Trade Study of Scripting Languages for Avionics Systems

Arnau Prat
German Aerospace Center (DLR)
Institute of Software Technology
38108 Braunschweig, Germany
Arnau.PratISala@dlr.de

Christoph Torens
German Aerospace Center (DLR)
Institute of Flight Systems
38108 Braunschweig, Germany
Christoph.Torens@dlr.de

Benjamin Carrick ∗
German Aerospace Center (DLR)
Institute of Software Technology
38108 Braunschweig, Germany
Benjamin.Carrick@posteo.de

Florian-Michael Adolf ∗†
German Aerospace Center (DLR)
Institute of Flight Systems
38108 Braunschweig, Germany
Florian@autonomy79.auro

Frank Giljohann
Rockwell Collins Deutschland GmbH,
Collins Aerospace | An RTX Business
69123 Heidelberg, Germany
Frank.Giljohann@collins.com

Daniel Lüdtke
German Aerospace Center (DLR)
Institute of Software Technology
38108 Braunschweig, Germany
Daniel.Luedtke@dlr.de

Abstract*—Developing software for avionics systems presents a multifaceted challenge, both in terms of complexity and cost. One approach to address these challenges while enabling customization and minimizing requalification efforts is through the utilization of the Lua scripting language. Lua empowers a diverse workforce, allowing for potential customization by end-users without necessitating full system re-verification, thereby saving valuable time and resources. Thus, we explore the use of scripting languages, which have different properties than compiled languages typically used for avionics systems. Scripting languages can reduce complexity by increasing the abstraction layer and supporting software development in the production environment, leading to faster iteration times. However, avionics systems have strict requirements for real-time behavior, memory constraints, and safety-critical certification, which scripting languages are not designed to fulfill. In this paper, we discuss the pros and cons of using scripting languages for safety-critical systems. We present a comparison of different off-the-shelf scripting engines, based on the following criteria: real-time performance, memory usage, portability, reliability, and certifiability to be used in safety-critical systems such as avionics systems. The result is a recommendation of the most suitable language to use in the avionics domain based on different criteria. We recommend Lua as the most suitable language for use in the avionics domain, as it has the best fit to the given requirements. We also suggest modifications that would need to be made to the engine and the scripts to make them certifiable with respect to the DO-178C standards and the additional tool qualification supplements. These include changes to the garbage collector, the type system, and the coding rules and verification methods.

***Index Terms*—scripting languages, interpreted languages, avionics systems, safety-critical systems, real-time requirements, memory management, software certification, software engineering, lua scripting language**

∗ Former organization German Aerospace Center (DLR)
† Author is currently affiliated with autonomy79.aero

I. Introduction

Scripting languages are getting increasingly popular for a multitude of programming tasks. These languages are easy to learn and powerful, especially for small tasks. One popular example for scripting languages is the Python language. However, today even complex programs can be written using these languages. Scripting languages are the tool of choice for data science projects, but also artificial intelligence applications, most specifically machine learning [1]–[3]. A domain where scripting languages are not typically used is in avionics systems. These systems have not only real-time requirements, but are also safety-critical and must therefore comply to strict regulation and standards of the aviation domain. The relevant standards for the safety-critical development of software are DO-178C Software Considerations in Airborne Systems and Equipment Certification [4], as well as DO-332 Object Oriented Technology and Related Techniques Supplement [5]. It is a common practice to reduce the features of a programming language, such as C, to a subset of features that can be shown to be safe [6], [7]. Therefore the use of a highly extensible scripting language seems counterintuitive at first glance. Despite these high requirements, the use of scripting languages could have multiple benefits, also for the development of safety-critical systems [8].

Scripting languages are a commonly used tool to create extensible software. For software which has to be highly customizable, using a scripting engine for application-level functionalities can be a big contribution to the cost-optimization of the development process on the long-run. As scripting languages usually provide a high level of abstraction customers can develop their own customizations of the base software without a heavyweight toolchain and in-depth knowledge of the complete system, like memory management. The current development processes for safety-critical software in the avionics area require a huge overhead. Several tools are required by the end user to fulfill the verification requirements for the developed software. Costs for development of even small features or extensions can get very high when such use of proprietary tools and libraries is required. This problem increases with third party products, such as certified, proprietary real-time operating systems and libraries.

