



HAL
open science

Obtaining DO-178C Certification Credits by Static Program Analysis

Daniel Kästner, Markus Pister, Christian Ferdinand

► **To cite this version:**

Daniel Kästner, Markus Pister, Christian Ferdinand. Obtaining DO-178C Certification Credits by Static Program Analysis. ERTS2022, Jun 2022, Toulouse, France. hal-03694553

HAL Id: hal-03694553

<https://hal.science/hal-03694553v1>

Submitted on 13 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Obtaining DO-178C Certification Credits by Static Program Analysis

Daniel Kästner, Markus Pister, Christian Ferdinand

AbsInt Angewandte Informatik GmbH, Science Park 1, D-66123 Saarbrücken, Germany

Abstract

Static analysis has evolved to be a standard method in the software development and verification process. Its formal method, Abstract Interpretation, is one of verification methods covered by the Formal Methods Supplement DO-333 of the DO-178C standard. Static program analysis can contribute to numerous verification goals of DO-178C at various stages of the development process. The main focus of static analysis methods are non-functional software quality hazards, e.g., violations of coding guidelines, violations of software architecture constraints, violations of resource bounds such as stack overflows and real-time deadlines, runtime errors, and data races. This article gives a brief overview of abstract interpretation and its applications to detect different classes of safety hazards. We will review the requirements of DO-178C/DO-333, from High-Level Requirements to requirements for verification of Executable Object Code, and pinpoint aspects that can be covered by static analysis methods. The article concludes with illustrating the relevant requirements for DO-330-compliant tool qualification of static analysis tools.

Keywords: DO-178C, DO-330, DO-333, certification, static analysis, abstract interpretation, tool qualification

1 Introduction

Some years ago, static analysis meant manual review of programs. Nowadays, automatic static analysis tools have been established in modern software development processes as they offer a tremendous increase in productivity by automatically checking the code under a wide range of criteria. Here, the term *static analysis* is used to describe a variety of program analysis techniques with the common property that the results are only based on the software structure. No execution of the program under analysis is needed. Static analysis can be applied to any kind of program representation, from the model level or the source code level to the executable object code level.

An important distinction of static analysis methods is the complexity of the program properties they aim at determining and the level of rigor at which they operate. In the simplest form, static analysis is focused on the program syntax: purely syntactical methods can be applied to check syntactical coding rules as contained in coding guidelines, such as MISRA C [39], SEI CERT C [44], or the Common Weakness Enumeration (CWE) [46]. They aim at a programming style that improves clarity and reduces the risk of introducing bugs. Compliance checking by static analysis tools has become common practice.

Syntactic rules play an important part in coding standards as they are easy to take into account while implementing code, and easy to check. However, ultimately, the objective is to prevent code defects which means that semantical properties have to be considered. To that end semantics-based static analyzers are needed which focus on the program semantics and compute invariants about variable values, pointer targets, etc. This is also relevant for coding standard compliance checking, since all commonly used coding guidelines also include semantical rules.

Depending on the level of rigor, semantics-based methods can be grouped into unsound and sound approaches, the essential difference being that when a sound method reports the property under analysis – such as freedom of runtime errors – as satisfied, this is guaranteed to be true. Abstract interpretation is a formal method for sound semantics-based static program analysis [9]. It supports formal correctness proofs: it can be proved that an analysis will terminate and that it is sound in the sense that it computes an over-approximation of the concrete program semantics. Abstract interpretation always provides full data and control coverage.

As of today, abstract interpretation-based static analyzers are most widely used to determine non-functional software quality properties [20, 17]. On the one hand that includes source code properties, such as compliance to coding guidelines, compliance to software architectural requirements, as well as absence of runtime errors and data races [24]. On the other hand also low-level code properties are covered such as absence of stack overflows and violation of timing constraints [21, 28].

Violations of non-functional software quality requirements often either directly represent safety hazards and cybersecurity vulnerabilities in safety- or security-relevant code, or they can indirectly trigger them. Hence they are invariably addressed by verification obligations in current safety and security norms, such as DO-178C [40], IEC-61508 [13], ISO-26262 [14], and EN-50128 [8].

In this article we will focus on the DO-178C standard [40], its formal method supplement DO-333 [41], and the DO-330 [42] which details the tool qualification requirements to be taken into account. We will review the software requirements and verification goals that are amenable to static analyses and hence identify the certification credits that can be obtained by applying different static analysis methods at different levels of the development process. Sec. 2 walks through the DO-178C norm and highlights the relevant software requirements and verification objectives. Sec. 3 illustrates the methodology of static program analysis and presents the fundamentals of its application

to compute various non-functional software properties. Sec. 4 then summarizes the verification goals supported by the various applications of static analysis. A brief overview of the tool qualification requirements relevant to static program analysis tools is given in Sec. 5, and Sec. 6 concludes.

2 DO-178C / DO-333

The DO-178C [40], published in December 2011, is a revision of DO-178B to take progress in software development and verification technologies into account. In general, the DO-178C aims at providing “guidance for determining, in a consistent manner and with an acceptable level of confidence, that the software aspects of airborne systems and equipment comply with airworthiness requirements.” It specifically focuses on model-based software development, object-oriented software, the use and qualification of software tools and the use of formal methods to complement or replace dynamic testing. Each of these key aspects is addressed by a dedicated *supplement* which modifies, complements, and completes the DO-178C core document. The supplements “should be used with and in the same way” as the core document [40]. In this overview we will specifically focus on the DO-178C core document and DO-333 (Formal Methods Supplement to DO-178C and DO-278A) [41].

The DO-178C first discusses general system aspects which are relevant for software development and defines the software life cycle processes, which then are addressed in turn: the software planning, development, and verification processes, as well as the configuration management, quality assurance, and certification processes. The norm also details the software life cycle data and addresses additional considerations, such as tool qualification requirements. In this section we will follow that structure and pinpoint the respective requirements and verification objectives amenable to static analysis techniques. Considerations for formal methods are addressed at the end of this section, tool qualification is addressed in Sec. 5.

2.1 System Aspects

System aspects relevant for software development include functional and operational requirements, performance requirements and safety-related requirements, including design constraints and design methods, in particular partitioning (cf. Sec. DO.2.1 of DO-178C [40]). The norm emphasizes that timing and performance characteristics require special attention since they affect the system software and the software-hardware boundaries and have to be included in the respective information flows.

