Towards Optimizing Certified Compilation in Flight Control Software*

Ricardo Bedin França[1,2], Denis Favre-Felix[1], Xavier Leroy[3], Marc Pantel[2], and Jean Souyris[1]

1 **AIRBUS Operations SAS**
316 Route de Bayonne, Toulouse, France
`{ricardo.bedin-franca,denis.favre-felix,jean.souyris}@airbus.com`

2 **Institut de Recherche en Informatique de Toulouse**
2 Rue Charles Camichel, Toulouse, France
`{ricardo.bedinfranca, marc.pantel}@enseeiht.fr`

3 **INRIA Rocquencourt**
Domaine de Voluceau, Le Chesnay, France
`xavier.leroy@inria.fr`

Abstract

This work presents an evaluation of the CompCert formally-proved compiler for level A critical flight control software. First, the motivation for choosing CompCert is presented, as well as the requirements and constraints of safety-critical avionics software. The evaluation of its performance (measured in WCET) is presented and the results are compared to those obtained with the currently used compiler. Finally, the paper discusses verification and certification issues that are raised when one seeks to use CompCert to compile such critical software.

Keywords and phrases Certified compilation, avionics software, WCET, code optimization

Digital Object Identifier 10.4230/OASIcs.xxx.yyy.p

1 Introduction

As "Fly-By-Wire" controls have become standard in the aircraft industry, embedded software have been extensively used to improve planes' controls while simplifying pilots' tasks. Since these controls have a crucial role in flight safety, flight control software must comply with very stringent regulations. In particular, any flight control software (regardless of manufacturer) must follow the DO-178 [1] guidelines for level-A critical software: when such software fail, the flight as a whole (aircraft, passengers and crew) is at risk.

The DO-178 advocates solid development and certification processes for avionics software, with specification, design, coding, integration and verification being thoroughly planned, executed, reviewed and documented. It also enforces the traceability among all development phases and the generation of coherent, verifiable software. Verification and tooling aspects are also dealt with: the goals and required verification levels are explained, and there are guidelines for the use of tools that automate developers' tasks.

In addition to the DO178 regulations, each airplane manufacturer usually has its own internal constraints: available hardware, delivery schedule, additional safety constraints, etc. On the other hand, as software tend to get larger and more complex, there is a permanent desire to use optimally the available hardware. Such a need is not necessarily in line with the

* This work was partially supported by ANR grant Arpège U3CAT.

aforementioned constraints: indeed, meeting them both is usually very challenging because performance and safety may be contradictory goals.

This paper describes the activities and challenges in an Airbus experiment that ultimately seeks to improve the performance of flight control softwares without reducing the level of confidence obtained by the development and verification strategy currently used. This experiment is carried around a very sensitive step in software development: the compilation. A compiler may be influential in a software performance, as advanced compilers are able to generate optimized assembly code and such optimizations may be welcome if, for some reason, the source code is not itself optimal – in high level programming languages, the source code is unlikely to be optimal. This work presents the performance-related analyses that were carried out to assess the interest of using an optimizing, formally-proved compiler, as well as the first ideas to make it suitable for application in certifiable software development.

The paper is structured as follows: Section 2 presents the fundamentals and challenges in the development of flight control software, and describes the methods used in this work to evaluate software performance, as well as the elements that weigh most in this aspect. Section 3 presents the CompCert compiler, the results of its performance evaluation and some ideas to use it confidently in such critical software. Section 4 draws conclusions from the current state of this work.

2 Flight Control Software and Performance Issues

2.1 An Overview of Flight Control Software

Since the introduction of the A320, Airbus relies on digital electrical flight control systems ("fly-by-wire") in its aircraft [2]. While older airplanes had only mechanical, direct links between the pilots' inputs and their actuators, modern aircraft rely on computers and electric connections to transmit these inputs. The flight control computers contain software that implement the necessary flight control laws, thus easing pilots' tasks – for example, a "flight envelope protection" is implemented not to let aircraft attain combinations of conditions (such as speed and G-load [2]) that are out of their specified physical limits and could cause failures.