Another problem is the extensibility of this software due to the architecture. Usually the software for safety-critical systems is designed in a monolithic way, which means that the full source code or at least a part of it is needed to extend the features. This means everyone who wants to write even small extensions needs developer access to libraries and the intellectual property of the vendor. The use of scripting engines could ease the process of adding new features to the software as well as restrict the access to the source code of the core system. The software could provide a public interface to modify certain functions of the software which is accessible from the scripting language. A layer between the script and the core software will execute these scripts and manage the access to the different interfaces. This way a developer can write modifications for the base software without the need for the toolchains and source code of the core software. However, the requirements of utilizing a scripting language for real-time avionics systems have to be considered, including aspects of software verification and certification.

The goal of this paper is to discuss and evaluate different implementations of scripting languages for the use in safety-critical system. The paper presents commonly used scripting languages and selects some of them for a detailed evaluation. The results show a recommendation on the most suitable language and necessary modifications to establish a certifiable scripting language to be used in the presented use case scenario. The rest of the paper is organized as follows: Section 2 gives an overview of related work. Section 3 presents the use-case used in this comparison. Section 4 discusses the main characteristics of scripting languages compared to compiled ones. Section 5 presents the list of candidate scripting languages of the evaluation. Section 6 discusses the methodology used to compare the different languages. Section 7 presents the results of the comparison. Section 8 discuss the results and presents possible improvements to make the selected scripting engine real-time capable. Finally, Section 9 gives the conclusions and an outlook to future work.

II. Related work

This section presents related work that describes challenges and future research for avionics platforms and describes the need for adopting cutting-edge technology and development methods. One specific step towards these challenges is the use of high-level languages, such as domain-specific languages [9]. Although scripting languages are not specifically mentioned, the motivation of using high-level languages is very similar to the approach that is presented in this work. A general overview of the characteristics of scripting languages is given in [10] and [11]. Scripting languages like Python are used increasingly in commercial projects. An example of this is a requirement analysis tool that uses AI to generate traceability links between requirements of safety-critical software [12]. Python has also been used for testing safety-critical systems, such as avionics systems [13]. In addition, there are several examples that motivates the use of Python to increase the overall productivity [8]. However, there is no major example in which Python has been used for safety-critical software.

Writing safety-critical software is a specialized field with requirements specific for this task. The verification and certification aspects are the most important factor for choosing a language. How to choose a programming language for use in safety-critical systems is detailed in [14]. The recommendations for languages in 1991 included ISO Pascal, Ada, Modula-2, CORAL 66, structured assembly languages, and C. However, with C being the only exception, developers would not typically use these languages. While some work argues for specific languages, such as Pearl [15], others see three options: Ada, Java and C/C++ [16]. As mentioned before, there are specific certification requirements for the avionics domain. [17] gives an overview of software certification for real-time safety-critical systems. [18] focus on memory management of safety-critical hard real-time systems. [19] details on the requirements of programming languages for safety-critical systems. As a result, these certification aspects have to be analyzed for scripting languages.

In practice, the C/C++ programming language plays a huge role, especially with the guidelines from MISRA [7]. The reason for this is its easy access to the hardware, low memory requirements as well as efficient run-time performance as noted in [20]. C/C++ remain the most popular languages for developing software in real-time embedded systems since they are close to the hardware. Also, it must not be underestimated that these languages are well understood: there is a history of uses in the safety-critical domains, there exist safe compilers, etc. A general comparison between scripting languages and system programming languages is presented in [21]. Although their big gain in popularity during the last few years [22], scripting languages still remain a minority in safety-critical applications. However there are a few examples of them expanding different domains that use them for these kind of applications, from robotics to aerospace.

Several examples can be found for Java. In [16] the challenges of certifying the interpreted language Java are compared with the challenges for certifying the compiled languages ADA and C/C++. Several papers have been published on how Java could be modified to be used for safety-critical systems. In [23] and [24] the use of Java for safety-critical applications is described. [25] describes garbage collection for safety-critical Java. [26] details how validating Java could be implemented for safety-critical applications. Another example [27] is a toolkit for testing safety-critical cockpit display systems with Java. [28] describe the use of a real-time Java VM for robotics. A real-time Java virtual machine was successfully used for avionics in the DARPA Program Composition for Embedded System [29], [30]. Some examples are starting to be found for other scripting languages. [31] analyse MicroPython to be used in robotics application. The industry is also following this trend and in 2019 VxWorks announced support for Python [32]. Although it does not support real-time requirements, this could be a first step on seeing more safety-critical applications run on top of a scripting engine.