The DO-178C defines five software levels (criticality levels) ranging from Level A (most critical) to Level E (least critical). According to Sec. DO.2.3 only partitioned software components can be assigned individual software levels by the system safety assessment process. Sec. DO.2.4, Architectural Considerations, states that “if partitioning and independence between software components cannot be demonstrated, all components are assigned the software level associated with the most severe failure condition to which the software can contribute”. The standard defines partitioning as a “technique for providing iso-

lation between software components to contain and/or isolate faults and potentially reduce the effort of the software verification process”. Among others, a partitioned software component “should not be allowed to contaminate another partitioned software component’s code, input/output, or data storage areas”, and it “should be allowed to consume shared processor resources only during its scheduled period of execution”. Thus, freedom of interference, both in the spatial and the temporal domain, is recognized as an important architectural property.

2.2 Software Planning & Development Process

In Sec. DO.4.4.2, Language and Compiler Considerations, the DO-178C points out that the software verification process needs to consider particular features of the programming language and compiler. In Sec. DO.4.5 the use of Software Development Standards is demanded which include Software Design Standards and Software Code Standards. One of the goals is to “disallow the use of constructs or methods that produce outputs that cannot be verified or that are not compatible with safety-related requirements”. The defined Software Development Standards have to be taken into account during software design and coding (cf. Sec. DO.5.2.2 and Sec. DO.5.3.2).

The *Software Design Standards* are defined to focus on algorithmic constraints like exclusion of recursion, dynamic objects, or data aliases (cf. Sec. DO.11.7e). They should also include complexity restrictions like maximum level of nested calls, use of unconditional branches, or number of entry/exit points of code components (cf. Sec. DO.11.7f).

The *Software Code Standards* focus on the programming language. They identify the programming language to be used and should define a safety-oriented language subset (cf. Sec. DO.11.8a). To improve readability (and hence verifiability) they should cover style rules like length restrictions, indentation, and documentation rules (cf. Sec. DO.11.8c), and impose further constraints on code complexity, e.g., regarding the complexity of logical and numerical expressions (cf. Sec. DO.11.8d).

2.3 Software Verification Process

Like the DO-178B, the DO-178C addresses the incompleteness of testing techniques: “Verification is not simply testing. Testing, in general, cannot show the absence of errors”. Since formal methods are sound they can completely satisfy some verification objectives while for others additional verification such as complimentary testing may be necessary. Purpose and objective of the software verification process are defined in the same way as in the DO-178B: The purpose of the software verification process is to detect and report errors that may have been introduced during the software development processes. Removal of the errors is an activity of the software development processes. The general objectives of the software verification process are to verify that the requirements of the system level, the architecture level, the source code level and the executable object code level are satisfied, and that the means used to satisfy these objectives are technically correct and complete.

As described in Sec. 2.1 non-functional software properties can affect the system and the software level, and consequently,

they are addressed at all levels of the software verification process. Design constraints may apply to ensure verifiability of the software.

The software verification process comprises reviews and analyses of high-level requirements, low-level requirements, the software architecture, the source code, and requires testing or formal analysis of the executable object code. One common verification objective at all levels is to demonstrate the compliance with the requirements of the parent level. As the system requirements include performance and safety-related requirements non-functional aspects like timing or storage usage can impact all stages. Consequently, the *compatibility with the target computer* is a verification objective among the high-level requirements, the low-level requirements, and at the software architecture level.

According to Sec. DO.6.2 the software verification activities have to address the “accuracy, completeness, and verifiability of the software requirements, software architecture, and Source Code”. In particular objective 6.3.2.d for reviews and analysis of low-level requirements demands to ensure that each low-level requirement can be verified. Objective 6.4.3.e demands to ensure that the Software Design Standards were followed during the software design process and that deviations from the standards are justified. Also during review and analyses of source code (Sec. DO.6.3.4) the Software Development Standards have to be addressed. Objective 6.3.4.c which aims at verifiability demands to ensure that no statements and structures that cannot be verified are contained in the Source Code. Objective 6.3.4.d demands to show conformance to the Software Code Standards defined, e.g., that complexity limits have been considered. Deviations from the standards have to be justified.

In Sec. DO.6.3 of [40] the system response time is given as an example of target computer properties relevant for high-level requirements and low-level requirements. Computing the response time requires the worst-case execution time to be known. At the source-code level the objective *accuracy and consistency* explicitly includes determining the worst-case execution time, the stack usage, and runtime errors (memory usage, fixed-point arithmetic overflow and resolution, floating-point arithmetic, use of uninitialized variables). All these characteristics can be checked using formal analysis (cf. Sec. FM.6.3.4 of [41]).

The data and control flow of the software is of crucial importance for the verification of functional and non-functional correctness properties. In the DO-178C, the verification goal 6.3.3.b (Consistency) demands that “a correct relationship exists between the components of the software architecture. This relationship exists via data flow and control flow.” It is complemented by the verification goal of Sec. DO.6.3.4.b (Compliance with the software architecture) which demands to “ensure that the source code matches the data flow and control flow defined in the software architecture”. Obviously the data and control flow of the implemented software must match the intended data and control flow as specified in the software architecture, and unintended data and control flow must be avoided. In particular, this implies demonstrating freedom of interference between

software components in mixed-criticality software. In addition, the data and control flow also determines the required effort for functional testing. In DO-178C, Objective 8 of Annex A Table A-7 requires that “Test coverage of software structure (data and control coupling) is achieved”, referencing Sec. DO.6.4.4.2.c which states that structural coverage analysis should “confirm that the requirements-based testing has exercised the data and control coupling between code components”. An in-depth discussion of data and control coupling analysis is given in [25].

Worst-case execution time and worst-case stack usage have to be considered at the Executable Object Code level. The reason is that the impact of the compiler, linker, and of hardware features on the worst-case execution time and stack usage has to be assessed. Both can be checked by formal analyses at the Executable Object Code level (cf. Sec. FM.6.7 of [41]). Runtime errors also can be addressed at the Executable Object level, e.g. to deal with robustness issues like out-of-range loop values and arithmetic overflows (cf. Sec. FM.6.7.b of [41]), or to verify the software component integration. The latter implies, e.g., detecting incorrect initialization of variables, parameter passing errors, and data corruption.

At the Executable Object Level complimentary testing is still required, e.g., to address transient hardware faults, or incorrect interrupt handling (cf. Sec. DO.6.4.3. of [40]). Formal analysis performed at the source code level can be used for verification objectives at the executable object code if property preservation between source code and executable code can be demonstrated (cf. Sec. FM.6.7.f of [41]).

