Programming language evaluation criteria for safety-critical software in the air domain.

ASHMORE, R., HOWE, A., CHILTON, R. and FAILY, S.

2022

Programming Language Evaluation Criteria for Safety-Critical Software in the Air Domain

Rob Ashmore, Andrew Howe, Rhiannon Chilton
Defence Science and Technology Laboratory
Portsdown West, UK
{rdashmore,aghowe,rchilton}@dstl.gov.uk

Shamal Faily
Robert Gordon University
Aberdeen, UK
s.faily@rgu.ac.uk

***Abstract*—Safety-critical software in the air domain typically conforms to RTCA DO-178C. However, latent failures might arise based on assumptions underpinning the programming language used to write the software, whereas the lack of empirical data may constrain the selection of a promising but untested language. To overcome this difficulty, we propose evaluation criteria drawn from RTCA DO-178C, to help quickly review the potential applicability of programming languages in the air domain. We illustrate the constraints by using them to evaluate the suitability of the Rust programming language.**

***Index Terms*—Programming languages, software and system safety.**

I. Introduction

Aircraft are long running platforms. Their design and engineering can take years, if not decades, with some aircraft remaining operational for several generations. Software used in the air domain is subject to certification, both at the outset of an aircraft's design and throughout its life. Such certification requires compliance with a given standard. The most commonly used standard for developing and certifying safety-critical software in the air domain is RTCA DO-178C [1]. Certification standards like RTCA DO-178C focus on the assurance of software, considering attributes of the software development process and the resulting product. The influence of the programming languages used to develop this software is less obvious, but remains of critical importance.

A classic programming language like C might be considered "safe" because it has been used successfully for several years. But, like any long-running technology innovation, decisions that underpinned a language's selection may have relied on assumptions which, with the passage of time, the drive for innovation, and a new generation of language maintainers, may be forgotten [2] or invalidated (e.g. in cases where memory allocation never returns a failure code [3]). As such, even when no changes are made to software, language selection could be a source of latent failures. Conversely, the lack of empirical data about how suitable a "novel" (in the context of safety-critical air domain software) programming language might be could render its use too risky given the long-term impact of its selection. This may be a sound decision, or it may prevent new languages (and language features) being used, even though these could conceivably increase safety, and productivity.

Although previous work has looked at the applicability of particular languages for high-assurance software, e.g. [4], such an impasse suggests the need for clearly identified programming language evaluation criteria that can be used to evaluate the suitability of programming languages in the air domain. Such criteria could inform the initial selection of a novel language (or a language subset) at the earliest stages of safety-critical software design. Moreover, given the risks that result from subjectivity when determining the extent of a system's change during recertification [5], the criteria could also help evaluators determine a language's continued suitability for new development activity.

To explore the form such criteria might take, this paper proposes a set of evaluation criteria, based on RTCA DO-178C, for reviewing the applicability of programming languages in the air domain. While not a focus of this work, we suggest that these criteria are likely applicable to other domains, as well.

We present some related work to our criteria in Section II, before presenting the criteria in Section III. We illustrate their application using Rust in Section IV. We conclude by summarising the contributions and implications of this work in Section V.

II. Related Work

RTCA DO-178C was released in 2011, and comprises descriptive paragraphs, summarised in 71 tabulated objectives. Since its release, the number of developments in programming languages and tools have been considerable.

RTCA DO-178C uses Software Levels to direct effort towards items with the greatest safety consequences. Four technology-specific supplements are associated with RTCA DO-178C. If a relevant technology is being used then, in addition to RTCA DO-178C, compliance with relevant supplements are also required. The supplements cover tool qualification, model-based development, Object-Oriented Technology and Related Techniques (OOT&RT) and formal methods. Of these, only the OOT&RT supplement (i.e. RTCA DO-332 [6]) influences programming language choice in the manner we are concerned with.

The adoption of formal methods (i.e. RTCA DO-333 [7]) and the choice of programming language are linked, not least because some languages are explicitly designed to support formal methods. However, in such cases, the adoption of formal methods is likely to be the driving factor in the

decision. That is, the choice of development approach leads to the programming language and the certification standard supplement. This language-first, standard-second influence is opposite to the one we are concerned with, where the choice of certification standard (specifically, RTCA DO-178C) leads to constraints on the programming language.