If we look at the space industry, some examples can be found of missions that use scripting engines as part of its flight software. The first example is found in the James Webb Space Telescope (JWST) mission [33], which uses JavaScript to schedule operations on-board. A second example is found in the Euclid mission [34], which uses MicroPython to also schedule different control procedures on the satellite itself. This trend is not only seen in big missions but also in CubeSats ones where scripting engines are used in the flight software [35]. Although one may think that in the previous examples, the scripting engine is used only for control and sequencing tasks and not for instruments or data handling (none of the key functionalities run on a virtual machine or interpeter), it is a first step toward using them also in this cases. This agrees with the effort lead by the European Space Agency (ESA) of porting MicroPython to LEON platforms and starting with some qualifications activities of the VM [36]. Although initially it would only be to used for non-critical functions. Another interesting field is robotics, where real-time requirements usually need to be met. An example of a scripting engine in this field, is ORCOS [37]. This provides a framework for building real-time robotic applications and it is based on the Lua scripting language.

Despite the previously presented examples, in the literature, no related work on language comparison regarding scripting engines to use in critical systems has been found by the authors. Although multiple examples can be found of trade studies for compiled languages for real-time systems [38] or more general reviews of programming languages for games [39]. As such, this work is the first systematic analysis and comparison for the use of scripting languages in safety-critical context to the best of the authors knowledge.

III. Use-cases

There are multiple use-cases in which scripting languages could be used in the avionics domain. This section details some of them and the motivation for it. The technical context used in this paper consists of the configuration of cockpit displays according to the ARINC 661 standard. Additionally, the scope of the software framework contains the reception of data from an ARINC 429 data bus, processing of this data and presentation of it on the cockpit display. The application programming interfaces (API) to the data bus and the display are provided by a library which is already available from the vendor. The scripting will have access to these APIs through a wrapper and shall connect the data acquisition with the presentation on the display. In terms of certification and tooling, the scripts shall be viewed independently from the rest of the software. The interpreter which executes the scripts will be part of the main software package and be certified with it, whereas the scripts will be certified independently. This enables the development of scripts for the flight system without the need for the tooling package of the rest of the software. The targeted platform for the interpreter consists of an RTOS. The work in this paper has the context of two business use-cases in which scripting languages could be used.

A. Business Use Case 1 - OEM Updates

When customers buy products from the vendor they usually get the software pre-installed on that product. Customizing this software can get complicated for the customer as often the intellectual property remains at the vendor and modifying the software needs at least some parts of the source code. Even if the source code of the software is available, the customer still needs to certify the modified software and needs appropriate tools to do this. The use of scripting languages shall ease the process to modify the software for both sides. This way, the customer has the possibility to rapidly prototype customizations without the need for an extensive toolchain. Therefore the use of the scripting shall need a minimum of tools for the rapid prototyping.

B. Business Use Case 2 - Small Business & Offset

Many contracts and customers demand a small business or in-country offset for developing the customizations of the vendor products. On one hand using a scripting language can ease the development process itself and avoid costly toolchains. The use of a scripting language can therefore support the involvement of these small companies. On the other hand, when providing a method to develop the scripts against a public interface, it might not be necessary to provide access to the source code of the core system. This way, the intellectual properties of the vendor do not have to be published to a third party.

IV. Characteristics of Scripting Languages

The use of scripting languages can bring several advantages compared to compiled low-level ones, especially in terms of flexibility and usability. On the other side, these languages can have some disadvantages with regard to run-time errors and indeterministic execution time, which may prevent them for being used in safety-critical systems, such as avionics systems. Table I presents some of the advantages and disadvantages in relation to some of the typical characteristics of scripting languages. As the table shows, an "off-the-shelf" scripting language may not be used in avionics systems, as the implementation will not fulfill the requirements for safety-critical systems. To fulfill the requirements, the implementation of the scripting engine may have to be changed. Additionally, changes to the syntax of the language itself, such as dynamic typing, may also have to be done. Considering this, modifying an existing language to meet the mentioned requirements may be complicated and the effort may increase with its complexity. On the other side, complex languages often contain constructs that ease the development process by providing convenient structures and methods. Thus, there is a trade-off between complexity and adaptability of the language.

