the executable object code level: verification by unit testing can be partially substituted by leveraging formal proofs. Intensive integration testing, in which the generated object code of the entire software system is verified, is still performed. Consequently, any safety risk in CompCert arises only from the potential failure to detect an error in semantic preservation.

Based on these considerations, we propose to classify CompCert as a *Criteria 2* tool, for which TQL 4 would be sufficient for Level A software. However, given the criticality of correct source-to-object translation, we recommend TQL 3, together with corresponding alleviations in development standards and structural coverage objectives.

In the following, we describe qualification artifacts targeting a DO-330 qualification at TQL-3 with alleviations in development standards, structural coverage objectives and the absence of low-level tool requirements refinement for informal tool requirements. The following sections provide details about the alleviations.

These artifacts may be adapted or extended by tool users as needed to fulfill the objectives defined by the safety standard applicable in their domain.

6.1. Structure

In this section, we give an overview of our qualification strategy highlighting the most important verification aspects and activities with a particular emphasis on leveraging the CompCert proof (see Section 3.2) within the qualification, based on the provisions of DO-333.

The structure of the CompCert qualification concept is illustrated in Figure 1. CompCert is a compiler that accepts ISO C99 [8]. Therefore, the tool operational requirements (TOR), which address user-relevant aspects of the tool, need to cover the ISO C99 language definition. This is achieved by defining at least one TOR for each paragraph of C99, yielding a full paragraph-level coverage of the input language specification.

Refining from these TOR, we define *tool requirements* (TR), i.e., the requirements *"[. . .] used to develop and verify a tool [. . .]"* [20]. They can be grouped into an *informal TR* and *formal TR* (FTR) part.

The FTR address those parts of CompCert that have been specified in Coq (and thus are formally proven). They consist of the Coq specifications, i.e., the specification of the C and assembly semantics, as well as the notion of semantic preservation. The CompCert translation functions (defined in Coq, as well) represent the so-called *low-level tool requirements* (LLTR) for the formal TR. LLTR are refined from TR such that the *"[...] Tool Source Code can be directly implemented without further information [...]"* [20]. The Coq theorem prover allows to extract a program implementing the specification from which the extraction has been done. We use this feature to generate OCaml code from the LLTR to implement the translation functions.

The informal TRs directly map to hand-written OCaml code without LLTRs, i.e., in this case the informal TRs and informal LLTRs are equivalent. This constitutes an alleviation from TQL-3.

The tool source code (both hand-written and extracted from Coq specifications) is realized in OCaml and compiled to the final CompCert executable.

The TRs are reverse-engineered on one hand from CompCert Coq code in order to identify FTRs and formal LLTRs, and on the other hand from hand-written OCaml files in order to

identify informal TR. The formal LLTRs are (after the initial reverse-engineering) developed in a classical top-down approach. The reason of this reverse-engineering aspect is that the code already existed when starting the qualification activities. Furthermore, it enables full statement-level structural test coverage of the hand-written OCaml code (see Section 7.8).

Tool Operational Verification and Validation Cases and Procedures (TOVVCP) are tests against the TOR whereas the *Tool Verification Cases and Procedures* (TVCP) test against the TR. This elaborated below in Section 6.2.

As an example of an ISO paragraph which is refined via informal TR, let's look at ISO C99 § 6.5.2.5, paragraph 4, which addresses compound literals. The corresponding TOR is `TOR_COMPOUND_LITERALS_004`[2]. The TOVVCP covers this TOR by `TOVVCP_COMPOUND_LITERALS_001...012` (except `..._007` and `..._011`). These are tests that address different aspects of compounds literals (static/dynamic values, member selection, function argument, l-value, ...). The TR covers this TOR via informal TR `TR_Function_ELAB_183` which requires CompCert to correctly elaborate compound literal postfix operators in expressions. The TVCP covers this TR via the test cases `TVCP_ELAB_COMPOUND_LITERALS_001...003`. Those test cases address the same aspects of compound literals than the corresponding TOVVCP test cases in the hand-written code parts. TOVVCP test cases verify against TORs, where TVCP test cases verify against TRs.