It is clear that the dependability of such a system is coupled with the dependability of its software, and the high criticality of a flight control system implies an equally high criticality of its software. As a result, flight control software are subject to the strictest recommendations of the DO-178 standard (whose current version is B): in addition to very rigorous planning, development and verification, there are "independence" guidelines (the verification shall not be done by the coding team) and every automated tool used in the software development process is also subject to verification whenever it is used. These systematic tool output verification activities can be skipped if the tool is "qualified" to be used in a given software project. Tool qualification follows an approach similar to the certification of a flight software itself, as its main goal is to show that the tool is properly developed and verified, thus being considered as adequate for the whole software certification process. The DO-178B makes a distinction between development and verification tools; development tools are those which may directly introduce errors in a program – such as a code generator, or a compiler – whereas verification tools do not have direct interference over the program, although their failure may also cause problems such as incorrect assumptions about the program behavior. The qualification of a development tool is much more laborious and requires a level of planning, documentation, development and verification that can be compared to the flight control software itself.

The software and hardware used in this work are similar to those described in [8]. The application is specified in the graphical formalism SCADE, which is then translated to C code by a qualified automatic code generator. The C code is finally compiled and linked to produce an executable file. The relevant hardware in the scope of this work currently comprises the microprocessor (MPC755), its L1 cache memory and an external RAM memory. The MPC755 is a single-core, superscalar, pipelined microprocessor, which is much less complex than modern multi-core processors but contains enough resources not to have an easily predictable time behavior.

In order to meet DO-178B guidelines, many verification activities are carried out through the development phases. While the code generator itself (developed internally) is qualified as a development tool, the compiler[1] is purchased and its inner details are not mastered by the development team. As a result, its qualification is not viable and its output must be verified. However, verifying the whole compiled code would be prohibitively expensive and slow. Since the code is basically composed of many instances of a limited set of "symbols", such as math operations, filters and delays, the simplest solution is to make the compiler generate constant code patterns for each symbol. This can be achieved by hindering code generator and compiler optimizations, and the code verification may be accomplished by verifying the (not very numerous) expected code patterns for each symbol with the coverage level required by the DO-178B, and making sure every compiled symbol has an expected pattern. Other activities (usually test-based) are also carried out to ensure code integration and functional correctness.

2.2 Estimating Software Performance

Given the critical, real-time nature of flight control software, the currently safest mechanism to carry out a schedulability analysis is worst-case execution time (WCET) analysis: it must be considered that the most pessimistic possible execution is really feasible and that, even in this case, the software computation time is adequate. Calculating software WCET is by no means simple: every hardware detail counts and complex hardware and software yield potentially huge analysis state spaces, making the search for an exact WCET nearly impossible. Thus, the main goal is usually finding a time which is proved higher than the actual WCET, but not much higher, in order to minimize resource waste - for software verification and certification means, the estimated/computed WCET must be interpreted as the actual one.

As explained by Souyris *et al* in [8], the "old" method of calculating the WCET of Airbus's automatically generated flight control software was essentially summing the execution times of small code snippets in their worst-case scenarios. The proofs that the estimated WCET was always higher than the actual one did not need to be formal, thanks to the simplicity of the processor and memory components available at that time - careful reviews were proved sufficient to ensure the accuracy of the estimations. On the other hand, currently available components are much more complex. Modern microprocessors have several resources – such as cache memories, superscalar pipelines, branch prediction and instruction reordering – that accelerate their average performance but make their behavior much more complicated to analyze.

While a WCET estimation that does not take these resources into account would make no sense, it is not feasible to make manual estimations of a program with such complexity. The

[1] For confidentiality reasons, the currently used compiler, linker and loader names are omitted.

current approach at Airbus [8] relies on AbsInt[2]'s automated tool a^3 [4] (which had to be qualified as a verification tool, according to the DO-178B) to compute the WCET via static code analysis. In order to obtain accurate results, the tool requires a precise model of the microprocessor and other influent components; this model was created during a cooperation between Airbus and AbsInt. In addition, sometimes it is useful (or even essential) to give a^3 some extra information about loops or register values to refine its analysis. Examples of these "hints", which are provided in annotation files, are shown in [8]. As the code is generated automatically, an automatic annotation generator was devised to avoid manual activities and keep the efficiency of the development process. In order to minimize this need for code annotations, and to increase overall code safety, the symbol library was developed so as to be as deterministic as possible.

