DO-178C Compliance of Verisoft Formal Methods

Blasum Holger, Frank Dordowsky, Bruno Langenstein, Andreas Nonnengart

DO-178C Compliance of Verisoft Formal Methods*

Holger Blasum† Frank Dordowsky‡ Bruno Langenstein§
Andreas Nonnengart§

December 3, 2011

Verisoft XT was a three-year research project funded by the German Federal Ministry of Education and Research (BMBF). The main goal of the project was the pervasive formal verification of computer systems. One of its sub-projects examined the application of formal methods in the avionics domain. Today's avionics software should be developed in accordance with the RTCA/EUROCAE standard DO-178B/ED-12B to achieve formal acceptance by certification authorities. This standard lists formal methods merely as alternative means but does not provide guidance on the use and acceptance of formal methods. Its successor DO-178C/ED-12C will provide this guidance in its Formal Methods Supplement. Although DO-178C was not published during project runtime, the available material nevertheless allowed us to examine the compliance of two of the formal methods and tools – VSE and VCC – that have been used in Verisoft XT. This paper summarises the results of this evaluation and thus may serve as a first step in the certification planning of a real avionics project that would use either one or both methods.

Keywords:
Avionics Software, DO-178C, Formal Methods, VCC, VSE

1 Introduction

Safety critical avionics systems are a natural candidate for the application of formal methods. Hence avionics software was one of the application scenarios within the Verisoft XT project [22], a three-year research project on the pervasive formal verification of computer systems funded by the German Federal Ministry of Education and Research (BMBF).

Avionics software must be formally accepted by a certification authority such as the FAA in the U.S. or EASA in Europe. Certification authorities and industry have agreed on the DO-178B set of guidelines [20] as a basis for software assurance in civil aviation, but this standard is also gradually adopted by military certification authorities. In DO-178B, the main verification method is testing, and formal methods are only mentioned as alternative means. DO-178B provides only little guidance on the application of formal methods and suggests to apply formal methods in areas that are difficult to test exhaustively, such as concurrency, distributed processing, redundancy management, or synchronisation.

In 2005, a joint RTCA and EUROCAE working group was charged to update the standard to consider the software engineering techniques that have evolved since the publication of DO-178B in 1992. The new issue of the standard, DO-178C/ED-12C, will contain a *Formal Methods Supplement* (FMS) that defines how formal methods can be used within a certification project.

One of the work packages of the Verisoft XT avionics subproject was to establish DO-178B conformant development and verification processes that are supported by formal methods. Instead of devising a process on its own, the Verisoft project team decided to examine the available material of the emerging DO-178C FMS. This was done for two of the Verisoft XT formal methods dominant in the avionics subproject: the Verification Support Environment (VSE, [1]) and the Verifying C Compiler (VCC, [8]).

In section 2 we provide a short summary of the FMS as available during run-time of the Verisoft XT project [5, 6]. Sections 3 and 4 introduce VSE and VCC and show how they could be used within an avionics project. For each method we identify the life cycle artefacts expressible by the method's notation itself. This determines the *scope* of the formal method, i.e. the development and verification processes where it can be applied. For our analysis, we consider the maximum scope of the methods. The processes that are outside of the scope of the examined formal method will not be considered. We also map the abstract concepts of the standard to concrete elements of both methods which constitutes our interpretation of the standard and determines the objectives that are within the scope of VSE and VCC.

In section 5, we summarise our discussion to which extent each objective is fulfilled. Finally, we discuss the issue of tool qualification because the achievement of some objectives depends on it, and conclude with a summary of our results and experience made with

*Work partially funded by the German Federal Ministry of Education and Research (BMBF), grant 01 IS 07 008.
†SYSGO AG
‡ESG Elektroniksystem- und Logistik GmbH
§German Research Centre for Artificial Intelligence (DFKI)

the adoption of the DO-178C.

This paper does not attempt to compare VSE with VCC nor both methods with other existing formal methods. Instead, it summarises the arguments that must be provided to the certification authority when proposing the application of either method in a real project. Moreover, the paper also identifies the shortcomings of both methods with respect to the DO-178C objectives – these non-compliances must be resolved by other means.

2 DO-178C and the Formal Methods Supplement

2.1 DO-178B

DO-178B lists a total of 20 life cycle data items – artefacts created during the software life cycle. Of these 20, only the following are relevant to our discussion: (1) *High-Level Requirements* (HLR), created in the software requirements process, (2) *Low-Level Requirements* (LLR) and (3) *Software Architecture*, created in the software design process, (4) *Source Code*, developed in the software coding process, and finally (5) *Executable Object Code* that is the output of the software integration process.

These life cycle data are shown as hexagons in Fig. 1. It also shows system requirements as an input into the software development process.

DO-178B lists a set of objectives that must be achieved by an avionics software project where the number of objectives that are applicable to the project depends on the criticality level of the application. The objectives are related to existence and properties of the life cycle data (e.g. accuracy, consistency, verifiability) or to the relation between life cycle data (compliance, traceability). These objectives have been taken over from DO-178B into the core section of DO-178C and will therefore be referred to as *core objectives* in the following.