The following example shows ISO paragraph § 6.5.6, paragraph 5 ("Additive operators") which is refined via formal TR. The corresponding TORs covering this paragraph are `BIN_OP_ADD_001` and `ADDITIVE_OPERATORS_005`.

The TOVVCP covers these two TORs by the test cases `ADDITIVE_OPERATORS_001`, `_004`, `_005`, `_008`, `_009`, `_011...016`, `_020`, `_023`, `_027`, `_029`, `_030`, and `_034...038`. They check the addition operator against all possible type combinations, check commutativity, associativity and addition of pointers. (Formal) TR covers the two TORs via

- `Binary_Bplus_001`,
- `Cop_classify_add_001`,
- `Cop_classify_add_cases_0_001`,
- `Cop_sem_add_001`,
- `Cop_sem_add_ptr_int_001`,
- `Cop_sem_add_ptr_long_001`,
- `Cop_sem_binary_operation_001`,
- `Cop_binarith_cases_0_001`,
- `Cop_binarith_type_001`,
- `Cop_classify_binarith_001`,
- `Cop_sem_binarith_001`,
- `Floats_Float32_add_001`,
- `Floats_Float_add_001`, and
- `Integers_Wordsize_add_001`.

These FTRs mark all Coq code parts that specify the behavior of the addition operator and therefore contribute to covering that TOR.

[2]Identifiers shortened and changed for simplicity

6.2. Verification

We perform the following kinds of verification activities (see Figure 1): The **Formal Proof** as sketched in Section 3.2 verifies the LLTR against the formal TR by means of the Coq theorem prover. This replaces traditional verification methods in compliance to DO-333 [21] and fully covers the verification of the LLTR against the formal TR.

The **Testing** stages are used to verify the CompCert executable at different levels: tool testing against the informal TR and tool operational testing against the TOR. At both levels, there is at least one test case for each particular requirement. This yields 100% test coverage for the informal TR and TOR. Each test case is assigned a unique requirement identifier which is documented, alongside the test case procedure and expected results, in the dedicated document: "*Tool Verification Cases and Procedures*" (TVCP) for the testing against informal TR and "*Tool Operational Verification and Validation Cases and Procedures*" (TOVVCP) for the testing against the TOR.

The hand-written code portions inherently carry a higher risk of introducing errors. This is mitigated by the above-mentioned requirement-based coverage of informal TR by test cases in combination of the reverse-engineering of informal TR from the hand-written OCaml code. The effort of both the reverse engineering and the test case coverage remain manageable because the hand-written code accounts for only about 20 % of the overall code base (see Section 3.3). Structural test coverage at statement level as demanded by DO-330 in TQL-3 is defined to cover low-level tool requirements. Since there are no informal LLTR, this is not applicable. Instead, there is a full requirements-based test coverage on informal TR.

The stages named **Reading** are associated with manual reviews of the following artifacts:

Reading	Effort hours:
TOR wrt. ISO C99	20
informal TR wrt. TOR	40
formal TR wrt. TOR	30
hand-written OCaml tool source code wrt. the informal TR	140
extracted OCaml tool source code wrt. the LLTR	70

All these review stages are based on dedicated check lists that address corresponding objectives from DO-330 and DO-333, including checks for compliance, consistency, accuracy, completeness, verifiability, just to name a few general aspects.

Each check list defines rules which determine the criteria for the particular review. Each rule is assigned a unique identifier to referencing in findings. Additionally, each rule refers to DO objective(s) or to verification activities (as defined in the tool qualification plan documents, see DO-330, Table T-1) to which the particular rule contributes to.

To give an intuition about the check list-based review approach, let's look into the details of the check list of the tool requirements Reading. Table 1 sketches the corresponding check list (simplified for space reasons). The rules are grouped as indicated by the grey headings. Each box with white background represents a rule alongside its description and mapping to the