2.3 Searching for performance gains

In a process with so many constraints of variable nature, it is far from obvious to find feasible ways to generate "faster" software: the impact of every improvement attempt must be carefully evaluated in the process as a whole - a slight change in the way of specifying the software may have unforeseen consequences not only in the code, but even in the highest-level verification activities. It is useful to look at the V development cycle (which is advocated by the DO-178B) so as to find what phases may have the most promising improvements:

- Specification: Normally, the specification team is a customer of the development team. Specification improvements may be discussed between the two parts, but they are not directly modifiable by the developers.
- Design: In an automatic code generation process, the design phase is put together with the specification.
- Coding: The coding phase is clearly important for the software performance. In the pattern coding level, there are usually few improvements to be made: after years of using and improving a pattern library, finding even more optimizations is difficult and time-consuming. However, the code generators and the compilers may be improved by relaxing this pattern-based approach in the final library code.
- Verification: In the long run, one must keep an eye on the new verification techniques that arise, because every performance gain is visible only if the WCET estimation methods are accurate to take them into account. Also, other verification techniques (unit verification, code generator/compiler verification) must be constantly evaluated because sub-optimal specification and coding choices might have been made due to a lack of strong verification techniques at one time.

This work presents the current state of the experiments that are being performed in order to improve the compilation process.

3 A new approach for compiler verification

3.1 Certification constraints for a compiler

In order to have a better idea of what can be considered as feasible when dealing with compilers in such critical software, it is useful to keep the DO-178B considerations in mind. It contains some very important recommendations:

[2] www.absint.com

- Compiler optimizations do not need to be verified if the software verification provides enough coverage for the given criticality level.
- Object code that is not directly traceable to source code must be detected and verified with adequate coverage.
- The Software Verification Plan must contain all assumptions about the correctness of the compiler, linker and loader.

Thus, an optimizing compiler must be qualified, or additional verification activities must be carried out to ensure traceability and compliance of the object code.

Section 2.1 states that the trust in a development process that includes a "black-box" compiler is achieved by banning all compiler optimizations in order to have a simple structural traceability between source and binary code patterns. In addition to traceability itself, other constraints are met with this approach:

- Unit verification can be accomplished by means of symbol library unit test with MCDC structural coverage, as there is a limited number of code patterns that can be generated for each symbol instance. The coverage of the whole automatically-generated code is ensured, as it is a concatenation of such tested patterns.
- With predictable code patterns, it is possible to know exactly what assembly code lines of the automatically-generated code require annotations to be correctly analyzed by a[3]. Since there are relatively few library symbols that require annotations, and they have just a few possible patterns, it is not difficult to find out where the annotations should be placed.
- Compiler analyses can be done automatically, as its correctness is established by a simple code inspection: every generated pattern for a given symbol must match one of the unit-tested patterns for the same symbol. Compiler, assembler and linker are also tested during the integration tests: as the object code is executed on the actual target computer, the DO-178B code compliance requirements would not be fulfilled if there were wrong code or mapping directives.

Thus, several objectives are accomplished with a non-optimized code, and a different approach would lead to many verification challenges. COTS compilers usually do not provide enough information to ensure their correctness, especially when taking optimizations into account. If developers could actually master a compiler behavior, the DO-178B tool qualification might give way to a more flexible (albeit laborious) way of compiling.

3.2 CompCert: Towards a trusted compiler

One can figure out that traditional COTS (Commercial off-the-shelf) compilers are not oriented to the rigorous development of flight control softwares – the notion of "validated by experience" tools is not acceptable for highly critical software development tools. However, there have been some advances in the development of compilers, with interesting works that discuss the use of formal methods to implement "correct" compilers, either by verifying the results of their compilation [6] or by verifying the compiler semantics [10, 5]. In the scope of this work, the most promising development is the CompCert[3] compiler. Its proved subset is broader in comparison to other experimental compilers, it compiles most of the C language (which is extensively used in embedded systems), and it can generate Assembly code for the MPC755.

[3] `http://compcert.inria.fr`

As explained in [5], CompCert is mostly programmed and proved in Coq, using multiple passes to perform an optimized compilation. Its optimizations are not very aggressive, though: as the compiler's main purpose is to be "trustworthy", it carries out basic optimizations such as constant propagation, common subexpression elimination and register allocation by graph coloring, but no loop optimizations, for instance. As no code optimizations are enabled in the currently used compiler, using a few essential optimization options could already give good performance results. In addition, less agressive optimization helps to preserve the source code control flow.