Moreover, Fig. 1 shows all *activities* that are relevant to the examination of the formal methods as arrows and *objectives* as labels to those arrows. According to the introduction of the DO-178C core document, activities provide a means for satisfying objectives.

2.2 DO-178C and FMS

The DO-178C *Formal Methods Supplement* (FMS) augments the objectives established in the core document. It introduces general concepts to accommodate a wide variety of different formal methods. A *formal method* is a *formal analysis* carried out on a *formal model*. A method is accepted as formal if it has an unambiguous, mathematically defined, syntax and semantics [5]. The formal model formulates *properties* consisting of certain software life cycle data, such as requirements.

A *formal analysis case* is the analogon to a test case and is a combination of the property to be analysed or proved along with all assumptions relied upon for the formal analysis. Sometimes *assumptions* must be made on the software or its environment in order to successfully complete the formal analysis.

A *formal analysis procedure* is the process of executing a formal analysis case to determine the results of the formal analysis and to compare these results against the expected results. The *formal analysis result* is the result of an execution of a formal analysis procedure.

The FMS provides guidelines on how formal methods can be used for the verification of the objectives shown in Fig. 1. In addition to the core document, it defines objectives that concern the usage of formal methods:

- *Correctness of Formal Analysis Cases and Procedures:* The formal analysis covers the objectives shown in Fig. 1. Moreover, any assumptions that were included in the formal analysis should be justified and any false assumptions (which would invalidate the analysis) should be identified and removed.

- *Correctness of Formal Analysis Results:* The formal analysis results are correct and discrepancies between actual and expected results are explained.

- *Correctness of Requirement Formalisation:* In case a natural language requirement is translated into a formal notation it must be demonstrated that the formalisation is a conservative representation of the informal requirement.

- *Formal Method Soundness:* A *sound* method never asserts that a property is true when it may not be true. All notations used for formal analysis shall have a precise, unambiguous, mathematically defined syntax and semantics.

DO-178B also introduces a set of objectives that consider the verification of the outputs of the verification process. These objectives include the test coverage of the requirements (*requirements coverage*) as well as the test coverage of the software structure (*structural coverage*). The FMS attempts to transfer these criteria to formal methods [5]:

- *Requirements Coverage:* Assurance that there is at least one formal analysis case for every requirement, and that all assumptions about the software and its environment have been justified and verified.

- *Complete Coverage of Each Requirement:* Demonstration that all possible paths through the code with all possible data values have been considered, and that all assumptions have been made explicit and have been verified.

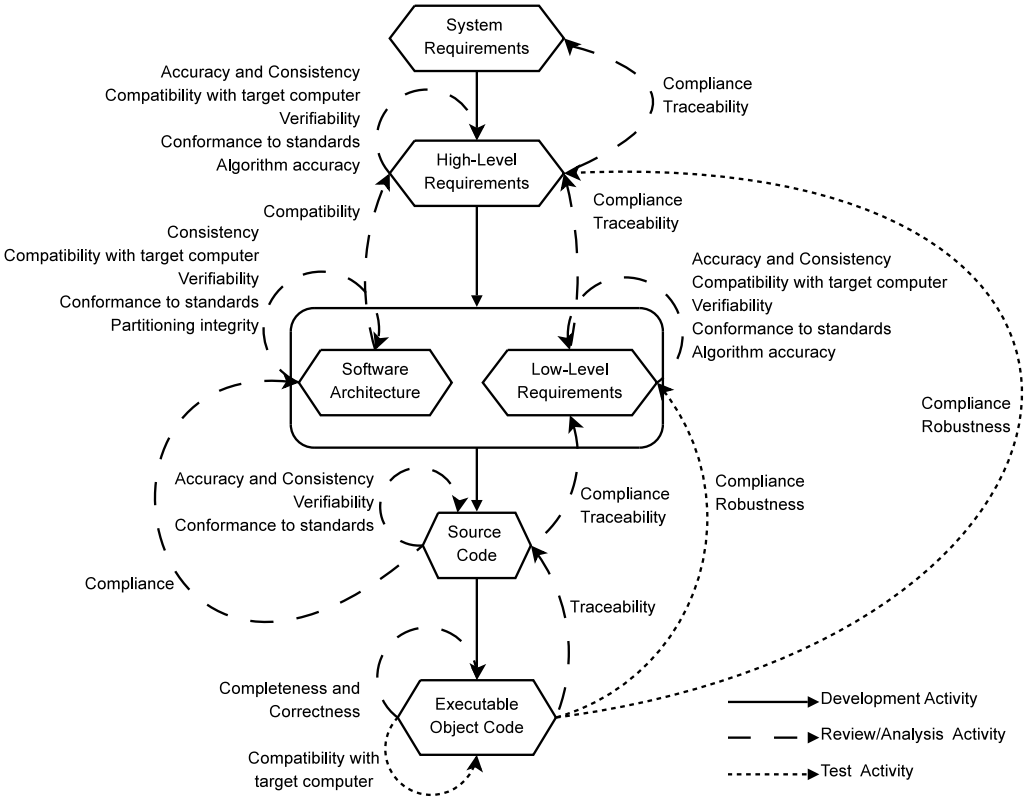

Figure 1: Development Artefacts, Verification Activities and Objectives, taken from [5]

- *Completeness of the Set of Requirements:* Assurance that the set of requirements is complete with respect to the intended functions, i.e. that the output is specified for all possible input conditions and that required input conditions are specified for all possible outputs.
- *Unintended Dataflow Detection:* Demonstration that all dependencies between inputs and outputs in the source code comply with the requirements.
- *Dead Code and Deactivated Code Detection:* Dead code is executable object code which cannot be executed in an operational configuration of the target computer and which is not traceable to a system or software requirement [20]. Dead code should be identified by review or analysis and removed.