2.4 Formal Methods (DO-333)

[41] defines formal methods as “mathematically based techniques for the specification, development, and verification of software aspects of digital systems”. It distinguishes three categories of formal analyses: deductive methods, such as theorem proving, model checking, and abstract interpretation. The computation of worst-case execution time bounds and the maximal stack usage are listed as reference applications of abstract interpretation. The importance of soundness is emphasized: “an analysis method can only be regarded as formal analysis if its determination of a property is sound. Sound analysis means that the method never asserts a property to be true when it is not true.”

Regarding the applicability of formal methods, the DO-333 states that “formal methods provide comprehensive assurance of particular properties only for those aspects that are formalized in the formal model, so defining the limits of the model is essential.” Formal analyses at the source code level have to be based on a formal source code semantics. For formal analyses done at the object code level, the object code becomes a formal model the semantics of which are treated as they are by the target hardware. When formal analysis is used to meet a verification objective, it has to be ensured that each formal method used is correctly defined, justified, and appropriate to meet this verification objective (cf. Sec. FM.6.2.1 of [41]):

- all notations used for formal analysis should be formal notations

- the soundness of each formal analysis method should be justified
- all assumptions related to each formal analysis should be described and justified; for example assumptions associated with the target computer or about the data range limits.

At the Executable Object Level, coverage analysis depends on whether formal methods are used to complement or to replace dynamic testing methods. When any low-level testing is used to verify low-level requirements for a software component then the entire software component will be subject to test coverage analysis consisting of requirements-based coverage analysis and structural coverage analysis. Requirements-based coverage analysis establishes that verification evidence exists for all of the requirements of the system. Structural coverage analysis is necessary since no exhaustive testing is achievable and used to ensure that the testing performed is rigorous and sufficient (cf. Sec. DO.6.4. of [40], Sec. FM.6.7.1 of [41], and Sec. FM.12.3.5 of [41]).

When only formal methods are used to verify requirements for a software component, a different coverage analysis for that component has to be performed, consisting of the following steps (cf. Sec. FM.6.7.2 of [41]):

- requirements-based coverage analysis to determine how well the implementation of the software requirements has been verified.
- complete coverage of each requirement to ensure that all assumptions made during the formal analysis are verified. If the assumptions are all verified then the formal analysis can give complete coverage of each requirement.
- completeness of the set of requirements
- detection of unintended dataflow relationships
- detection of extraneous code, including dead code and deactivated code
- review and analyses of the formal analysis cases, procedures, and results for the executable object code.

3 Static Analysis & Abstract Interpretation

Static analysis often is perceived as a technique for source code analysis at the programming language level, but it can also be applied at the binary machine code level. In that case it does not compute an approximation of a programming language semantics, but an approximation of the semantics of the machine code of the microprocessor.

The theory of abstract interpretation [9] is a mathematically rigorous formalism providing a semantics-based methodology for static program analysis. The semantics of a programming language is a formal description of the behavior of programs. The most precise semantics is the so-called concrete semantics, describing closely the actual execution of the program. Yet in

general, the concrete semantics is not computable. Even under the assumption that the program terminates, it is too detailed to allow for efficient computations. The solution is to introduce an abstract semantics that approximates the concrete semantics of the program and is efficiently computable. This abstract semantics can be chosen as the basis for a static analysis. Compared to an analysis of the concrete semantics, the analysis result may be less precise but the computation may be significantly faster. Abstract interpretation based static analyzers have been demonstrated to scale up to industry-size software projects containing millions of line of code [27, 30].

In the remainder of this section we will describe how static analysis, and abstract interpretation in particular, can be applied to code guideline checking, software architecture analysis, runtime error analysis, and to determine worst-case stack usage and worst-case execution times. Code guideline checking, software architecture analysis and run-time error analysis operate at the source code level. Worst-case stack usage analysis and worst-case execution time analysis are performed at the binary level, because they have to take the instruction set and hardware architecture into account. As explained above, a sound analysis computes a safe over-approximation of the concrete semantics and reports any potential defect of the defect classes under analysis:

- For worst-case execution time analysis soundness means that the reported WCET is never below the actual execution time in some execution environment. Overestimations may occur.
- In the same way, the computed stack height must never be below the stack usage in any concrete execution. Overestimations may occur.
- For run-time error analysis soundness means that the analysis never omits to signal an error that can appear in some execution environment. False alarms may occur.

3.1 Code Guideline Checking

Coding guidelines aim at improving code quality and can be considered a prerequisite for developing safety- or security-relevant software. In particular, obeying coding guidelines is strongly recommended by all current safety standards. The norms do not enforce compliance to a particular coding guideline, but define properties to be checked by the coding standards applied. As an example, the ISO 26262 gives a list of topics to be covered, including enforcement of low complexity, enforcing usage of a language subset, enforcing strong typing, and use of well-trusted design principles (cf. [15], Table 1). The language subset to be enforced should exclude, e.g., ambiguously defined language constructs, language constructs that could result in unhandled runtime errors, and language constructs known to be error-prone. As discussed in Sec. 2 the DO-178C is less prescriptive, but mandates coding guidelines nonetheless.

There is a variety of code guidelines, in particular for C and C++, which are widely used in industry. The most prominent guidelines for C are MISRA C:2012 [39], ISO/IEC TS 17961

[16], the SEI CERT C Coding Standard [44], and the MITRE Common Weakness Enumeration CWE [46]. Prominent coding standards for C++ include MISRA C++:2008 [38], the SEI CERT C++ Coding Standard [43], the C++ Core Guidelines [7], the Adaptive AUTOSAR C++ Coding Guidelines [4], and the Joint Strike Fighter Air Vehicle C++ Coding Standards [34]. However, as of 2021, the discussion which C++ language features should be admitted to which extent for safety-critical software projects, is in full swing, and there is no clear consensus yet [19].

Most coding guidelines try to make coding rules easy to follow for programmers, and easy to check by automatic tools, hence, many coding rules are formulated at the syntactic level. They can be addressed by static analyzers operating purely on the program syntax.