3.3 Performance evaluation of CompCert

In order to carry out a meaningful performance evaluation, the compiler was tested on a prototype as close as possible to an actual flight control software. As this prototype has its own particularities with relation to compiler and mapping directives, some adaptations were necessary in both the compiler and the code. To expedite this evaluation, CompCert was used only to generate assembly code for the application, while the "operational system" was compiled with the default compiler. Assembling and linking were also performed with the default tools, for the same reason. Figure 1 illustrates the software development chain and the tools used for each step.

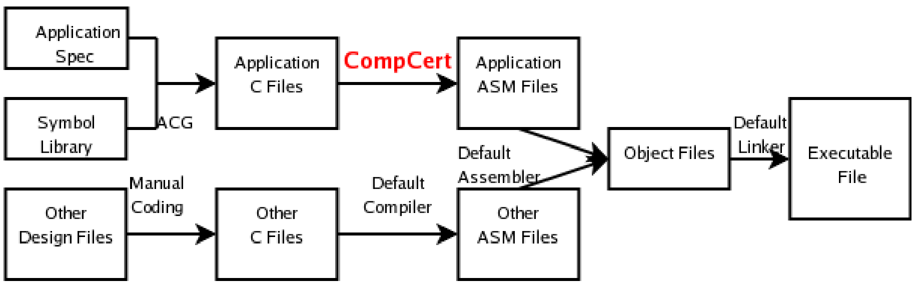

Figure 1 The development chain of the analyzed program

About 2500 files (2.6MB of assembly code with the currently used compiler) were compiled with CompCert (version 1.7.1-dev1336) and with three configurations of the default compiler: non-optimized, optimized without register allocation optimizations, and fully optimized. A quick glance at some CompCert generated code was sufficient to notice interesting changes: the total code size is about 26% smaller than the code generated by the default compiler. This significant improvement has its roots in the specification formalism itself: a potentially long sequential code is composed by a sequence of mostly small symbols, each one with its own inputs and outputs. Thus, a non-optimizing compiler must do all the theoretically needed load and store operations for each symbol and, even if those inputs and outputs are stored in cache, the program execution is slower than if they were kept inside the processor registers. In fact, the register allocation has to be done manually for the non-optimized code and CompCert manages to generate more compact Assembly code by ignoring the user-defined register allocation. Listing 1 depicts a non-optimized simple symbol that computes the sum of two floating-point numbers. As this symbol is often in a sequence with other symbols, it is likely that its inputs were computed just before and its output will be used in one of the next scheduled instructions. If there are enough free registers, CompCert will simply keep these variables inside registers and only the `fadd` instruction will remain, as shown in Listing 2.

Listing 1 Example of a symbol code

```
lfd f3, 8(r1)
lfd f4, 16(r1)
fadd f5, f4, f3
stfd f5, 24(r1)
```

Listing 2 Its optimized version

```
fadd f5, f4, f3
```

As the local variables are usually kept in stack, closer analysis showed that CompCert generates code with about 76% fewer cache reads and 65% fewer writes. Table 1 compares these results with those of the default compiler in optimized configurations, with the default non-optimized code as the reference.

	Code Size	Cache Reads	Cache Writes
CompCert	-25.7%	-76.4%	-65.1%
Default (optimized without register allocation)	+0.8%	+19.9%	+23.4%
Default (fully optimized)	-38.2%	-81.8%	-76.6%

Table 1 Code size and memory access comparison

In order to see the effects of this code size reduction, a^3 was used to compute the WCET for all analyzed nodes. The results are encouraging: the WCET of the CompCert compiled code was 12.0% lower than the reference. Without register allocation, the default compiler presented a reduction of only 0.5% in WCET, while there was a reduction of 18.4% in the WCET of the fully optimized code. The WCET comparison for each of the analyzed nodes is depicted in Figure 2. The WCET improvement is not constant over all nodes: some of them do not have many instructions, but they do have strong performance "bottlenecks" such as hardware signal acquisitions, which take considerable amounts of time and are not improved by code optimization. In addition, CompCert's recent support for small data areas was not used in the evaluation, while it is used by the default compiler. Nonetheless, the overall WCET is clearly lower.