  For deactivated code, i.e. code that is only executed in certain configurations of the target computer environment, the operational configuration needed for normal execution of this code should be established and additional verification cases and procedures must be developed to show that this code cannot be inadvertently executed.

3 DO-178C Scope of VSE

The *V*erification *S*upport *E*nvironment (VSE) is a tool that supports the formal development of complex large scale software systems from abstract high level specifications down to the code level. It provides both an administration system to manage structured formal specifications and a deductive component to maintain correctness on the various abstraction levels (see Fig. 2). Taken together these components guarantee the overall correctness of the complete development. VSE has been developed on behalf of the German Federal Office of Information Security (BSI) to satisfy the needs in software developments according to the standards ITSEC and Common Criteria. Deployments of VSE span several industrial and research projects, among others the control system of a heavy robot facility, the control system of a storm surge barrier, a formal security policy model conforming to the German signature law and protocols for chip card based biometric identification [1, 7, 14, 16].

Method. VSE supports development of a structured formal specification, which is organised around a *development graph* consisting of development objects (elementary specifications like theories, modules etc.) and links between them. Specifications are either in the style of *Abstract Data Types* (ADT) using predicate logic or *State Based Systems* using *Temporal Logic of Actions* (TLA).

User Interaction. Large specifications are broken down into smaller parts. The means of structuring specifications (visualised by links in the graph) are the following: For sequential systems ADTs can be *parametrised*, *instantiated* and *enriched*. State based specifications can be *combined* to describe concurrent

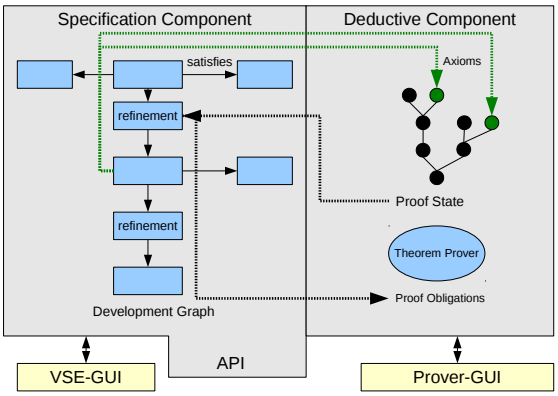

Figure 2: Architecture of VSE

systems consisting of components described as separate specifications.

Furthermore, development objects can be added to introduce further properties (*requirements*) of another development object block. A *satisfies link* between both objects indicates the relationship and is associated with a *deduction unit*. Logically the requirements have to be derivable from the system model. A deduction unit hosts the proof obligations that are sufficient to show the claim made by the link, and a context with the theories containing the axioms and lemmas that can be used in the proof.

A typical development process starts with a (structured) formal description of the *system model* on a high abstraction level. In a *refinement process* the abstract system model can be related to more concrete models. This is in correspondence with a software development that starts from a high level design and then descends to the lower software layers such that in a sense higher layers are implemented based on lower layers. Each such step can be reflected by a *refinement step* in VSE. These steps involve programming notions in the form of *abstract implementations*, that can later be exploited to generate source code. Each refinement step gives rise to proof obligations showing the correctness of the implementations. VSE maintains a deduction unit for each refinement step. Refinements also can be used to prove consistency of specifications, because they describe a way how to construct a model.

VSE includes a code generator that can produce C code from the abstract programs in a refinement or state based specification. This component of VSE can be seen as a translation process similar to what is done in conventional programming language tools. Therefore, application of this component within a DO-178C project should also follow the guidance offered by the "Model Based Design Supplement". However, for simplicity, we have not considered that supplement – where source code generation is used this will be stated clearly.

VSE in FMS Terms. Fig. 3 shows the life cycle data that can be expressed with VSE notation in grey coloured hexagons. VSE can be used to even capture system requirements (for an example see [14]). *System Requirements*, *High-Level* and *Low-Level Requirements* as well as the *Software Architecture* can be described as ADT or TLA specifications. From the abstract programs VSE can produce source code.

- A *property* can be any statement that can be expressed in first order logic or temporal logic. The properties of a system can be collected in a development object and connected to the system specification with a satisfies link. A correct refinement of the system specification would then automatically be a correct refinement of the properties, if the proof obligations of the satisfies link can be proved.
- The *formal analysis cases* are the proof obligations generated by VSE on refinement and satisfies links. Deduction units encapsulate data relevant for the proofs – they form a representation of the formal analysis cases within VSE.
- A *formal analysis procedure* is represented in VSE by an interactive proof of the generated proof obligations. VSE attempts to automatically prove the theorems but may require support by the user. VSE presents the current proof state to the user, and the user can choose the next rule to be applied or change the heuristics so that the system can complete the proof. Deduction units store partial and completed proofs.
- *Formal analysis results* are partial or completed proofs stored in deduction units.
- All *assumptions* that are used in a VSE proof need to be specified in the VSE system. Assumptions are therefore part of the VSE development graph. In order to distinguish assumptions from requirements for review they can be separated into specific deduction units reserved for assumptions.