However, obeying syntactic coding guidelines can reduce the risk of programming errors but not prevent them. Hence, all coding standards also explicitly contain semantic rules. These rules require a deeper understanding of the code as they focus on semantical properties which requires knowledge about variable values, pointer targets etc. To address such rules, and, in consequence, identify semantical code defects, *semantics-based* static analyses can be applied. Many semantical rules are associated with runtime errors due to undefined or unspecified behaviors of the programming language used (cf. Sec. 3.5). All safety norms, including DO-178C, consider demonstrating the absence of such runtime errors explicitly as a verification goal. They typically address general runtime errors (e.g., division by zero, invalid pointer accesses, arithmetic overflows), and additionally consider corruption of content, synchronization mechanisms, and freedom of interference in concurrent execution. This is reflected, e.g., in MISRA C:2012, e.g., by Rule 1.3. (*goal: preventing undefined or critical unspecified behavior*) and Directive 4.1 (*goal: minimization of run-time failures*). Rule 1.3 and Directive 4.1 are examples for semantical guidance.

In contrast to other coding standards, MISRA C provides a clear classification which rules are based on semantic properties: typically such rules are labeled as system/undecidable. This is important since – due to their inherent undecidability – for semantical rules there cannot always be a correct and precise “yes” or “no” answer. As discussed above, there can always be false alarms (false positives), and, in case of unsound analyzers, also missed defects (false negatives). Therefore, developers have to be aware of the class and the operational context of the static analysis tool in use.

3.2 Software Architecture Analysis

All current safety norms require determining the data and control flow in the source code and making sure that it is compliant to the intended control and data flow as defined in the software architecture. In traditional static code analysis, data accesses via pointer variables and control flow by function pointer calls might be missed.

Using sound static analysis based on abstract interpretation, it is possible to guarantee the absence of runtime errors that could cause memory corruption and control flow corruption. Further-

more, it is possible to guarantee that in the analysis, all data and function pointer targets are considered and that the possible data and control coupling is fully captured. This way, a safe approximation of the data and control coupling between software components can be determined. That makes it possible to detect critical data and control flow errors and allows to complement traditional code coverage criteria by the degree of data and control coupling covered by the testing process, helping to identify relevant previously untested scenarios. In addition, freedom of spacial interference between software components can be demonstrated at the source code level [25].

3.3 Stack Usage Analysis

In safety-critical systems, stack overflows can cause catastrophic damage. The run-time stack (often just called “the stack”) typically is the only dynamically allocated memory area. It is used during program execution to keep track of the currently active procedures and facilitate the evaluation of expressions. The maximal stack usage has to be statically known: at configuration time of the system sufficient stack space has to be reserved for each task.

However, the stack height cannot be easily determined from the source code, since it depends on the dynamic call depth of functions, on compiler optimizations, and on link-time optimizations. Overestimating the maximum stack usage means wasting memory resources. Underestimation can lead to stack overflows where memory cells from the stacks of different tasks or other memory areas are overwritten. This can cause crashes due to memory protection violations and can trigger arbitrary erroneous program behavior, if return addresses or other parts of the execution state are modified. In consequence stack overflows are typically hard to diagnose and hard to reproduce, but they are a potential cause of catastrophic failure. One example is the series of accidents caused by unintended acceleration of the 2005 Toyota Camry: the expert witness’ report commissioned by the Oklahoma court in 2013 identifies a stack overflow as most probable failure cause [6, 47].

For safe stack size analysis, it is important to work on fully linked binary code, since the effects of code generation – inserting padding bytes, register allocation, etc. – or link-time optimizations have to be taken into account. Hence the static stack usage analysis cannot be based on the source code but must work on the executable machine code. It approximates the semantics of the machine code of the microprocessor by using an abstract model of the processor architecture. The abstract model does not need to cover the entire state of the microprocessor, only the parts affecting the stack space are needed. The hardware state relevant for worst-case stack analysis includes the processor registers and the memory cells. For a naive analysis, only the stack pointer register is needed, but for precise results it is important to perform an elaborate value analysis on the contents of processor register and memory cells.

In the following we will give an overview of the structure and analysis phases of StackAnalyzer [2], which is an example tool from this category. First, the control-flow graph is reconstructed from the input file, the binary executable. Then a static value analysis computes value ranges for registers and address

ranges for instructions accessing memory. StackAnalyzer reports computed branch and call instructions in case their targets cannot be automatically resolved by its value analysis, as well as unbounded recursions. Missing information such as recursion bounds and call targets can be manually specified in a formal annotation language, *AIS* [21]. Function pointer targets can also be automatically imported from an Astrée analysis. By concentrating on the value of the stack pointer during value analysis, StackAnalyzer computes how the stack increases and decreases along the various control-flow paths. This information can be used to derive the maximum stack usage of the entire task. StackAnalyzer takes the entire application into account and interprocedurally analyzes each call site with its precise stack height. The results of StackAnalyzer are presented as annotations in a combined call graph and control-flow graph. It shows the critical path, i.e., the path on which the maximum stack usage is reached which gives important feedback for optimizing the stack usage of the application under analysis. Experimental results show that the analysis is fast and precise so that only few explicit annotations are needed [21].

3.4 Worst-Case Execution Time Analysis

In real-time systems the overall correctness depends on the correct timing behavior: each real-time task has to finish before its deadline, hence, reliable bounds of the worst-case execution time (WCET) of real-time tasks have to be determined.

With end-to-end timing measurements timing information is only determined for one concrete input. Due to caches and pipelines the timing behavior of an instruction depends on the program path executed before. Therefore, usually no full test coverage can be achieved and there is no safe test end criterion. Techniques based on code instrumentation modify the code which can significantly change the cache and pipeline behavior (probe effect): the times measured for the instrumented software do not necessarily correspond to the timing behavior of the original software.

One safe method for timing analysis is static analysis by Abstract Interpretation which provides guaranteed upper bounds for WCET of tasks. Static WCET analyzers are available for complex processors with caches and complex pipelines, and, in general, support both single- and multi-core processors. A prerequisite is that good models of the processor/System on-Chip (SoC) architecture can be determined. However, there are modern high performance SoCs which contain unpredictable and/or undocumented components that influence the timing behavior. Analytical results for such processors are unrealistically pessimistic.

A hybrid WCET analysis combines static value, loop and path analysis with measurements to capture the timing behavior of tasks. Compared to end-to-end measurements the advantage of hybrid approaches is that measurements of short code snippets can be taken which cover the complete program under analysis. Based on these measurements a worst-case path can be computed.

In the following we will focus on applications of static analysis to WCET computation and illustrate the basic principles to establish their contribution to verification goals of DO-178C.

For an overview of methods and tools for WCET analysis we refer to [48], and recommend [33] for a general survey on methods for timing analysis on multi-core processors.