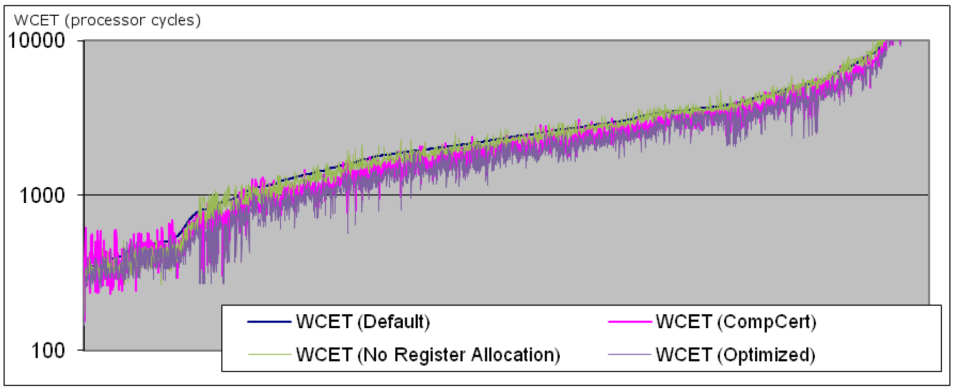

Figure 2 WCET for all analyzed program nodes

The results of these WCET analyses lead to two main conclusions:

- In the analyzed code, which represents well the kind of code found in flight control software, little improvements can be found without enabling the compiler to do an optimized register allocation.

- Even though CompCert does not perform many optimizations, the performance of its code is not too far from that of a COTS optimizing compiler, and significantly better than a non-optimizing one. This also emphasizes that an optimized register allocation is very important for the performance of automatically-generated, symbol-based code.

3.4 Generating annotations for WCET analysis

As mentioned in Section 2.2, annotations over automatically-generated code are necessary to increase precision of the WCET analysis whenever an accessed memory address or a loop guard depends on the value of a floating-point variable, or a static variable that is not updated inside the analyzed code. We have prototyped a minor extension to the CompCert compiler that supports writing annotations in C code, transmitting them along the compilation process, and communicating them to the a^3 analyzer. The input language of CompCert is extended with the following special form:

```
__builtin_annotation("0 <= %1 <= %2 < 360", i, j);
```

which looks like a function call taking a string literal as first argument and zero, one or several C variables as extra arguments. Semantically and throughout the compiler, this special form is treated as a *pro forma* effect, as if it were to print out the string and the values of its arguments when executed. CompCert's proof of semantic preservation therefore guarantees that control flows through these annotation statements at exactly the same times in the source and compiled code, and that the variable arguments have exactly the same numerical values in both codes. At the very end of the compilation process, when assembly code is printed, no machine instructions are generated for annotation statements. Instead, a special comment is emitted in the assembly output, consisting of the string argument (`"0 <= %1 <= %2 < 360"` in the example above) where the `%`i tokens are substituted by the final location (machine register, stack slot or global symbol) of the i-th variable argument. For instance, we would obtain

```
# annotation: 0 <= r3 <= @32 < 360
```

if the compiler assigned register `r3` to variable `i` and the stack location at stack pointer plus 32 bytes to variable `j`. The listing generated by the assembler then shows this comment and the program counter (relative to the enclosing function) where it occurs. From this information, a suitable annotation file can be automatically generated for use by the a^3 analyzer.

Several variants on this transmission scheme can be considered, and the details are not yet worked out nor experimentally evaluated. Nonetheless, we believe that this general approach of annotating C code and compiling these annotations as *pro forma* effects is a good starting point for the automatic generation of annotations usable during WCET analysis.

3.5 CompCert and the avionics software context

After the successful performance evaluation, it is time to study more thoroughly the feasibility of using CompCert in an actual flight control software development. Given all the constraints and regulations explained in this paper, this task is very long, as all constraints from several actors (customers, development, verification, certification) must be taken into account.

When an automatic code generator is used, it is clear that the customers want a highly reactive development team. A million-line program (with a great deal of its code being generated automatically) must be coded and verified in a few days; with such a strict schedule, little or no manual activities are allowed.

The development team also has its rules, in order to enforce correct methods and increase development safety. Thus, the compiler must generate a code that complies to an application binary interface (in this case, the PowerPC EABI) and other standards, such as IEEE754 for floating-point operations. Although this work used two compilers to build the whole software, CompCert will have to deal with all the program parts (the ACG-generated code is much bigger, but also simpler than the rest); it will also have to do assembling and linking.