Although certification standards assess software quality, RTCA DO-178C has also been used to derive evaluation criteria for other software engineering artifacts. Marques and de Cunha [8] devised process requirements for certifying airborne military software from several standards, including RTCA DO-178C, but these requirements are agnostic of programming language. Sarkis et al. [9] have drawn recommendations on the use of design models from RTCA DO-178C. Although their work does not consider programming languages, it does consider areas where programming langauge choice could have an impact, such as traceability and automatic code generation.

Because user studies evaluating programming languages are expensive, Sadowski & Kurniawan [10] have examined the effectiveness of usability inspections methods like heuristic evaluation [11]. Drawing from the well-known usability evaluation heuristics and design principles for evaluating programming languages [12], they developed eleven language feature heuristics including areas such as consistency, error-proneness, and hidden dependencies.

III. Evaluation Criteria

A. Deriving the Criteria

Drawing on Sadowski & Kurniawan's [10] idea of using principles as a "discount" evaluation criteria, we used RTCA DO-178C to derive evaluation criteria for assessing programming language applicability for safety-critical software in the air domain. Because it represented the most stringent level of assurance, we focused on Software Level A of RTCA DO-178C. Although not all software within a modern air system will be at this level, it provides a useful basis for novel programming languages for any airborne application.

The criteria were devised by applying a three-step process.

1) The main body of RTCA DO-178C was examined, to identify paragraphs influencing programming language choice. For example, Para. 4.5c indicates that language constructs incompatible with safety-critical requirements are prohibited. This might mean C coding standards prevent use of `void*`. These paragraphs are concerned with easy detection of unsuitable constructs (e.g. via static analysis tools), or whether a programming language's design guarantees their absence.
2) We reviewed Annex A of RTCA DO-178C to identify specific objectives influencing programming language choice. For example, Table A-5 Objective 3 requires that source code be verifiable. This means source code cannot contain unverifiable structures, and should not have to be altered for testing. This might mean source code should not contain undefined behaviour, e.g. `(i++ * i++)` in C.
3) The influences identified were organised and categorised in three areas: language features and tooling; traceability; and verification. The OOT&RT supplement was considered as a fourth area. These areas and criteria are summarised in Table I.

B. Language Features and Tooling

How does the language prevent the introduction, or support the detection, of errors?

Errors may be prevented by *language features*, and detected by *static analysis tools*. Tools that check compliance with source code standards such as MISRA C [13] are one example; tools such as Infer[1] that perform more complex, intra-procedural analysis are another. By detecting constructs yielding undefined behaviour, these tools help ensure that source code is verifiable.

Tool qualification is relevant when tools (e.g. static code analyser, test doubles) are deployed in projects. Tools that analyse, but do not change, the source code may fail to detect errors but they cannot introduce errors. Such tools require less qualification evidence than tools that can introduce errors into the airborne code.

Although important, floating point accuracy issues are governed by the execution platform and the form of equations, rather than language choice. Language agnostic tools like Herbie [14] can help reformulate equations for greater accuracy.

Some language features may cause significant changes to execution time. For example, garbage collection that occurs at unpredictable times, making it difficult to bound the *worst-case execution time* (although automated garbage collection can avoid programmer-induced issues, like failing to free memory when it is no longer needed, or attempting to use memory that has been freed).

The use of *dissimilar, redundant components* within the software development life cycle is comparatively rare. A desire, or need, to use such components would favour programming languages with a suitably diverse tool ecosystem.

How does the language complicate, or significantly enable, partitioning?

Many aspects of partitioning are unrelated to the choice of programming language, for example, being covered by the Operating System (OS) or hardware. There are, however, at least two areas where partitioning may influence the choice of programming language. First, language-based extensions might allow specific types of protection to be implemented by hardware. The Capability Hardware Enhanced RISC Instructions that can be included in C code [15] are a prominent example. Second, some languages might be designed specifically to be embedded within a partitioned sandbox. Lua [16], a scripting language used by Nmap [17], and widely used in games and other applications is a notable example. The sandbox itself needs to be implemented in a programming language, the implications of which need to be considered.