TOR compliance
Each TOR is correctly implemented in one or several TRs. DO-330 T-3.1/4/5/8, Sect. 6.1.3.1a/d/e/h
Derived TRs are justified. DO-330 T-3.1, Sect. 6.1.3.1a
Informal Requirements Consistency
Each informal TR is complete, i.e., there is a complete description of the perimeter of the requirement. DO-330 T-3.1, Sect. 6.1.3.1a
Each informal TR is accurate and unambigous, i.e., precise terms are used. DO-330 T-3.1, Sect. 6.1.3.1b
Each informal TR is verifiable, i.e., the granularity is adapted to verification objectives (size of requirement is not too large). DO-330 T-3.1, Sect. 6.1.3.1f
Each informal TR is consistent at 2 levels, i.e., within the requirement itself and in the aggregation of all requirements. DO-330 T-3.1, Sect. 6.1.3.1b
Compatibility with operational environment
Requirements dedicated to the correct interoperability with C pre-processor, assembler and linker are defined and relevant. The same holds for the interoperability with verification tools that exploit CompCert's output data, i.e. annotations and debug information. DO-330 T-3.3, Sect. 6.1.3.1c
Change Management
The revision Table outlines precisely the reasons for change and impact of change (item modified). DO-330 T-0.2, Sect. CC1

Table 1. Sample reading check list

DO-330 objectives.

Any finding during a Reading is documented by a reference to a violated rule, a description of the finding and additional meta data like reviewer name, date, etc. A surrounding process ensures appropriate addressing of findings including a verification by the finding reporter.

The table above includes the time required for the Readings in hours to give an intuition about the effort. For minor updates of the tool, no full Reading needs to be performed again. Instead, a delta Reading can be done that only assesses the modifications and their effects. The efforts of such delta Readings depend on the size of the modifications, but are typically much smaller (below 5 hours). After a specific tool release has been submmitted for qualification, all the relevant code and qualification artifacts are fixed in a frozen version control branch. Changes are only caused by findings during the different assessments of the data in the scope of the qualification or because of the identification of a tool issue.

6.3. Traceability

Traceability between the different requirement artifacts is an important part of the DO objectives. To establish trace data and to allow their analysis, all requirements follow naming schemes. Additionally, syntax rules are established that allow to formally define a requirement and link to other requirements. By this, coverage analysis between the different requirement artifacts can be computed and assessed.

6.4. Qualification Package Contents

The qualification package for CompCert contains all qualification artifacts produced for that tool release (as detailed in Section 7). The most important content is:

- Tool life cycle plans: Tool Qualification Plan (TQP), Tool Development Plan (TDP), ...
- Compliance documents: Tool Accomplishment Summary (TAS), Tool Configuration Index (TCI)
- Requirements data: TOR, TR, ...
- Verification data: TOVVCP, TVCP, ...
- Trace data
- Test cases including execution framework

7. DO-330 Coverage

The following discussion refers to Objectives T-0.1 to T-10.4 as defined in Tables T-0 to T-10 of DO-330 [20]. Objectives that are either applicable only to levels higher than TQL-3 or that must be fully addressed by the tool user are omitted for space reasons. Each mentioned objective is categorized according to its coverage: partial coverage indicates that the tool user must complement the available data, whereas full coverage indicates that the objective is fully addressed.

7.1. Table T-0: Tool Operational Processes

Objectives T-0.2 ("Tool Operational Requirements are defined"), T-0.4 ("Tool Operational Requirements are complete, accurate, verifiable and consistent"), and T-0.5 ("Tool Operation complies with the Tool Operational Requirements") are fully addressed by the tool developer, i.e., AbsInt, through writing the TOR, conducting a TOR Reading review, executing the TOVVCP test cases and recording the results in TOVVR, as well as analyzing TOR coverage based on computed trace data.

Objectives T-0.6 ("Tool Operational Requirements are sufficient and correct") and T-0.7 ("Software life cycle process needs are met by the tool") are partially covered, as the TOR Reading verifies the requirements against ISO C99 to contribute to the tool user needs. The tool user must extend this information in the user TOR document by checking the TORs against their own needs.

7.2. Table T-1: Tool Planning Process

Objectives T-1.1 ("Tool development and integral processes are defined") and T-1.2 ("Transition criteria, inter-relationships, and sequencing among processes of tool processes are defined.") are partially covered by providing tool developer-variants of the required plan documents, such as the *Tool Qualification Plan*, *Tool Development Plan* and others. These documents define all relevant processes of the tool life cycle in accordance with DO-330.

Completion of these objectives requires the tool user to provide corresponding user-specific variants of these planning documents, thereby defining the processes applicable in the tool-usage context.