4 DO-178C Scope of VCC

Microsoft Research's VCC [18] is a tool that can be used to verify that existing code conforms to requirements. The workflow starting with "code" it suggests conceptually therefore is "opposite" to VSE of the previous section 3. The largest piece of software verified by VCC has been Microsoft's Hyper-V [17].

Method. VCC belongs to a class of code verification tools (other examples are Caveat, Frama-C, KeY, SPARKAda, VeriFast) that apply backward-reasoning to calculate program properties: for each statement of a program, given a logical proposition that holds after the statement ("postcondition") and a certain rule set, it is possible to calculate the least

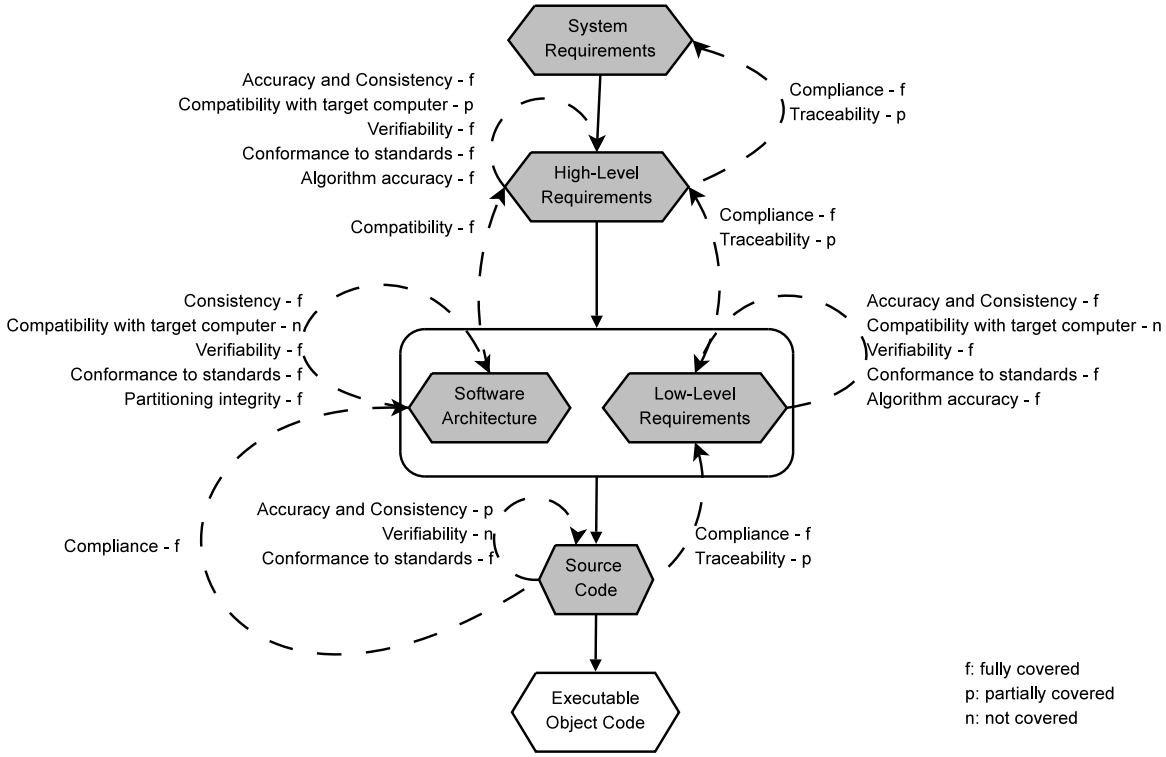

Figure 3: Scope of VSE

powerful proposition that has to hold before the statement as a first-order formula ("weakest precondition", WP [12]). It is checked whether the specification implies the WP ("verification condition"). If so, the implementation satisfies the specification.

User Interaction. The user interface is much simpler than VSE's. The engineer writes code annotations corresponding to LLRs to the code's functions (in first-order logic) and pushes the button "verify". After a tool run there are three possible outcomes: (a) verification is successful, (b) the tool can find a counter-example, (c) time-out: the tool cannot show compliance in reasonable time.

In case of (a) the result can be double-checked to detect contradictions in a precondition of a function. This feature called "smoke check" is very useful in practice, but at most a heuristic. In case of (b) the verification engineer can inspect the counter-example via the tool's model viewer. Counter-examples indicate that either the annotations are wrong (this is the typical case) or that they are too weak for the automated proof search. In case of (b) or (c) the task is to refine the auxiliary annotation in a subsequent iteration. However, to test whether a first-order logic formula holds in general is only semi-decidable. In practice this means that only for very simple functions it suffices to formulate just the pre- and postconditions. In all other cases, the engineer has to provide further auxiliary annotations to assert that certain properties hold at certain intermediate points so that the automatic inference engine is guided along the way. Annotations also can define specification state ("*ghost state*") useful for modelling low-level hardware or high-level properties of the system.