[1] https://fbinfer.com/

TABLE I
AREAS AND CRITERIA THAT MAY INFLUENCE CHOICE OF PROGRAMMING LANGUAGE, WITH CORRESPONDING PARAGRAPHS AND OBJECTIVES FROM RTCA DO-178C AND (WHERE STATED) RTCA DO-332.

Area	Criteria	Paragraphs	Objectives
Language Features and Tooling	How does the language prevent the introduction, or support the detection, of errors? Includes language features, static analysis tools, tool qualification, worst-case execution time and use of dissimilar, redundant components.	4.4, 4.4.1, 4.5, 5.3.2, 6.3.4, 11.8	A-1:3, A-1:4, A-1:5, A-2:6, A-5:3, A-5:4, A-5:6
	How does the language complicate, or significantly enable, partitioning? Includes user-modifiable software, deactivated code and parameter data items.	5.2.3, 5.2.4, 6.6	A-2:4, A-4:13, A-5:9
Traceability	How does the language complicate, or significantly enable, bi-directional traceability between low-level software requirements, source code and test cases? May include (where relevant) traceability to high-level software requirements.	5.5, 6.3.4	A-2:4, A-5:5
	How does the language complicate, or significantly enable, traceability between source and object code? Includes correspondence between source code and object code, as well as analysis of the outputs of the software-software integration process.	4.4.2, 6.3.5	A-1:3, A-5:7, A-7:9
Verification	How does the language complicate, or significantly enable, testing of executable object code? Includes typical errors, normal testing, robustness testing, software-software integration and software-hardware integration.	6.4.2, 6.4.2.1, 6.4.2.2, 6.4.3	A-6:1, A-6:2, A-6:3, A-6:4, A-6:5
	How does the language change the way that test coverage is measured or achieved? Includes structural coverage, as well as data coupling and control coupling.	6.4.4.2	A-7:5, A-7:6, A-7:7, A-7:8
OOT&RT Supplement	How does the language include OOT&RT features? Includes objects and subclasses, inheritance, polymorphism, class-based type conversions, software exception handling, dynamic memory management and virtualisation.	RTCA DO-332: OO.1.6.1.1, OO.1.6.2	RTCA DO-332: OO.A-7:OO_10, OO.A-7:OO_11

Partitioning is also relevant for *user-modifiable software*. In this context, the wider system needs to be protected against the possible effects of the modified software. This protection includes memory integrity and over-consumption of shared resources.

End user adaptability can also be achieved via *parameter data items*. Like partitioning, many of the considerations associated with parameter data items are language agnostic. However, there may be cases where the language offers strong protections against incorrect data being loaded, e.g. Domain Specific Languages (DSLs), where language semantics may only allow properly structured data to be loaded (rather than relying on hand-crafted parsing in a generic language like C, for example).

Parameter data items are often used to select different types of behaviour, which use different code sequences. In such cases, appropriate protection against unintended execution of the non-selected or *deactivated code* is required.

C. Traceability

How does the language complicate, or significantly enable, bi-directional traceability between low-level software requirements, source code and test cases?

RTCA DO-178C emphasises the importance of *bi-directional traceability* from high-level software requirements, through design to low-level software requirements, which can be refined to code. In addition, test cases are expected to be traceable to requirements.

Programming language considerations are most notable between low-level requirements, source code and unit tests. Automating traceability of source code to unit tests might be particularly valuable, e.g. to prioritise regression tests, so developers can get rapid feedback on the effects of a change.

A simple form of traceability could be implemented via a combination of specially formatted source code comments and scripts. DSLs, with appropriate Integrated Development Environment (IDE) support, could make this more formal[2]. For

[2]mbeddr is one example: http://mbeddr.com/.

some languages, wider tool integration could extend automated traceability to high-level software requirements.

How does the language complicate, or significantly enable, traceability between source and object code?