Objective T-1.3 requires that "the tool development environ-

ment is selected and defined". This objective is fully addressed by the definition of the environment in the Tool Development Plan.

Objective T-1.4 ("Additional considerations are addressed") is partially covered by a corresponding section in the TQP, which must be complemented by the tool user.

No formal tool development standards are required as requested by Objective T-1.5 ("Tool development standards are defined"). This is mitigated for TQL-3 by specifying the Formal Tool Requirements (FTRs) and the Formal Low-Level Tool Requirements (FLLTRs) in the Coq language, which thereby serves as the design standard. Naming rules on requirements exist and are sufficient for requirement standards. Tool coding standards are not necessary because the hand-written OCaml code is reviewed to extract (via reverse-engineering) the informal TRs, and because OCaml's functional nature prevents common errors such as uninitialized variables or function pointer misuse. Additionally, the amount of (integration) testing still performed on the embedded software gives enough confidence to justify the alleviation.

Objectives T-1.6 ("Tool plans comply with this document") and T-1.7 ("Development and revision of tool plans are coordinated") are fully satisfied with respect to the planning artifacts produced for Objectives T-1.1 and T-1.2. All tool plan documents explicitly demonstrate compliance with DO-330 by mapping their content to the relevant objectives and by documenting the coordination and revision mechanisms applied during their development.

7.3. Table T-2: Tool Development Processes

Objectives T-2.1 ("Tool Requirements are developed"), T-2.2 ("Derived tool requirements are defined"), T-2.3 ("Tool architecture is developed"), T-2.4 ("Low-level tool requirements are developed"), and T-2.5 ("Derived low-level tool requirements are defined") are fully covered by reverse-engineering the TR from both hand-written OCaml code and the Coq specification.

Objective T-2.6 ("Tool Source Code is developed") is fully addressed through OCaml code extraction from the FLLTR (transition functions implemented in Coq) combined with maintaining the hand-written OCaml code from informal TRs. Objective T-2.7 ("Tool executable Object Code is produced.") is addressed by the OCaml compiler, which produces the tool executable object code.

Finally, Objective T-2.8 ("Tool is installed in the tool verification environments") is covered by recording the underlying verification environment during the tool build and testing processes, demonstrating in which environment these processes have been executed.

7.4. Table T-3: Verification of Outputs of Tool Requirements Processes

Objectives T-3.1 ("Tool Requirements comply with Tool Operational Requirements"), T-3.2 ("Tool Requirements are accurate and consistent"), T-3.3 ("Requirements for compatibility with the tool operational environment are defined"), T-3.4 ("Tool Requirements define the behavior of the tool in response to error conditions"), T-3.5 ("Tool Requirements define user instructions and error messages"), and T-3.6 ("Tool Requirements are verifiable") are fully addressed by a checklist-based Reading of the TR against the TOR.

There are no tool requirements standards defined that would constitute a TQL-3 alleviation. Nevertheless, Objectives T-3.7 ("Tool Requirements conform to Tool Requirements Standards") and T-3.8 ("Tool Requirements are traceable to Tool Operational Requirements") are fully addressed by means of a trace data analysis. This analysis ensures that all Tool Requirements (TRs) comply with the defined requirement naming scheme and verifies that each TR is traceable to at least one Tool Operational Requirement (TOR).

Finally, the accuracy of the algorithms, as required by Objective T-3.9 ("Tool Requirements are accurate"), is fully addressed by the reverse-engineering of the informal TR from hand-written OCaml files as described in Sec. 6.1 which is confirmed through reading the OCaml source code.

7.5. Table T-4: Verification of Outputs of Tool Design Process

The objectives T-4.1 ("Low-level Tool requirements comply with Tool Requirements.") and T-4.2 ("Low-level Tool requirements are accurate and consistent.") are fully covered by the application of the formal proof for FTR. No LLTR have been developed from the informal TR because informal TR got reverse-engineered from OCaml source code. This constitutes an alleviation from TQL-3.

There are no tool design standards defined that would constitute a TQL-3 alleviation for Objectives T-4.4 ("Low-level Tool Requirements conform to Tool Design Standards") and T-4.9 ("Tool architecture conforms to Tool Design Standards"). Even in the absence of explicit tool design standards, these objectives are satisfied by the formalization of LLTR and FTR in Coq. The Coq formalization inherently enforces a precise and unambiguous specification of both the architecture and the low-level requirements, thereby eliminating the need for additional design standardization of the Coq code.