3.4.1 Timing Predictability

In general, a system is predictable if it is possible to predict its future behavior from the information about its current state [5]. The primary sources of uncertainty in execution time of an instruction sequence are the program input and the hardware state in which its execution begins. Hardware-related timing predictability can be expressed as the maximal variance in execution time due to different hardware states for an arbitrary but fixed input. Analogously, software-related timing predictability corresponds to the maximal variance in execution time due to different inputs for an arbitrary but fixed initial hardware state.

Even in single-core processors timing predictability is compromised by performance-enhancing hardware mechanisms like caches, pipelines, out-of-order execution, branch prediction and other mechanisms for speculative execution, which can cause significant variations in timing depending on the hardware state. On multi-core architectures, in addition the inter-core parallelism becomes relevant. To interconnect the several cores, buses, meshes, crossbars, and also dynamically routed communication structures are used. In that case, the interference delays due to conflicting, simultaneous accesses to shared resources (e.g. main memory) are the main cause of imprecision.

3.4.2 Fully Static WCET Analysis

Static analysis by Abstract Interpretation can provide guaranteed upper bounds for WCET of tasks. Over the past decades, a standard architecture has emerged [10, 12] which neither requires code instrumentation nor debug information and is composed of the following major building blocks:

Decoding: The instruction decoder identifies the machine instructions and reconstructs the call and control-flow graph. To ensure safety of later analysis results, this graph itself must be safe, i.e., all possible paths that can occur during execution of the program must be represented.

Value analysis: Value analysis aims at statically determining the contents of the registers and memory cells at each program point and for each execution context. The results of the value analysis are used to predict the addresses of data accesses, the targets of computed calls and branches, and to find infeasible paths caused by conditions that always evaluate to true, or always evaluate to false in a specific context.

Micro architectural analysis: The execution of a program is statically simulated by feeding instruction sequences from the control-flow graph to a micro-architectural timing model which is centered around the cache and pipeline architecture. It computes the system state changes induced by the instruction sequence at cycle granularity and keeps track of the elapsing clock cycles.

Path analysis: Based on the results of the combined cache/pipeline analysis the worst-case path of the analyzed code is computed with respect to the execution timing. The execution time of the computed worst-case path is the worst-case execution time for the task.

A tool which implements this architecture is the static WCET analyzer aiT [45]. It is available for a variety of microprocessors including multi-core processors which can be configured in a timing-predictable way to avoid or bound inter-core interferences like Infineon AURIX TC27x [1].

3.4.3 Hybrid WCET Analysis

Hybrid WCET analysis tools combine static context-sensitive path analysis with real-time instruction-level tracing to provide worst-case execution time estimates. By its nature, an analysis using measurements to derive timing information is aware of timing interference due to concurrent execution and multi-core resource conflicts, because the effects of asynchronous events (e.g. activity of other running cores or DRAM refreshes) are directly visible in the measurements.

An example tool is the hybrid WCET Analyzer TimeWeaver [29, 3], which builds on non-intrusive instruction-level tracing: the probe effect is avoided since no code instrumentation is needed. The computed estimates are safe upper bounds with respect to the given input traces, i.e., TimeWeaver derives an overall upper timing bound from the execution time observed in the given traces. Thus, the coverage of the input traces on the analyzed code is an important metric that influences the quality of the computed WCET estimates.

The trace information needed for running TimeWeaver is provided by embedded trace units of modern processors, like NEXUS IEEE-ISTO 5001, Infineon TriCore MCDS, or ARM CoreSight. They allow the fine-grained observation of a program execution on single- and multi-core systems.

The main inputs for TimeWeaver are the fully linked executable(s), timed traces and the location of the analyzed code in the memory (entry point, which usually is the name of a task or function). The analysis proceeds in several stages: decoding, loop/value analysis, trace analysis, and path analysis, most of which are shared with aiT [45]. The main difference is that micro-architectural analysis is replaced by the trace analysis stage:

Trace analysis: The given traces are analyzed such that each trace event is mapped to a program point in the control-flow graph and the trace points and segments are defined. In case a preemptive system has been traced, interrupts are detected and reported. The extracted timing information, i.e., the clock cycles which have been elapsed between two consecutive trace points are annotated to the CFG in a context-sensitive manner [29].

3.5 Run-Time Errors and Data Races

In this section we focus on source-level runtime errors due to undefined or unspecified behaviors of the programming language used. Examples are faulty pointer manipulations, numerical errors such as arithmetic overflows and division by zero,

data races, and synchronization errors in concurrent software. Such errors can cause software crashes, invalidate separation mechanisms in mixed-criticality software, and are a frequent cause of errors in concurrent and multi-core applications. At the same time, these defects also constitute security vulnerabilities, and have been at the root of a multitude of cybersecurity attacks, in particular buffer overflows, dangling pointers, or race conditions [23].

Runtime errors are one important cause of software-induced memory corruption in safety-critical systems. The other two main causes are stack overflows, and miscompilation, where the compiler silently generating erroneous code from a correct input program. As described above, with abstract interpretation, the absence of runtime errors and stack overflows can be proven; when using a formally proven compiler like CompCert [22, 18] no miscompilation is possible, hence all main sources of software-induced memory corruption can be covered.

In the following we will give a brief overview of the Astrée analyzer as an example of sound static runtime error analysis tools [31][37]. To achieve high precision Astrée provides a variety of abstract domains covering, e.g., intervals, octagons, digital filters, finite state machines, and interpolations. The memory domain empowers Astrée to exactly analyze pointer arithmetic and union manipulations and to perform a type-safe analysis of absolute memory addresses. Floating-point computations are precisely modeled while keeping track of possible rounding errors. Astrée also implements a low-level concurrent semantics [35] which provides a scalable sound abstraction covering all possible thread interleavings. The interleaving semantics enables Astrée, in addition to the classes of runtime errors found in sequential programs, to report data races, and lock/unlock problems, i.e., inconsistent synchronization. The set of shared variables does not need to be specified by the user: Astrée assumes that every global variable can be shared, and discovers which ones are effectively shared, and on which ones there is a data race. Since Astrée is aware of all locks held for every program point in each concurrent thread, Astrée can also report all potential deadlocks. The abstract domains are parameterized, which enables users to fine-tune the precision of the analyzer to the software under analysis to minimize the number of false alarms.