The history of programming is one of increasing levels of abstraction, enabling greater productivity and wider participation. However, abstraction also widens the gap between what programmers create (and review) and the instructions executed on hardware. We need confidence this gap does not introduce any additional safety risks. For example, when working with languages like C, this entails understanding tools like compilers, linkers, and loaders.

Checking the *correspondence between source code and object code* is a key issue, particularly given that tools may insert additional features in the object code. Such features need to be identified and verified. Correspondence checking confirms the output from the compiler. Confirming tool output is one part of allowing a tool to be used without explicit tool qualification. An alternative would be the use of a qualified compiler, but in our experience this is comparatively rare.

Checks are also required on linker output. These should ensure there are no memory overlaps or missing software components. More generally, there should be checks on the outputs of the *software-software* integration process.

D. Verification

How does the language complicate, or significantly enable, testing of executable object code?

Requirements-based testing is emphasised in RTCA DO-178C; test cases should consider error sources inherent in the software development process. Some error sources may be *typical errors*, including those associated with particular features of the chosen programming language, e.g. memory leaks in C programs. Such issues can also be identified by testing with language agnostic tools, e.g. Valgrind[3].

RTCA DO-178C includes *normal range test cases*; these consider how software should respond to normal (i.e. expected) inputs and conditions. These test cases often use the notion of equivalence classes, where test results for one input in the class can reasonably be extended across the entire class.

RTCA DO-178C also includes *robustness test cases*; these consider how software should respond to abnormal (i.e. unexpected) inputs and conditions. These include aspects that may be dependent on programming language. For example, iterators may prevent errors associated with out-of-range loop counts.

Testing of executable object code also includes *software-software integration*. This includes topics like parameter passing, initialisation, data corruption (especially of global variables), incorrect sequencing and inadequate numerical resolution. Language-based features and tools can influence these topics.

The final aspect of executable object code testing is *software-hardware integration* on the target computer. It is concerned with, for example, memory management, stack overflow and resource contention. These may be affected by the choice of programming language as well as compiler options. For example, the `_FORTIFY_SOURCE` macro, which can be used with the GNU Compiler Collection to introduce lightweight checks to detect buffer overflows.

How does the language change the way that test coverage is measured or achieved?

The requirements-based testing approach of RTCA DO-178C provides confidence that software exhibits the desired behaviour. Measuring structural coverage helps demonstrate that the software does not contain any unwanted behaviour (i.e. behaviour that cannot be traced to a requirement).

RTCA DO-178C's objectives include a series of increasingly demanding levels of *structural coverage* on source or object code, beginning with statement coverage, moving through branch and on to Modified Condition / Decision Coverage (MC/DC). Whilst these levels of coverage are applicable to most programming languages, language features may have an effect. For example, if features favour the creation of many code branches or if they require certain behaviour in multi-clause conditionals (e.g. always requiring a default case).

The objectives also include testing the *data coupling* between code components. This is concerned with the route from when a piece of data (used by two components) is set (in one component) to the point when it is used (in another component). As with structural coverage, this is likely to be mostly language agnostic. However, it should be facilitated by languages with clear scoping rules that simplify type conversion.

Control coupling between code components is also considered. This is mostly concerned with the call-graph of the software; interrupt and exception handling are also relevant. Like data coupling, this is largely language agnostic, but may be complicated by language features obfuscating the call structure, such as function pointers, and multiple levels of object inheritance.

E. OOT&RT Supplement

How does the language incorporate OOT&RT features?

RTCA DO-332[4] considers the use of classes to define types, and the ability to create new classes via subclassing as distinguishing features of Object-Oriented Technology (OOT). Consequently, if a programming language uses *objects and subclasses* then we judge it to fall within the OOT portion of RTCA DO-332.

Programming languages also fall within the scope of RTCA DO-332 if they use "*related techniques*": inheritance, polymorphism (either parametric or ad-hoc, via overloading), class-based type conversions, software exception handling, dynamic memory management and virtualisation. While the specific details vary, using these techniques should not result in unpredictable, or poorly understood, behaviour, nor should it result in data corruption or an inconsistent program state.