The verification phase will be significantly impacted, given all the assumptions that were based on a code with predictable patterns:

Unit verification The unit verification of each library symbol will have to be adapted. With no constant code patterns, there is no way to attain the desired structural coverage by testing a number of code patterns beforehand. It would be too onerous to test the whole code after every compilation. A possible solution is to separate the verification activities of the source and object code. The verification of the source code can be done using formal methods, using tools that are already familiar inside Airbus, such as Caveat[4] and Frama-C [5] [9]. Object code compliance and traceability can be accomplished using the formal proofs of the compiler itself, as they intend to ensure a correct object code generation. In this case, only one object code pattern needs to be verified (e.g. by unit testing) for each library symbol and the test results can be generalized for all other patterns, thanks to the CompCert correctness proofs.

WCET computation A new automatic annotation generator will have to be developed, as the current one relies on constant code patterns to annotate the code. The new generator will rely on information provided directly by CompCert (Section 3.4) to correctly annotate the code when needed.

Compiler verification It is clear that the CompCert formal proofs shall form the backbone of a new verification strategy. An important point of discussion is how these proofs can be used in an avionics software certification process. The most direct approach is qualifying the compiler itself as a development tool, but it is far from a trivial process: the qualification of a development tool is very arduous, and qualifying a compiler is a new approach that will require intensive efforts to earn the trust of certification authorities. Thus, CompCert has to meet DO-178B level A standards for planning, development, verification and documentation, and these standards largely surpass the usual level of safety achieved by traditional compiler development processes. An alternative method of verification, which is also being discussed, is using its correctness proofs in complementary (and automatic) analyses that will not go in the direction of qualifying CompCert as a whole, but should be sufficiently well-thought-out to prove that it did a correct compilation.

4 Conclusions and Future Work

This paper described a direction to improve performance for flight control software, given their large number of development and certification constraints. The motivation for using a formally proved compiler is straightforward: certifying a COTS compiler to operate without restrictions (such as hindering every possible code optimization) would be extremely hard,

[4] `http://www-list.cea.fr/labos/gb/LSL/caveat/index.html`
[5] `http://frama-c.com`

if not impossible, as information related to its development are not available. While the largest part of the work – the development of an appropriate development and verification strategy to work with CompCert – has just started, the performance results are rather promising. It became clear that the “symbol library” automatic code generation strategy implies an overhead in load and store operations, and a good register allocation can mitigate this overhead.

Future work with CompCert include its adaptation to the whole flight control software and the completion of the automated mechanism to provide useful information that can help in the generation of code annotations. Also, discussions among development, verification and certification teams in Airbus are taking place to study the needed modifications throughout the development process in order to use CompCert in a development cycle at least as safe as the current one. Parallel studies are being carried out to find new alternatives for software verification, such as Astrée [3], and evaluate their application in the current development cycle [9].

In addition, the search for improvements in flight control software performance is not limited to the compilation phase. The qualified code generator is also subject to many constraints that limit its ability to generate efficient code. Airbus is already carrying out experiments in order to study new alternatives, such as the Gene-Auto project [7].

References

1 DO-178B: Software Considerations in Airborne Systems and Equipment Certification, 1982.

2 Dominique Brière and Pascal Traverse. AIRBUS A320/A330/A340 Electrical Flight Controls: A Family of Fault-Tolerant Systems. In *FTCS*, pages 616–623, 1993.

3 Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of Abstractions in the ASTRÉE Static Analyzer. In Mitsu Okada and Ichiro Satoh, editors, *ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006.

4 Reinhold Heckmann and Christian Ferdinand. Worst-case Execution Time Prediction by Static Program Analysis. In *In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004*, pages 26–30. IEEE Computer Society, 2004.

5 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

6 George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. *SIGPLAN Not.*, 33(5):333–344, 1998.

7 Ana-Elena Rugina and Jean-Charles Dalbin. Experiences with the Gene-Auto Code Generator in the Aerospace Industry. In *Proceedings of the Embedded Real Time Software and Systems (ERTS^2)*, 2010.

8 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 *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

9 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.

10 Martin Strecker. Formal Verification of a Java Compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2002.