VCC in FMS Terms. When used to verify low-level requirements expressed as annotations co-located with functions of the source code (typical usage, Fig. 4), VCC has a smaller scope than VSE.

- The *properties* typically verified with VCC are low-level requirements.
- A *formal analysis case* is implementation code, its VCC annotations, and the parameters VCC is invoked with.
- A *formal analysis procedure* in VCC is the automatic generation of verification conditions and the subsequent invocation of the theorem prover to prove these conditions.
- The *formal analysis result* is the result reported by VCC after testing whether the proof attempt is accepted, as described above.
- *Assumptions*, if possible to be formalised by first order logic, can be expressed with the `requires` annotation.

5 DO-178C Compliance of VSE and VCC

The goal of this analysis is to examine how far the means of both methods and their tools can contribute to the demonstration of compliance to the objectives.

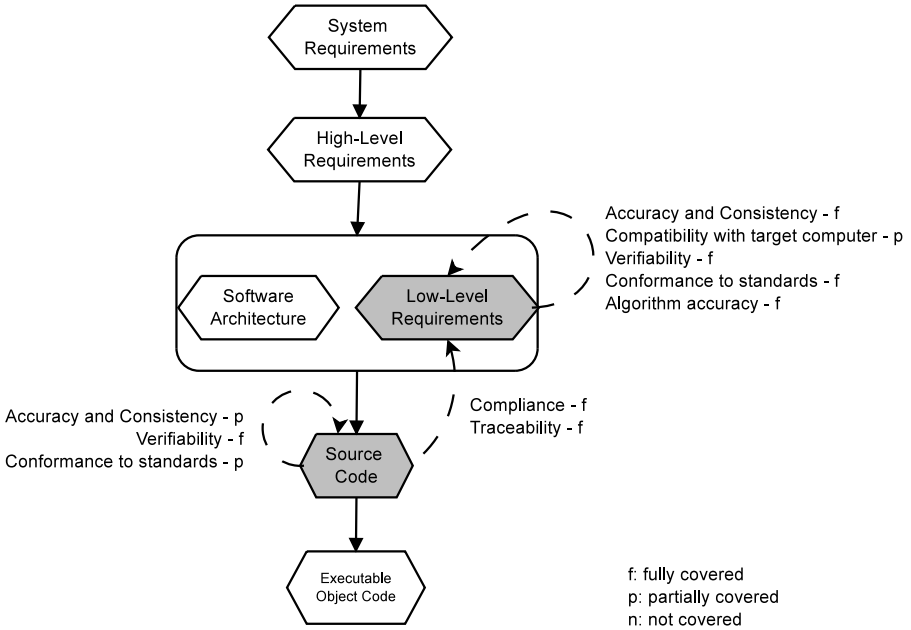

Figure 4: Scope of VCC

In a certification project, this analysis is documented, objective by objective, in the planning documents and then agreed with the certification authority. Similarly, for each method we collected answers to guidance questions derived from each objective, but due to lack of space, we only can summarise this discussion in the following subsections.

5.1 Coverage of the Core Objectives

Figures 3 and 4 display the results of our analysis on the core objectives for VSE and VCC respectively where the objectives have been tagged with the degree to which the method can support the demonstration of compliance: fully covered (f), partially covered (p), or not covered (n). We rate an objective as *fully covered* by the method if it can be assured by means of the method alone.

The objective is *partially covered*, if additional verification methods are necessary to show compliance with the objective, for example, by providing additional paper-and-pencil meta-arguments that cannot be formulated within the methods formalism.

An objective is considered as being *not covered* by the method if it cannot be assured by the means of that method at all; alternative verification methods such as testing must be applied instead.

For the specifications, there is full coverage of the objectives *compliance*, *accuracy*, *consistency*, *verifiability*, *standards conformance*, *algorithm accuracy*, and *software architecture compatibility* (VSE only), since this is the genuine strength of formal methods.

There is a fundamental difference between VSE and VCC with respect to source code: VSE generates it, whereas VCC operates on manually written source code. VCC complains if it is not able to verify the code and therefore ensures source code *verifiability*. On the other hand, the current code generator of VSE produces code that includes elements that are considered hard to verify, such as recursion for example.

In order to demonstrate *accuracy* and *consistency* for source code, DO-178C requires to consider properties such as stack and memory usage, worst case execution times, fixed point arithmetic overflow, etc. It may be possible to express some of these with formal notations of VSE and VCC, but other properties are just out of scope of the method, so that accuracy and consistency of source code can only partially be fulfilled by both methods.

VSE fully ensures the *conformance to standards* of the source code because it generates the code. For VCC, conformance to standards of the manually written source code must be checked with code reviews, but the tool assists with checking for some common programming errors.

Target computer compatibility is covered only partially as far as it is possible to express properties of the target environment with the formal notations of the methods. An example of modelling hardware in VCC (interrupt vector modelled by ghost state) is [2].