[3]https://valgrind.org/

[4]Para. OO.1.6.1.1

IV. Working Example: Rust

To illustrate these criteria, we use them to evaluate the applicability of the Rust programming language [18]. Rust is a high-performance systems programming language, suitable for embedded systems. The language has been specifically designed with features that enforce memory safety and thread safety [19]. As such, many of the typical errors associated with comparative languages like C are prevented by design, often through Rust's ownership model. Rust also has features that encourage robust programming.

A. Language Features and Tooling

How does the language prevent the introduction, or support the detection, of errors?

Rust's *language features* enforce memory safety and thread safety. These are tightly linked to the concepts of ownership and borrowing. The Rust compiler ensures that: each variable has an owner; there can only be one owner at a time; and when the owner goes out of scope, the memory associated with the variable is freed. These rules allow the compiler to safely schedule dynamic memory management. They mean Rust programs are largely "memory safe" and, because there is only one owner, "thread safe". To allow for situations when C functions need to be used, and raw memory allocations are necessary, the `unsafe` keyword can scope potentially dangerous code.

Compared to memory safety, protecting against arithmetic overflow is more complicated. In debug mode, checks are made, with code terminating if overflow occurs. However, these checks are not included in release mode, but they can be implemented, if desired.

Rust's design means that many issues detected by C-based *static analysis tools* cannot occur. Nevertheless, tools are being developed to address other language-related issues. For example, Prusti [20] checks whether a program will produce an error condition that results in termination; Kani [21] checks properties associated with dynamic dispatch.

The main tools associated with Rust are the compiler (`rustc`), and the associated build system and package manager (`cargo`). The output from these tools can be inspected, so *tool qualification* need not apply. However, if checks cannot be made then the open source, relatively fast-changing nature of these tools may complicate the process of tool qualification. There are also plans to develop a qualified Rust compiler tool chain[5].

Rust's ownership and scoping rules make dynamic memory management both safe and predictable. Whilst the latter aspect removes a possible difficulty when estimating *worst-case execution time*, this remains a challenging endeavour.

For practical purposes, there is currently only a single Rust compiler and package manager available. This means that, should they be required, *dissimilar, redundant components* cannot be used. Based on our practical experience, we believe the likelihood of this being required is very low. It is also likely that additional compilers will become available.

How does the language complicate, or significantly enable, partitioning? Rust's approach to "ownership" means programs are both memory safe and thread safe. They are memory safe in the sense that typical errors like use after free and double free cannot occur. They are thread safe in the sense that only a single thread can own a shared resource, which prevents typical errors like data races (Figure 1). These properties enable partitioning.

Fig. 1. Transfer of ownership.

Consider the following function definitions[a] that implement a simple channel for communicating between threads:

```
fn send<T: Send>(chan: &Channel<T>, t: T);
fn recv<T: Send>(chan: &Channel<T>) -> T;
```

Suppose, also, there is a utility function `print_vec` that prints a vector element-by-element. Now, consider the following code snippet:

```
1 let mut vec = Vec::new();
2 vec.push(4);
3 send(&chan, vec);
4 print_vec(&vec);
```

A vector is created and populated in one thread (Lines 1, 2) before being sent down the channel to another thread (Line 3). Following Rust's ownership rules, ownership of the vector is transferred to the receiving thread. Hence, the compiler will report an error (Line 4) when the original thread tries to use a variable it does not own.

[a]Based on: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html.

Despite these benefits, the systems programming nature of Rust means that it is designed to access many low-level system features. As such, it is unlikely that *user modifiable software* will be written directly in Rust.

As a systems programming language, Rust has features that assist in reading data files. It also has a series of protections that make it easier to use *parameter data items*. For example, attempts to write beyond the end of an array will be detected. Rust also has a special `Result type` (Figure 2) that encapsulates a value and a potential error. This allows for the easy detection of errors, e.g. when translating a string to an integer. Such features further simplify the use of parameter data items.

Different configurations can include or exclude code from a compiled Rust object. If the "inclusion decision" can be made at compile time, this provides a powerful way of managing *deactivated code*. If the decision is made at run time then Rust's memory safety and thread safety features should be helpful.

[5]https://ferrous-systems.com/ferrocene/

Fig. 2. Use of the `Result` type.