Additionally, the tool architecture review (by Reading) is performed during TR verification, addressing T-4.6 ("Algorithms are accurate."), T-4.7 ("Tool architecture is compatible with Tool Requirements.") and T-4.8 ("Tool architecture is consistent.").

Concerning Objective T-4.5 ("Low-level Tool requirements are traceable to Tool Requirements."), there is no trace data between LLTR and TR because of the formal proof of formal TR versus formal LLTR and the absence of LLTR for informal TR. This fully addresses T-4.5.

T-4.10 ("Protection mechanisms, if used, are confirmed") is not applicable because no protection mechanisms are required as CompCert is no multi-function tool.

7.6. Table T-5: Verification of Outputs of Tool Coding & Integration Process

The objectives T-5.1 ("Tool Source Code complies with low-level tool requirements.") and T-5.2 ("Tool Source Code complies with tool architecture.") are fully satisfied by a twofold

tool source code review: (1) the hand-written OCaml code is reviewed directly against the informal TR (as no LLTR have been refined from the informal TR), and (2) the extracted OCaml code is reviewed against the formal LLTR.

Due to the nature of OCaml and the reverse-engineering of the informal TR from the hand-written OCaml code, no additional tool code standards are required. This constitutes an alleviation from TQL-3 and fully covers Objective T-5.4 ("Tool Source Code conforms to Tool Code Standards.").

Traceability of the tool source code to the low-level tool requirements, as required by Objective T-5.5, is demonstrated through direct traceability of the extracted OCaml code to the formal LLTR (verified by Reading), and through the reverse-engineering of the informal TR from the hand-written OCaml code.

Objective T-5.6 ("Source code is accurate and consistent.") is alleviated from TQL-3 for the formal parts based on the extent of testing activities performed during tool qualification and the availability of the formal proof. The hand-written code parts are additionally covered by the tool source code review.

Objective T-5.7 ("Output of tool integration process is complete and correct.") is addressed by an analysis of the tool build process. The build process fails if any non-suppressed warning or error is produced. Justifications are provided for each class of suppressed warnings, demonstrating that they do not compromise compliance with the low-level tool requirements. The linked executable does not use any external component not involved in the build process (e.g., it does not load dynamic libraries from the host system). Consequently, failures due to missing components or incorrect use of external APIs are excluded.

7.7. Table T-6: Testing of Outputs of Integration Process

The objectives T-6.1 ("Tool Executable Object Code complies with Tool Requirements."), T-6.2 ("Tool Executable Object Code is robust with Tool Requirements."), T-6.3 ("Tool Executable Object Code complies with low-level tool requirements."), and T-6.4 ("Tool Executable Object Code is robust with low-level tool requirements.") are fully addressed by the following activities:

Test procedures and test cases are developed and executed against the informal TR (TVCP).

There is no testing against informal LLTR, as no informal LLTR exist. This constitutes a TQL-3 alleviation. Testing against formal LLTR is replaced by the following argumentation chain: (1) the formal proof ensures compliance of the FLLTR with the FTR, (2) compliance of the extracted OCaml source code is ensured by reviewing it against the FLLTR, and (3) confidence in the OCaml compiler is provided by the tests performed on the CompCert executable. These are both the qualification tests and those tests performed by the tool user on the application compiled by CompCert.

7.8. Table T-7: Verification of Outputs of Tool Testing

The objectives T-7.1 ("Test procedures are correct."), T-7.2 ("Test results are correct and discrepancies explained."), T-7.3 ("Test coverage of Tool Requirements is achieved."), T-7.4 ("Test coverage of low-level tool requirements is achieved."), T-7.8 ("Analysis of requirements-based testing (structural coverage to the level of statement coverage) is achieved."), and T-7.9 ("Analysis of requirements-based testing (data coupling and control coupling) is achieved.") are fully addressed by checklist-based reviews of both the TOVVCP and the TVCP, by reviewing the corresponding test results for correctness and adequate discrepancy explanations, and by analyzing the trace data to demonstrate coverage.