Traceability is only partially covered by both methods: Neither method provides any means to capture references to the informal requirements from which the formal requirements have been developed. It would be useful to enhance the tools in that respect. On the other hand, traceability between specifications and source code is fully covered – for VCC because the low-level requirements are co-located with the source code as annotations, and for VSE because the source code is generated. However, the code generator must be formally verified or qualified to support this argument, see section 6. The traceability between specifications that represent requirements or the software architecture is an interesting issue: natural language

requirements are often represented as a set of individual 'shall' statements which allows contemporary requirement management tools to manage traceability as a relation between these 'shall' statements. This is different for VSE, because currently the user cannot get a comprehensive output showing which axioms from a certain theory are necessary to establish the validity of a certain formula. With a coarser granularity, the development graph that displays refinement links between deduction units could be used as a means of showing traceability. It is also possible to display, for a proved lemma (that includes the proof obligations), the list of all lemmas and axioms that are used in the proof. Certification authorities and applicants must agree on a coarser interpretation of traceability that is suitable to formal methods such as VSE, and it is likely that additional guidance must be developed on this topic.

Partitioning integrity is difficult to assure by testing alone, so formal methods were considered as candidate for proper assurance early on [21]. In the Verisoft project, both VSE and VCC have been used to prove partitioning properties: for VSE, we adapted the approach of [13] to describe partitioning in PikeOS, and VCC has been used to verify certain spatial partitioning properties of the PikeOS kernel [3].

Robustness as well as *compliance with high-level and low-level requirements* are verification objectives of the executable object code in relation to high-level and low-level requirements. An accepted way to formally verify such properties is to establish property preservation between source and object code and then to verify the properties for the source code. However, since the executable object code is not within the scope of VSE and VCC, there are no means provided by these methods to show property preservation. Therefore, within the context of both VSE and VCC, robustness as well as compliance of object code to high-level and low level requirements, can only be verified traditionally with testing.

Completeness and correctness of the output *of the integration process* is not within the scope of VSE and VCC and must therefore be assured by other means.

5.2 Coverage of Objectives specific to Formal Methods

Table 1 summarises the tools' coverage of objectives specific to formal methods.

Correctness of Formal Analysis Cases, Procedures and Results. VSE and VCC both check the formal analysis cases automatically, in VSE as proof obligations, in VCC as verification conditions. The correctness of the proof obligations and verification conditions does not depend on an error prone manual process but on the correct implementation of the tools instead. The subsequent proofs and results depend on the correctness of the underlying calculus and the correctness of the implementation of the theorem provers. The correctness of the calculus constitutes the soundness of the formal methods, see below. The correctness of theorem provers is discussed in the context of tool qualification in section 6. The proof procedure itself is standardised and repeatable in both tools.

Formal Method Soundness. The concept of proofs in VSE relies on a very small set of basic rules that in turn are based on well known calculi. They have been shown to be sound, and the proof obligations generated by VSE have shown to be sufficient [19].

VCC's specification language is quite rich to cover most of the semantics of the C language. The full justification of its soundness is still work-in-progress, so at the time of writing we have rated its soundness as "partial". The fact that soundness of core language parts [8, 9] and the memory model [10] has been demonstrated as well as our experience with the tool gives us optimism that these efforts will be completed.

Correctness of Requirement Formalisation. This objective is concerned with the transition from informal statements to formal requirements. It can only be assured by review, independent of the formal method.

5.3 Coverage of Verification of Verification Objectives

Table 2 summarises the coverage of VSE and VCC on verification of verification objectives.

Requirements Coverage. All requirements are formulated in the notation of the formal methods. The tools generate the formal analysis cases as proof obligations on the complete set of requirements. Therefore, each tool assures completeness of requirements coverage.

Complete Coverage of Each Requirement. Proofs generated by VSE cover all paths through the abstract code that is a refinement from the specifications. The generated source code is a one-to-one mapping between abstract programs and generated source code so that there is a corresponding path through the abstract program specification. Proofs generated by VCC also cover all paths through the source code. For both tools, the proofs are generated by an automated theorem prover so all assumptions must be made explicit – this is an advantage of the automatic prover over a human prover. However, justification of the assumptions must be verified by manual review.

Unlike the FMS's unit proof example where there are simply no assumptions, in the verification of reactive systems assumptions often are of the form of global invariants. In VCC formal analysis cases for example, these global invariants are expressed as "requires" clauses, such as the global invariants in [3, Fig. 3] – their justification is manual [3, Sect. 3].

Completeness of the Set of Requirements. The weakest interpretation of this objective is a syntactic check that all input and all output variables of the program are indeed included in at least one formal statement. There is no tool support by either VSE or

Table 1: Coverage of Objectives specific to Formal Methods

	Coverage	
DO-178C Objective	VSE	VCC
Correctness of Formal Analysis Cases and Procedures	full	full
Correctness of Formal Analysis Results	full	full
Correctness of Requirement Formalisation	no	no
Formal Method Soundness	full	partial

Table 2: Coverage of Verification of Verification Objectives

	Coverage	
DO-178C Objective	VSE	VCC
Requirements Coverage	full	full
Complete Coverage of Each Requirement	full	full
Completeness of the Set of Requirements	partial	partial
Unintended Dataflow Detection	no	no
Dead Code Detection	no	no
Deactivated Code Detection	full	full

VCC for this check so it can only be performed manually. However, we do not believe that the DO-178C FMS regards this syntactic check as evidence for this objective.