Consider the following snippet:

```
fn main() {
  let x:&str = "12.34";
  let y:f32 = match x.parse() {
    Ok(v) => v,
    Err(why) => panic!("{}", why)
  };
  println!("{}", y);
}
```

If the string can be parsed then `Ok()` extracts the relevant value. If it cannot then `Err()` extracts the associated error message. Natural use of the `Result` type in Rust makes this type of error-trapped string (and consequently, file) handling commonplace. It avoids the need to explicitly check a return value, as is often the case with C.

B. Traceability

How does the language complicate, or significantly enable, bi-directional traceability between low-level software requirements, source code and test cases?

With regards to *bi-directional traceability*, Rust includes a `[#test]` attribute, which identifies tests. However, this does not provide any special functionality for linking tests with specific requirements. As with most languages, this could be achieved by local implementations that rely on specially formatted comments.

Does the language complicate, or significantly enable, traceability between source and object code?

Some memory safety and thread safety properties can be checked at compile time, while others can only be detected at run time. To perform the run time checks, the Rust compiler introduces a significant amount of additional object code that, in RTCA DO-178C terms, is not directly traceable to the source code (Figure 3). This complicates the process of establishing *correspondence between source code and object code*. Despite these complications, with care, it is possible to establish this correspondence.

In Rust, *software-software integration* is achieved using `cargo`. This has been designed to be both a build system and a package manager. In the former role, it compiles code; in the latter role, it manages external dependencies and makes distributable packages. This multi-purpose nature makes it slightly harder to gain confidence in the tool than would be the case for a conventional compiler.

Integrating software written in different languages (e.g. C and Rust) is complicated by Rust's default behaviour. For example, Rust mangles function names and does not guarantee `struct` properties. Attributes can be used to obtain C-compatible behaviour (specifically, `#[no_mangle]` and `#[repr(C)]`).

Fig. 3. Source to object code correspondence.

Consider the following code[a]. For brevity, the body of the `calc_sum` function is not listed:

```
fn calc_sum(in_arr:&[i32]) -> i32 { ... }

fn main() {
  let my_arr:[i32; 100] = [1; 100];
  println!("{}", calc_sum(&my_arr));
}
```

In this case, the object code associated with the call to the `calc_sum` function is as follows:

```
movl $100, %esi
callq playground::calc_sum
```

The value 100 (i.e. the size of the array) is loaded into the `esi` register before the call is made. This allows bounds checking to be implemented within `calc_sum`.

[a]This code was created using the Rust Playground: https://play.rust-lang.org/.

C. Verification

How does the language complicate, or significantly enable, testing of executable object code?

Rust's design means that *typical errors* like "use after free" can be detected at compile time. Others can be reliably detected at run time. This significantly simplifies testing of the executable code.

As with many programming languages, conducting *normal range test cases* is easy with Rust. The language has no significant peculiarities that would make it difficult to establish test equivalence classes.

Rust's design also makes it relatively easy to conduct *robustness test cases*. For example, the idiomatic way of looping through an array is via an iterator, rather than an indexed variable. This means the loop implicitly knows the array bounds, thus preventing an out-of-range access.

In terms of arithmetic overflow, Rust's behaviour complicates the conduct of robustness tests. In particular, overflow protection is included in debug builds (with overflows causing panics) but excluded from release builds. It can be included in the latter by either using a compiler option by passing `-C debug-assertions=on` to the Rust compiler, or setting `debug-assertions` field of a Cargo profile, or by protecting individual operations, e.g. `sum = sum.checked_add(i).unwrap();`. This range of options, and the resulting potentially different behaviour between debug and release builds needs to be carefully managed.

Verifying the *software-software integration* is simplified by the typical errors Rust prevents by design. However, as noted in Section IV-B, cases where Rust and C are used alongside each other require special care.

Rust's intended use as a systems programming language, together with its intended use for embedded systems mean that *software-hardware integration* testing should be relatively straightforward.

How does the language change the way that test coverage is measured or achieved?