Statement-level requirements-based coverage is achieved by reverse-engineering the informal TR from the hand-written OCaml source code. This ensures that each OCaml statement is mapped to an informal TR describing the exact intended behavior of that code fragment. In addition, test cases are developed for each informal TR. This process provides confidence that no unintended functionality exists.

We claim an alleviation for verification of outputs of tool testing on the formal part (extracted OCaml code) which is replaced by the argumentation chain provided in Section 7.7.

7.9. Table T-8: Tool Configuration Management Process

Table T-8 requires the "identification of configuration items" (Objective T-8.1) and the establishment of "baselines and traceability" (T-8.2), "problem reporting, change control, change review, and configuration status accounting" (T-8.3), "archive, retrieval, and release" (T-8.4), and "tool life cycle environment control" (T-8.5).

All objectives are fully addressed by the corresponding life cycle processes defined in the *Tool Configuration Management Plan*. This plan identifies all configuration items, specifies their identification scheme, describes the use of version control to track and record changes, and defines the processes for customer relationship management, tool problem reporting, change control and review. It also defines the procedures for creating tool releases as well as for archiving and retrieving them.

7.10. Table T-9: Tool Quality Assurance Process

Table T-9 contains objectives to obtain assurance that "tool plans and standards are developed and reviewed for consistency" (T-9.1), that "tool processes comply with approved plans" (T-9.2), that "tool processes comply with approved standards" (T-9.3), that "transition criteria for the tool life cycle processes are satisfied" (T-9.4), and that "tool conformity is conducted" (T-9.5).

The *Tool Quality Assurance Plan* (TQAP) defines quality objectives for the project plans, the top-level documents such as the TAS and TCI, and all other tool life cycle data. These objectives ensure compliance with the above DO-330 objectives. As required by DO-330, the quality assurance process is performed with independence to ensure the objectives are satisfied.

Compliance of both life cycle data and process executions with these quality objectives is verified through quality inspections. Each inspection reviews a defined subset of data with respect to a particular quality aspect. For example, a defined portion of the Tool Operational Requirements is checked against

the quality objectives for TORs. The subset size is specified by quality control rates; for the TOR inspection, 10% of all TORs must be examined.

The results of all inspections are recorded in the Tool Quality Assurance Records (TQAR). All inspections are conducted either as part of the tool planning review or the tool conformity review.

7.11. Table T-10: Tool Qualification Liaison Process

The objectives T-10.1 ("Communication and understanding between the applicant and the certification authority is established"), T-10.2 ("The means of compliance is proposed and agreement is obtained"), and T-10.3 ("Compliance substantiation is provided") require the tool user to create top-level plan documents such as the TQP, TAS, and TCI. These user documents are based on the corresponding developer variants, resulting in a partial contribution from the tool developer to these objectives.

Objective T-10.4 ("Impact of known problems on the Tool Operational Requirements is identified and analyzed") is fully addressed by the tool developer, as the TORs are created by the developer based on the user needs. Identified issues are handled according to defined processes, which include identification, impact and root cause analysis, implementation of workarounds, and creation of fix releases.

8. Contributions to the State of the Art

Previous work on compiler qualification is typically based on compiler test suites and/or fit-for-purpose certificates, where the compiler is considered as a black box on an existing test suite without insights into the compiler implementation. In contrast to that, the described qualification strategy shows a profound application of DO-330 objectives for the qualification of a compiler that makes use of formal methods for verifying the compilation step. The formal proof on the translation functions implies full data and control coverage, which is superior even to testing with full path coverage.

A DO-330 qualification requires to demonstrate coverage criteria both in the requirements-based verification of the tool source code as well as between the different requirement hierarchies. We present a rigorous breakdown from the supported input language (ISO C99) via the different levels of requirements (TOR, TR, LLTR) to the tool source code. The formal proof provides higher assurance than any test-based qualification campaign can reach. For the hand-written OCaml code, *each code line* is associated to an informal tool requirement and each tool requirement is verified by at least one test case. In addition to that, each tool operational requirement is verified by at least one test case. Each tool operational requirement is covered by at least one tool requirement.