In its data and control flow analysis module, Astrée tracks accesses to global, static, and local variables in case those accesses are made outside of the frame in which the local variables are defined (e.g., because their address is passed into a called function). The soundness of the analysis ensures that all potential targets of data and function pointers are taken into account. Function pointer targets are automatically resolved and can be exported in AIS format to support binary-level static analyzers. Astrée's data and control flow reports show the number of read and write accesses for every global, static, and out-of-frame local variable, lists the location of each access and shows the function from which the access is made. All variables are classified as being thread-local, effectively shared between different threads, or subject to a data race. Astrée also supports data and control coupling analysis [26], and can check compliance to commonly used coding guidelines such as MISRA C/C++,

CWE, SEI CERT C/C++, Adaptive Autosar C++, etc. Furthermore, Astrée includes a program slicer, and a user-configurable taint analysis [24].

Practical experience on avionics and automotive industry applications are given in [31][36][32]. They show that industry-sized programs of millions of lines of code can be analyzed in acceptable time with high precision for runtime errors and data races.

4 Coverage of Verification Objectives

In this section we give an overview of the verification objectives of DO-178C to which static analysis methods can be applied. The relevant verification objectives are summarized in Annex-A of [41]. We will explicitly list the sections of the DO-178C and the DO-333 that address verification objectives to which sound static analysis for WCET (worst-case execution time), WCSU (worst-case stack usage), and RTE (runtime errors), as well as simple unsound code guideline checking (CG) can contribute. The section names of the DO-333 match the corresponding sections in the DO-178C, e.g. Sec. FM.6.3.1.c of DO-333 corresponds to Sec. 6.3.1.c of DO-178C, so in the following we will just use the numbering from [41] for simplicity.

In general, worst-case execution time analysis, worst-case stack usage analysis and runtime error analysis contribute to all objectives related to the target environment. Stack usage, response times and execution times are determined by the target computer. They can cause violations of high-level requirements, violations of low-level requirements, incompatibility of the software architecture with the target computer, and they can affect the accuracy and consistency of the source code. Sound static source code analysis enables sound data and control flow analysis which is required to demonstrate consistency between software architecture level and source code level. Detecting runtime errors is required to deal with robustness issues like out-of-range loop values and arithmetic overflows, or to verify the software integration. The latter implies, e.g., detecting incorrect initialization of variables, parameter passing errors, and data corruption.

The sections Sec. FM.6.3.1.c, FM.6.3.2.c, and FM.6.3.3.c address the compatibility with the target computer, hence, sound static analyses for WCET, WCSU, and RTE are relevant for all of them. The timing aspect is emphasized as response times and is explicitly listed as an example in Sec. FM.6.3.1.c and Sec. FM.6.3.2.c. Sound analyzers for WCET, WCSU and RTE can report unreachable code and dead code, thus also contributing to Sec. FM.6.3.3.a, Sec. FM.6.3.4.a, and Sec. FM.6.3.4.e. Compliance to low-level requirements (Sec. FM.6.3.4.a) also necessitates the absence of programming defects as reported by sound RTE analysis. The data and control flow analysis provided by sound RTE analysis relates to Sec. FM.6.3.3.b and FM.6.3.4.b. Subsequently, Sec. FM.6.3.4.f explicitly lists determining worst-case execution time, stack usage, and absence of runtime errors as verification objectives.

Sec. FM.6.7 addresses the formal analysis of the executable object code. Sound analysis of WCET and WCSU both contribute to Sec. FM.6.7.e by computing safe bounds to the worst-

Obj.	Ref.	WCET	WCSU	CG	RTE
Table FM.A-3					
3	FM.6.3.d FM.6.3.1.c	✓	✓		✓
Table FM.A-4					
3	FM.6.3.d FM.6.3.2.c	✓	✓		✓
5	FM.6.3.f FM.6.3.2.e			✓	✓
8	FM.6.3.3.a	✓*	✓		✓
9	FM.6.3.c FM.6.3.3.b	✓*	✓*		✓
10	FM.6.3.d FM.6.3.3.c	✓	✓		✓
11	FM.6.3.e FM.6.3.3.d	✓*	✓*	✓	✓
12	FM.6.3.f FM.6.3.3.e	✓*	✓*	✓	✓
13	FM.6.3.3.f	✓	✓		✓
Table FM.A-5					
1	FM.6.3.a FM.6.3.4.a	✓	✓		✓
2	FM.6.3.a FM.6.3.4.b				✓
3	FM.6.3.e FM.6.3.4.c			✓	✓
4	FM.6.3.f FM.6.3.4.d			✓	✓
5	FM.6.3 FM.6.3.4.e			✓	✓
6	FM.6.3.b FM.6.3.c FM.6.3.4.f	✓	✓	✓	✓
7	6.3.5.a	✓	✓		✓
Table FM.A-6					
1	FM.6.7.a FM.6.7.c	✓	✓		✓**
2	FM.6.7.b FM.6.7.c	✓	✓		✓**
3	FM.6.7.d FM.6.7.c	✓	✓		✓**
4	FM.6.7.b FM.6.7.c	✓	✓		✓**
5	6.4.e FM.6.7.e	✓	✓		✓**
* Leveraging the built-in value analysis ** Using CompCert to ensure semantic preservation					

Table 1: Coverage of DO-178C verification objectives by sound static WCET analysis (WCET), sound static stack usage analysis (WCSU), static code guideline checking (CG) and sound static runtime error analysis (RTE).

case execution time or stack consumption of the software under certification. The possibility to deeply investigate the behavior of the analyzed software on the assembly level allows to derive information about timing contribution of specific parts of the software but also to trace back contents of memory cells and/or register values to the corresponding source code. Hence sound static binary code analysis as needed for WCET and WCSU also contributes to the paragraphs *a* till *d* in Sec. FM.6.7.

As specified in Sec. FM.6.7.f, relevant properties can also be verified on source code level if property preservation between source and object code level is ensured. For the language C, this can be achieved by using the formally verified compiler CompCert [22]. The code it produces is proven to behave exactly as specified by the semantics of the source C program. Leveraging this, sound static RTE analysis is relevant for all points in Sec. FM.6.7 and also contributes to Sec. FM.6.6.a: program slicing and taint analysis, e.g., as available in Astrée, support identifying program parts contributing to run-time errors, and program parts affected by data corruption, respectively, hence possibly pinpointing flaws in the Data Item File. Furthermore a type-safe analysis of absolute addresses at the source code level as available in Astrée can detect incorrect hardware addresses, and also reports memory overlaps. The value analysis component of sound binary-level analyzers for WCET and WCSU supports proving that memory accesses are made to the expected memory regions at the Executable Object Level. These analyses contribute to Sec. FM.6.3.5.a which deal with review and analyses of the output of the integration process.