Rust includes common ways of managing control flow (e.g. `if`, `while` and `for`). These constructs do not change the way that *structural coverage* is measured or achieved. However, the `match` statement differs from the C `switch` statement in some important ways. First, each arm of the `match` can include lists or ranges of options. Achieving MC/DC of a list-based arm requires testing of each list item[6]. Second, `match` arms can have guards, which are separate logical conditions that also have to be true in order for the associated code to be executed (Figure 4). Achieving MC/DC of a guard based arm would require the guard and the arm-defining condition each to separately influence the decision as to whether the code is executed or not.

Fig. 4. MC/DC and `match`.

Consider the following code snippet:

```
let z = 121;
let letters = match z
  // ((z%2) == 1) && (z >= 0) && (z <= 9)
  0..=9 if z%2 == 1 => "1-digit, odd",
  0..=9 => "1-digit, even",
  _ if z%3 == 0 => "n-digit, mult. of 3",
  _ => "not interesting"
;
println!("", letters);
```

This example shows how the `(z%2 == 1)` guard alters the equivalent C code for the arm conditions. This example also illustrates the use of an underscore as a default case.

In each situation, it is relatively easy to determine what cases need to be tested in order to achieve MC/DC. However, to the best of our knowledge, there is no tool-based support either to assist in the definition of such tests, or to monitor this type of coverage.

Data coupling is significantly simplified by Rust's notion of variable ownership. This provides the compiler with explicit knowledge of where variables (i.e. data) can be changed. Also, the Rust compiler detects, and provides warnings for, unused data items.

When considering *control coupling*, Rust, like C, allows functions to be called indirectly via function pointers. These pointers may refer to a function whose identity is not known at compile time. This could make it difficult to determine whether control coupling has been achieved. Rust also has the notion of dynamic traits, which allow dynamic dispatch. These represent another potential complication for control coupling, although their use could be prevented by a coding standard (similar to the use of MISRA C).

[6]The same would be required if the arm (or case) was expressed as a disjunction (i.e. or) of the items in the list, which would be the natural way of expressing the construct in C.

D. OOT&RT Supplement

How does the language incorporate OOT&RT features?

Rust supports some features of object-oriented programming. For example, it combines data and behaviour in the same type. It also uses encapsulation to hide implementation details. However, Rust does not use inheritance or the notion of subclasses. More specifically, it does not include both *objects and subclasses* and, as such, does not fall under the OOT part of RTCA DO-332.

The following paragraphs consider *related features* in the context of RTCA DO-332.

Instead of using inheritance, Rust uses traits to achieve polymorphism (i.e. code that can work with multiple types). A trait (e.g. draw, for a user interface component) is defined as a specific entity. This trait then has to be defined explicitly for each relevant type. Compile time errors occur if the trait is used on a type for which it has not been defined. Hence, there is no confusion about which code will be executed. The issues discussed in RTCA DO-332 (e.g. ambiguity, confusing traceability and inappropriate use of generic data) are also avoided.

Rust's lack of subclassing removes many potential issues associated with class-based type conversions. For example, supertypes cannot be downcast to a subtype, removing the possibility of inheritance-based data corruption.

In terms of software exception handling, from a RTCA DO-332 perspective, the main concern is that incorrect handling (or non-handling) of exceptions may leave a program in an inconsistent state. As noted in Section IV-A, Rust's `Result` type supports the handling of recoverable errors. The only other approach to exception handling is the `panic!` macro which immediately stops execution, due to an unrecoverable error. This combination of approaches means that exceptions (i.e. recoverable and unrecoverable errors) cannot leave a program in an inconsistent state.

Rust makes extensive use of dynamic memory management, but its ownership model means that this is demonstrably safe.

Within the context of RTCA DO-332, Rust offers no direct support to virtualisation.

V. Conclusion

Based on the contents of RTCA DO-178C and its OOT&RT supplement (i.e. RTCA DO-332) we have devised a set of evaluation criteria that may influence the choice of programming language for safety-critical software in the air domain. These criteria relate to language features and tooling, traceability, verification, and OOT&RT.

In presenting the evaluation criteria, our work makes three contributions.