The proposed qualification strategy based on the semantic preservation proof provides a far higher confidence in the correctness of the compilation step than traditional approaches. It also paves the path for significant improvements of application performance and development efficiency by allowing to use compiler optimizations without compromising source-to-target code traceability. It redefines the state of the art for compiler qualification that so far was purely test-based.

The methodology described here can be generalized to other (formal) compiler projects.

9. Conclusion

In this article, we presented our qualification strategy for applying CompCert to safety-critical avionics software in compliance with DO-178C, DO-333, and DO-330. The qualification kit realizes a profound application of DO-330 objectives for the qualification of a compiler that makes use of formal methods for verifying the compilation step. Our approach advances the state of the art in compiler qualification for avionics software at the highest assurance level under DO-178C, enabling significant gains in both application performance and development efficiency.

References

[1] Verasco – formal verification of static analyzers and compilers. https://verasco.imag.fr/wiki/Main_Page, n.d. Project funded by ANR, reference ANR-11-INSE-003.

[2] R. Bedin França, S. Blazy, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS 2012: Embedded Real Time Software and Systems*, 2012.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[4] E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *EMSOFT '08*, pages 255–264. ACM, 2008.

[5] R. B. França, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris. Towards Formally Verified Optimizing Compilation in Flight Control Software. In P. Lucas and R. Wilhelm, editors, *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *Open Access Series in Informatics (OASIcs)*, pages 59–68, Dagstuhl, Germany, 2011. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[6] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Prog. Lang. Syst.*, 18(3):300–324, 1996.

[7] IEC 60880. Nuclear power plants instrumentation and control systems important to safety software aspects for computer-based systems performing category a functions, 2006.

[8] ISO. International standard ISO/IEC 9899:1999, Programming languages – C, 1999.

[9] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *ESOP 2012: 21st European Symposium on Programming*, volume 7211 of *LNCS*, pages 397–416. Springer, 2012.

[10] D. Kästner, J. Barrho, U. Wünsche, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. Blazy. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In *ERTS2 2018 - Embedded Real Time Software and Systems*, Toulouse, France, Jan. 2018. 3AF, SEE, SIE. Archived in the HAL-INRIA open archive, https://hal.inria.fr/hal-01643290/file/ERTS_2018_paper_59.pdf.

[11] D. Kästner, X. Leroy, S. Blazy, B. Schommer, M. Schmidt, and C. Ferdinand. Closing the gap – the formally verified optimizing compiler CompCert. In *SSS'17: Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium*, pages 163–180. CreateSpace, 2017.

[12] D. Kästner, L. Mauborgne, S. Hahn, S. Wilhelm, J. Herter, C. Cullmann, and C. Ferdinand. Sound non-interference analysis for C/C++. In A. Ceccarelli, M. Trapp, A. Bondavalli, and F. Bitsch, editors, *Computer Safety, Reliability, and Security - 43rd International Conference, SAFECOMP 2024, Florence, Italy, September 18-20, 2024, Proceedings*, volume 14988. Springer, 2024.

[13] D. Kästner, M. Pister, and C. Ferdinand. Obtaining DO-178C Certification Credits by Static Program Analysis. *Submitted to the 11th European Congress on Embedded Real Time Software and Systems (ERTS 2022)*, March 2022.

[14] D. Kästner, S. Wilhelm, C. Mallon, S. Schank, C. Ferdinand, and L. Mauborgne. Automatic Sound Static Analysis for Integration Verification of AUTOSAR Software. In *WCX SAE World Congress Experience*. SAE International, Apr. 2023.

[15] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, Jan. 2016. SEE.

[16] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, Jan. 2016.

[17] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 83–94, New York, NY, USA, 2000. Association for Computing Machinery.

[18] NULLSTONE Corporation. NULLSTONE for C. `http://www.nullstone.com/htmls/ns-c.htm`, 2007.

[19] Radio Technical Commission for Aeronautics. RTCA DO-178C. Software Considerations in Airborne Systems and Equipment Certification, 2011.

[20] Radio Technical Commission for Aeronautics. RTCA DO-330. Software Tool Qualification Considerations, 2011.

[21] Radio Technical Commission for Aeronautics. RTCA DO-333. Formal Methods Supplement to DO-178C and DO-278A, 2011.

[22] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI '11*, pages 283–294. ACM, 2011.