Code guideline checkers, either as standalone tools, or included in sound RTE analyzers such as Astrée typically also compute code metrics, and provide checks to demonstrate conformance to thresholds defined. These capabilities support requirements of the Software Design Standards (e.g., recursion, dynamic objects, call nesting levels) and of the Software Code Standards (e.g., style rules, expression complexity). They also support traceability requirements by appropriate coding rule checks. In summary, they contribute to various verification objectives, in particular the objectives Sec. FM.6.3.3.d, Sec. FM.6.3.3.e, Sec. FM.6.3.4.c, Sec. FM.6.3.4.d, and Sec. FM.6.3.4.e, in general supporting to check the verifiability of software design and implementation.

5 Tool Qualification

Whenever the output of a tool is either part of a safety-critical system to be certified or the tool output is used to eliminate or reduce any development or verification effort for such a system, that tool needs to be qualified. DO-178C regulates *when* a tool qualification is to be applied and DO-330 [42] *gives guidance* on the tool qualification requirements.

5.1 Relevant Tool Qualification Levels (TQL)

First, the necessary qualification activities and results have to be identified. For this, DO-330 defines the so-called tool qualification level (TQL). There are five different levels, from the most critical level TQL-1 down to TQL-5.

The TQL is determined by the potential tool impact and the

software level. There are three tool impact categories where *Criteria 1* denotes the highest impact and *Criteria 3* the lowest. *Criteria 1* does not apply to static verification tools since it is associated with tools whose output is part of the airborne software. Analysis and verification tools are subject to *Criteria 2/3*. The difference between the latter two categories is whether the output of the tool is used to justify the elimination or reduction of other verification or development activities or not. The following table illustrates how the TQLs are assigned based on *Criteria* and *Software Level*.

Software Level	Criteria		
	1	2	3
A	TQL-1	TQL-4	TQL-5
B	TQL-2	TQL-4	TQL-5
C	TQL-3	TQL-5	TQL-5
D	TQL-4	TQL-5	TQL-5

To summarize, the tool qualification levels relevant for static analyzers are TQL-4 and TQL-5. TQL-4 applies when the software under analysis is Level A/B and the tool categorized as criteria 2. In all other cases, TQL-5 applies. In the following, we summarize the qualification material that is required for a TQL-4 qualification, and outline a well-proven example on how this can be supported.

5.2 Tool Qualification Requirements

The tables *T-0* to *T-10* in DO-330 define requirements (called objectives) to the tool qualification. Each table addresses a different process associated with the life cycle of the tool under qualification in order to cover the relevant aspects of the affected processes.

Summarizing the objectives in these tables, the qualification data to be provided for a TQL-4 verification tool boils down to the following. First, tool (operational) requirements have to be specified, i.e., the intended tool behavior must be defined in an explicit and detailed manner. Second, test cases have to be created which cover all those functional requirements. This includes providing descriptions of test case objectives, execution procedures, expected results, and records of their execution. Unique identifiers for requirements and test cases allow to establish trace data between those data elements which in the end allows to claim coverage of all requirements by successfully passed test case executions. For the static analyzers described in Sec. 3 these requirements are fully covered by so-called *Qualification Support Kits* (QSKs).

In addition to the tool behavior, the qualification material needs to address certain aspects of the tool development processes. The *Tool Qualification Plan* (TQP) gives an overview to the tool qualification as a whole. The *Tool Development Plan* (TDP) includes the objectives, standards, and tool life cycle(s) to be used in the tool development processes. The *Tool Verification Plan* (TVP) is a description of the activities to satisfy the tool verification process objectives. The *Tool Configuration Management Plan* (TCMP) establishes the methods to be used to satisfy the objectives of the TCM process throughout the tool life cycle. The *Tool Quality Assurance Plan* (TQP) establishes the methods to be used to satisfy the objectives of the tool quality assurance process. The execution of all activities defined in

these plans needs to produce evidence records that are archived and part of the qualification data. AbsInt provides these evidences about the applied tool life cycle activities as part of the QSKs.

Typically, static analyzers are categorized as so-called *Commercial Off-The-Shelf* (COTS) tools, basically meaning that the tool developer is different from the tool user. To qualify COTS tools, the tool developer provides the basis for the required qualification material from a “tool developers” perspective. The tool user then needs to either adapt the material or to describe how its tool use maps to the data provided by the developer. For example: the tool user might needs to define how the tool is used and which functionality (from the provided operational requirements) is used.

To provide high confidence in the correct functioning of a tool it is necessary to demonstrate that the tool works correctly in the operational context of its users. This is a common requirement of most current safety standards. The correct functioning of a tool might be affected by the OS version, system libraries installed, software patch levels, etc. For the tools listed in Sec. 3 there is dedicated support for tool users to automatically execute the QSK test cases and automatically create all data required for the certification package.

6 Conclusion

Static analysis has evolved to be a standard method in the software development and verification process. It can be applied to various verification activities required for DO-178C certification, in particular by performing code guideline checking, data and control coupling analysis, interference analysis, worst-case stack and execution time analysis, and runtime error analysis. In this article we precisely identified the verification requirements and objectives that can be covered by static analysis and its formal method, abstract interpretation. For each application of static analysis mentioned above we summarized the underlying analysis concepts and illustrated them with practical examples. We also summarized the required tool qualification activities, and illustrated them with a well-proven example approach to tool qualification.