First, we proposed the use of evaluation criteria for informing the selection of programming languages, principally in the

air domain, but also for other domains that invoke safety-critical software. As Section II indicates, the use of such criteria for quickly evaluating the usability of programming languages in general is not new. However, we believe our work represents a valuable attempt at deriving such criteria, in the context of modern programming languages that may be suitable for use in safety-critical domains.

Second, we analysed RTCA DO-178C and relevant supplements to devise suitable evaluation criteria for safety-critical air domain software. The criteria were *not* designed to provide a compelling argument that any particular programming language is safe. Instead, they highlight key aspects of a programming language that can help evaluators understand the rationale for using (or not using) it in the air domain.

Finally, we applied our evaluation criteria to the Rust programming language. Our results indicate that, despite detailed and interesting aspects that require attention, no significant barriers appear to prevent the use of Rust for safety-critical software in the air domain.

No language is perfectly suited for the development of safety-critical software in the air domain. For example, when compared with C, neither has features that enable traceability between requirements, source code and test cases. Likewise, neither provides protection against the full range of numeric issues that can occur. These challenges may be overcome through a combination of procedural means (e.g. requirements standards to facilitate traceability and design reviews to confirm algorithm accuracy) and appropriate testing (e.g. robustness testing that specifically targets potential numerical issues).

Compliance with RTCA DO-178C is just one factor that needs to be weighed when choosing a language (or languages) for a safety-critical development. For example, language stability, developer availability and developer productivity are all relevant. Exploring these factors is left for future work.

ACKNOWLEDGEMENTS

This document is an overview of UK MOD sponsored research. The contents of this document should not be interpreted as representing the views of the UK MOD, nor should it be assumed that they reflect any current or future UK MOD policy.

REFERENCES

[1] RTCA, "Software Considerations in Airborne Systems and Equipment Certification," RTCA, Tech. Rep. DO-178C, December 2011.

[2] S. Van Trees, "A few kind words on aviation standards and development assurance," in *2018 Integrated Communications, Navigation, Surveillance Conference (ICNS)*, 2018, pp. 1–12.

[3] G. Kudrjavets, J. Thomas, A. Kumar, N. Nagappan, and A. Rastogi, "When malloc () never returns null–reliability as an illusion," *arXiv preprint arXiv:2208.08484*, 2022.

[4] T. J. Jennings, "The benefits of using SPARK for high-assurance software," *Ada User Journal*, 2012.

[5] G. Ferreira, "Software certification in practice: How are standards being applied?" in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 100–102.

[6] RTCA, "Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A," RTCA, Tech. Rep. DO-332, December 2011.

[7] ——, "Formal Methods Supplement to DO-178C and DO-278A," RTCA, Tech. Rep. DO-333, December 2011.

[8] J. Marques and A. M. da Cunha, "A set of requirements for certification of airborne military software," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, 2019, pp. 1–7.

[9] A. Sarkis, J. Marques, and L. A. V. Dias, "Recommendations for the usage of design models in aviation software," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2021, pp. 56–63.

[10] C. Sadowski and S. Kurniawan, "Heuristic evaluation of programming language features: Two parallel programming case studies," in *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '11. Association for Computing Machinery, 2011, p. 9–14.

[11] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '90. Association for Computing Machinery, 1990, p. 249–256.

[12] T. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[13] MISRA, "Guidelines for the Use of the C Language in Critical Systems," MISRA, Tech. Rep. MISRA C:2012, March 2013.

[14] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 1–11, 2015.

[15] A. Richardson, "Complete Spatial Safety for C and C++ using CHERI Capabilities," Ph.D. dissertation, University of Cambridge, 2019.

[16] R. Ierusalimschy, *Programming in Lua*. Roberto Ierusalimschy, 2006.

[17] G. Lyon, *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Nmap Project, 2009.

[18] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. Association for Computing Machinery, 2014, p. 103–104.

[19] R. Jung, "Understanding and evolving the rust programming language," Ph.D. dissertation, 2020.

[20] V. Astrauskas, A. Bílỳ, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust," in *NASA Formal Methods Symposium*. Springer, 2022, pp. 88–108.

[21] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, "Verifying dynamic trait objects in rust," *Proceedings of the ICSE-SEIP (2022, to appear)*, 2022.