In order to show that the output is specified for all input conditions, one can show that the disjunction of the preconditions (expressed in the specification language) of all operations cover all admissible input conditions. Conditions that are not permitted must be excluded via assumptions, as discussed above. These statements must be formulated manually – in VSE as a theorem, in VCC as an annotation. It can then be proved with assistance of the theorem provers of the tools.

In order to check that all output is covered by an input condition, one must create a logical statement that expresses that for all possible output of all output variables there is a combination of input variables and admissible input values that will produce the output in a way that is covered by the specification.

Again, in VSE and VCC, this statement must be prepared manually but can then be proved with assistance of the tools' theorem provers.

In summary: It is possible to formulate the completeness of the set of requirements as properties in the formal notations of both methods and then to prove these properties subsequently. There is however no support of both tools to automatically create these properties from existing specifications so that they must be created manually and then be reviewed for completeness and adequacy, hence our judgement "partial".

We want to point out, that completeness in general does not mean, that the complete set of possible input values as defined by the type of input parameters and the whole space of system states has to be covered. The operations to be specified may make sense only for a restricted subset of possible inputs. Then the specification will contain (implicit or explicit) preconditions, that describe, for which inputs the specification holds. In particular this is the case for VCC, where pointers can be used. Where this follows from properties of the system proved elsewhere, it is reasonable to assume that these pointers are valid and the structures they reference are well formed.

In a VSE state based specification, it may happen, that you can show that some states are not reachable from the initial state. Then it would not be necessary to specify the behaviour of the system in these unreachable states.

Unintended Dataflow Detection. Both methods and their tools do not provide a dataflow analysis capability, so that this objective must be ensured by other means.

Dead Code Detection. It is not possible to detect dead code with VSE. VCC provides the smoke test heuristic to detect code that is unreachable by logical specification. However, also VCC is not able to detect all dead code with the rigour intended in DO-178C and as possible with testing. Therefore, dead code detection must still rely on structural coverage analysis.

Deactivated Code Detection. It is usually possible to formally model the activation conditions of the deactivated code with the VSE or VCC notation and then to verify that the deactivated code is only executed in the intended configurations.

6 Tool Qualification

The coverage of some DO-178C objectives relies on the correct operation of the tools. DO-178 classifies tools as software development tool or as software verification tool. *Software development tools* are tools whose output is part of the airborne software – they therefore can introduce errors. The VSE code generator is an example of a software development tool. *Software verification tools* do not introduce errors into the airborne software but may fail to detect

them. The proof engines of both VSE and VCC as well as the generators of proof obligations and verification conditions are software verification tools.

DO-178 requires tool qualification if processes are reduced, eliminated or automated by the use of the tool without verifying their output with the verification methods listed in the standard. Some commercial tools such as the SCADE tool suite of Esterel Technologies offer qualification kits for their code generators. There are also a number of commercial verification tools, mostly for static analysis, structural test coverage analysis, and code verification.

Since the proofs are too complex to be constructed or even reviewed manually, the assurance of the tools' correctness is essential for the acceptance of the methods and their tools in industry. There are some approaches to demonstrate the correctness of the tools:

1. Formal verification of the tools: This may be a valid option for the VSE code generator. The proof engines of VSE and VCC, however, are very likely too complex for formal verification of themselves.

2. Standard tool qualification with testing as main verification method: This is feasible for the VSE code generator because it is not too different from some commercially available qualified code generators. The theorem provers are far more complex and it may be difficult to qualify them.

3. Another approach that has been proposed in the literature [11, 15] is to use a proof checker on the completed proofs. A proof checker is a much simpler tool so that its formal verification or qualification in accordance with DO-178C should be achievable with reasonable effort. However, this can only work if VSE and VCC can export the proof object data in a format that the proof checker can read.

4. Using different proof engines that employ different proof strategies, heuristics or decision procedures (*cross validation*). However, the proof engines must exchange proof data for which no standardised exchange format exist. Moreover, it is also possible that one prover may fail to prove a theorem that has successfully been proved by another prover.

5. Using service history as an argument for tool acceptance. The tools have been used in many projects and have improved over a long period of time. However, certification authorities will only accept service history as qualification evidence if the tools have been developed in compliance with DO-178B section 12.3.5. This is very likely not the case because the tools were developed in an academic work environment.

A combination of test-based tool qualification, proof checker and partial formal verification is probably the most viable approach at this time of writing. For VCC for example, work that could be used for a tool qualification has already started [4].

7 Summary

VSE is a very comprehensive method with a broad scope. Its major disadvantage is the need for the developer to master logical calculi such as ADT and TLA. Moreover, he or she must also understand the proof system in order to support the theorem prover effectively.

VCC, on the other hand, can be used as a standalone tool but can also be considered as an extension to the C programming language which is a familiar concept to developers, and can be executed within the development environment (Microsoft Visual Studio). A drawback, as a result of the relative ease of use, is the limited scope of established VCC use. Moreover, the language definition of VCC is still being worked upon at this time of writing, and syntax and semantics of the language are not fixed in a standard or manual.

The authors believe that the largest obstacle for using VSE and VCC in a certification context is their lack of tool qualification.

Despite FMS annex FM.B we found the verification of verification objectives difficult to apply to VSE and VCC, probably because they have been carried over from the testing method [5] and the standard tries to be as generic as possible.