References

- [1] AbsInt GmbH. aiT WCET Analyzer Website. <http://www.AbsInt.com/ait>.
- [2] AbsInt GmbH. StackAnalyzer Website. <http://www.AbsInt.com/sa>.
- [3] AbsInt GmbH. Timeweaver website.
- [4] AUTOSAR. Guidelines for the use of the C++14 language in critical and safety-related systems, 2018.
- [5] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, and W. Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, 2013. Accepted.
- [6] M. Barr. Bookout v. Toyota, 2005 Camry software Analysis by Michael Barr. <http://www.safetyresearch.net/Library/BarrSlides.html>, volume 3666 of LNCS, pages 202–213. Springer, September 2013.
- [7] H. S. Bjarne Stroustrup. C++ Core Guidelines. <https://isocpp.github.io/CppCoreGuidelines/CppCoreG> [retrieved: Jan. 2020].
- [8] CENELEC EN 50128. Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems, 2011.
- [9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL’77*, pages 238–252. ACM Press, 1977.
- [10] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. Phd thesis, Uppsala University, 2003.
- [11] C. Faure and V. Delebarre. Automatic proof of freedom from interference with iffree. In *Proceedings of the 10th European Conference on Software Architecture Workshops, Copenhagen, Denmark, November 28 - December 2, 2016*, page 36, 2016.
- [12] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of LNCS, pages 469–485. Springer, 2001.
- [13] IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.
- [14] ISO 26262. Road vehicles – Functional safety, 2018.
- [15] ISO 26262. Road vehicles – Functional safety – Part 6: Product development at the software level, 2018.
- [16] ISO/IEC. Information Technology – Programming Languages, Their Environments and System Software Interfaces – Secure Coding Rules (ISO/IEC TS 17961), Nov. 2013.
- [17] D. Kästner. Applying Abstract Interpretation to Demonstrate Functional Safety. In J.-L. Boulanger, editor, *Formal Methods Applied to Industrial Complex Systems*. ISTE/Wiley, London, UK, 2014.
- [18] 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.
- [19] D. Kästner, C. Cullmann, G. Gebhard, S. Hahn, T. Karos, L. Mauborgne, S. Wilhelm, and C. Ferdinand. Safety-Critical Software Development in C++. In A. Casimiro, F. Ortmeier, E. Schoitsch, F. Bitsch, and P. Ferreira, editors, *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops*, pages 98–110, Cham, 2020. Springer International Publishing.
- [20] D. Kästner and C. Ferdinand. Efficient Verification of Non-Functional Safety Properties by Abstract Interpretation: Timing, Stack Consumption, and Absence of Runtime Errors. In *Proceedings of the 29th International System Safety Conference ISSC2011*, Las Vegas, 2011.
- [21] D. Kästner and C. Ferdinand. Proving the Absence of Stack Overflows. In *SAFECOMP ’14: Proceedings of the 33th International Conference on Computer Safety, Reliability and Security*, volume 8666 of LNCS, pages 202–213. Springer, September 2014.

- [22] 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.
- [23] D. Kästner, L. Mauborgne, and C. Ferdinand. Detecting Safety- and Security-Relevant Programming Defects by Sound Static Analysis. In J.-C. B. Rainer Falk, Steve Chan, editor, *The Second International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2017)*, volume 2 of *IARIA Conferences*, pages 26–31. IARIA XPS Press, 2017.
- [24] D. Kästner, L. Mauborgne, S. Wilhelm, and C. Ferdinand. High-Precision Sound Analysis to Find Safety and Cybersecurity Defects. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, Jan. 2020.
- [25] D. Kästner, L. Mauborgne, S. Wilhelm, and C. Ferdinand. Static Data and Control Coupling Analysis. *Submitted to the 11th European Congress on Embedded Real Time Software and Systems (ERTS 2022)*, March 2022.
- [26] D. Kästner, L. Mauborgne, S. Wilhelm, and C. Ferdinand. Static Data and Control Coupling Analysis. In *ERTS2 2022 - Embedded Real Time Software and Systems. To appear*, Toulouse, France, Mar. 2022.
- [27] D. Kästner, A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, A. Schmidt, H. Hille, S. Wilhelm, and C. Ferdinand. Finding All Potential Runtime Errors and Data Races in Automotive Software. In *SAE World Congress 2017*. SAE International, 2017.
- [28] D. Kästner, M. Pister, G. Gebhard, M. Schlickling, and C. Ferdinand. Confidence in Timing. *Safecomp 2013 Workshop: Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR)*, September 2013.
- [29] D. Kästner, M. Pister, S. Wegener, and C. Ferdinand. TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis. In S. Altmeyer, editor, *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, volume 72 of *OpenAccess Series in Informatics (OASICS)*, pages 1:1–1:11, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [30] D. Kästner, B. Schmidt, M. Schlund, L. Mauborgne, S. Wilhelm, and C. Ferdinand. Analyze This! Sound Static Analysis for Integration Verification of Large-Scale Automotive Software. In *Proceedings of the SAE World Congress 2019 (SAE Technical Paper)*. SAE International, 2019.
- [31] D. Kästner et al. Finding All Potential Runtime Errors and Data Races in Automotive Software. In *SAE World Congress 2017*. SAE International, 2017.
- [32] D. Kästner et al. Analyze This! Sound Static Analysis for Integration Verification of Large-Scale Automotive Software. In *Proceedings of the SAE World Congress 2019 (SAE Technical Paper)*. SAE International, 2019.
- [33] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3), jun 2019.
- [34] L. Martin. Joint strike fighter air vehicle c++ coding standards for the system development and demonstration program, 2005.
- [35] A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):63, Mar. 2012.
- [36] A. Miné and D. Delmas. Towards an Industrial Use of Sound Static Analysis for the Verification of Concurrent Embedded Avionics Software. In *Proc. of the 15th International Conference on Embedded Software (EMSOFT'15)*, pages 65–74. IEEE CS Press, Oct. 2015.
- [37] A. Miné et al. 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.
- [38] MISRA (Motor Industry Software Reliability Association) Working Group. MISRA C++:2008 Guidelines for the use of the C++ language in critical systems, 2008.
- [39] MISRA (Motor Industry Software Reliability Association) Working Group. *MISRA-C:2012 Guidelines for the use of the C language in critical systems*. MISRA Limited, Mar. 2013.
- [40] Radio Technical Commission for Aeronautics. RTCA DO-178C. Software Considerations in Airborne Systems and Equipment Certification, 2011.
- [41] Radio Technical Commission for Aeronautics. RTCA DO-333. Formal Methods Supplement to DO-178C and DO-278A, 2011.
- [42] Radio Technical Commission for Aeronautics. Software Tool Qualification Considerations, 2011.
- [43] Software Engineering Institute SEI – CERT Division. SEI CERT C++ Coding Standard.
- [44] Software Engineering Institute SEI – CERT Division. *SEI CERT C Coding Standard – Rules for Developing Safe, Reliable, and Secure Systems*. Carnegie Mellon University, 2016.
- [45] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. 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.
- [46] The MITRE Corporation. CWE – Common Weakness Enumeration. <https://cwe.mitre.org> [retrieved: July 2019].
- [47] Transcript of Morning Trial Proceedings had on the 14th day of October, 2013 Before the Honorable Patricia G. Parrish, District Judge, Case No. CJ-2008-7969. http://www.safetyresearch.net/Library/Bookout_v_Toy October 2013.
- [48] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.
- [49] B. Zimmer, C. Dropmann, and J. U. Hanger. A systematic approach for software interference analysis. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, pages 78–87, 2014.