In our analysis approach, the executable object code is out of scope for both methods. As a consequence it is not possible to formally verify the robustness and compliance of the executable objects code with the high-level and low-level requirements formulated in the methods annotations within the tools alone, without using meta-arguments about property preservation. Together with the lack of tool qualification the examined formal methods can only be used as a supplement to traditional verification methods. This incurs additional costs that industry will only accept for highly critical systems where testing alone will not provide enough confidence.

A highly critical function where the application of formal methods is advisable is for example the contingency handling of a UAV when losing the command and control link in civil airspace. We therefore propose to use VSE or VCC in a project like this to obtain real experience of the usage of these formal methods in a certification context.

This paper summarises the analysis of two of the formal methods that have been used in the Verisoft XT project, with respect to the objectives of the upcoming DO-178C, and identifies their strengths and weaknesses in that respect. It provides, in short form, the arguments to be provided to certification authorities when applying these methods in a real avionics project. The paper does not attempt to compare both methods with other existing methods in order to identify the most suitable method for this purpose.

However, the authors believe that this type of analysis as well as the arguments can be applied to other formal methods as well.

References

[1] S. Autexier, D. Hutter, B. Langenstein, H. Mantel, G. Rock, A. Schairer, W. Stephan, R. Vogt, and A. Wolpers. VSE: Formal methods meet industrial needs. *International Journal on Software Tools for Technology Transfer, Special issue on Mechanized Theorem Proving for Technology*, 3(1), 2000.

[2] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Formal verification of a microkernel used in dependable software systems. In B. Buth, G. Rabe, and T. Seyfarth, editors, *SAFECOMP 2009*, volume 5775 of *LNCS*, pages 187–200, Hamburg, Germany, 2009. Springer.

[3] C. Baumann, T. Bormer, H. Blasum, and S. Tverdyshev. Proving memory separation in a microkernel by code level verification. In *AMICS / ISORC*, 2011.

[4] T. Bormer and M. Wagner. Towards testing a verifying compiler. In Bernhard Beckert and Claude Marché, editors, *FoVeOOS*, volume KIT-INFO-TR 2010-13. Karlsruhe Institute of Technology, Technical Report, 2010.

[5] D. Brown, H. Delseny, K. Hayhurst, and V. Wiels. Guidance for using formal methods in a certification context. In *Proceedings of the Embedded Real Time Software and Systems Conference, Toulouse*, pages 1/7 – 7/7, 2010.

[6] D. Brown and K. Hayhurst. SC-205/WG-71 Information Paper: Formal Methods Technology Supplement. Technical Report IP0602 Rev. I, RTCA SC-205 / EUROCAE WG-71 Sub-Group 6, June 2009.

[7] L. Cheikhrouhou, G. Rock, W. Stephan, M. Schwan, and G. Lassmann. Verifying a chipcard-based biometric identification protocol in VSE. In Janusz Górski, editor, *SAFECOMP 2006*, volume 4166 of *LNCS*, pages 42–56. Springer, 2006.

[8] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, 2009. Available at `http://research.microsoft.com/vcc`.

[9] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 480–494, 2010.

[10] E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for C. *Electron. Notes Theor. Comput. Sci.*, 254:85–103, 2009.

[11] L.M.G. de Vries. Applying formal methods in the DO-178B certification process. Technical Report NLR TP 95547, National Aerospace Laboratory NLR, Amsterdam, The Netherlands, February 1996.

[12] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[13] B. L. DiVito. A model of cooperative noninterference for integrated modular avionics. In C. B. Weinstock and J. Rushby, editors, *Dependable Computing for Critical Applications*, volume 12, pages 269–286, San Jose, CA, 1999. IEEE Computer Society.

[14] D. Hutter, B. Langenstein, G. Rock, J. Siekmann, W. Stephan, and R. Vogt. Formal software development in the verification support environment. *Journal of Experimental and Theoretical Artificial Intelligence*, 12(4):383–406, 2000.

[15] J. Joyce. Use of machine-assisted theorem-proving as a means of verifying critical software in the context of RTCA DO 178C. In *Workshop on Theorem Proving in Certification, December 6 - 7, 2010, Cambridge, UK*, December 2010.

[16] B. Langenstein, R. Vogt, and M. Ullmann. The use of formal methods for trusted digital signature devices. In J. N. Etheredge and B. Z. Manaris, editors, *FLAIRS Conference*, pages 336–340. AAAI Press, 2000.

[17] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In A. Cavalcanti and D. Dams, editors, *FM*, volume 5850 of *LNCS*, pages 806–809. Springer, 2009.

[18] Microsoft Research. VCC homepage. `http://vcc.codeplex.com`.

[19] W. Reif. Vollständigkeit einer modifizierten Goldblattlogik und Ersetzung der Omega-Regel durch Induktion. Master's thesis, TU Karlsruhe, 1984.

[20] RTCA/EUROCAE. DO-178B/ED-12B Software Considerations in Airborne Systems and Equipment Certification, December 1992.

[21] J. Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical Report DOT/FAA/AR-99/58, SRI International, 2000.

[22] Das Verisoft XT Projekt. `http://www.verisoftxt.de`.