Language Feature	Advantages	Disadvantages
Garbage Collection	No manual memory management	Indeterministic Run-time
	No pointers needed	Higher memory consumption
	Better support for data structures of variable size	Program can run out of memory
Weak Typing System	Faster development	Additional programming errors
	Automatic intended type conversions	Automatic unintended type conversions
Run-time Loading	Faster testing and debugging	Decreased run-time performance
		Higher memory consumption

TABLE I: Advantages and disadvantages of scripting engines

The concepts of scripting engines diverge significantly from those of compiled languages. In contrast to scripting languages, which are designed to be as flexible as possible, compiled languages are focused on static run-time behavior and memory consumption. This flexibility is typically achieved through a high degree of dynamic language elements, whereby the code is evaluated at run-time instead of compile time. This section presents two aspects of scripting languages that differ from the concept of compiled languages and are of particular importance for the use in safety-critical systems: This section will examine the interpretation techniques and memory management employed by scripting languages.

A. Interpretation Techniques

Direct interpretation represents the most basic form of code interpretation. In this approach, the scripting engine reads the source code, evaluates the statements, and executes the methods in accordance with the aforementioned interpretation. This approach is typically slow due to the necessity of evaluating each statement before execution, which results in significant overhead. To address this issue, a method known as just-in-time (JIT) compilation was developed, which represents a compromise between the dynamic behavior of an interpreter and the performance of compiled machine code. Rather than merely interpreting the source code, the JIT compiler will translate code segments into machine code at run-time and store these compiled segments in memory. In the event that the program is required to execute the same code segment a second time, the script engine will now utilize the compiled code segment in lieu of translating it once more. This approach enables the execution performance to be achieved that is similar to that of compiled languages for segments that have already been translated.

In the context of real-time systems, this advantage will not result in the same performance gain. The rationale for this discrepancy lies in the substantial divergence between the worst-case execution time, during which the code segment must undergo compilation, and the best-case execution time, during which previously compiled segments will be executed. In the context of real-time systems, it is imperative to consider the worst-case execution time, as previously compiled code may be deleted depending on the caching algorithms. The worst-case execution time will not differ significantly from that of a directly interpreted implementation. However, the JIT compiling technique requires a considerable amount of memory for the storage of the compiled code, which limits the advantages of this technique.

A second aspect of the interpretation techniques is the representation of the scripting language. For the purpose of writing scripts, a textual representation is the optimal choice, as it is important for human readability. Many implementations will directly use the textual representation and parse it in the interpreter. While it is relatively simple for a human to read the textual code, the computer must parse the code, which results in rather slow performance and high memory consumption. To enhance performance and reduce memory requirements, modern interpreters typically employ the concept of byte-code. This is a distinct instruction set that translates all expressions of the scripting language into a binary representation that can be readily read by an interpreter. With this binary representation, the code will exhibit a markedly reduced memory footprint when loaded into the interpreter and a higher execution performance compared to a textual parser. However, this requires an additional compilation step in order to generate the aforementioned byte code.

B. Memory Management

Scripting languages memory management is primarily influenced by the object management and dynamic typing featured in most implementations. Scripting languages rely heavily on dynamic memory allocation, as they are designed to evaluate the majority of their code at run-time. This implies that the size of an object is typically unknown until it is created, and therefore the memory layout cannot be determined at compile-time. Furthermore, in contrast to static typing, the assignment of a variable to another variable can result in the creation of a third variable if the type of the original variable is altered as a consequence of the assignment.

Because memory is finite and objects must be created and released frequently, it is necessary for scripting engines to implement some form of memory management. Typically, scripting engines provide automatic memory management methods to streamline the development process. The rationale behind this approach is that it allows application programmers to focus on solving a given problem without having to worry about low-level details such as allocating and freeing memory. To achieve this automatic memory management, so-called garbage collectors are implemented to check if objects in memory still have references on them and therefore are reachable by the code. If that is no longer the case, the object is destroyed and the memory it had occupied freed by the garbage collector. According to DO-178C, garbage collection is a preferred method for memory management.

In the context of real-time systems, most implementations of garbage collectors are not viable due to their non-deterministic nature with respect to the invocation and duration of a single collection run. The most common approach to garbage collection is the "tracing garbage collection" algorithm, which records references to objects within a given data structure and monitors the traces of references between these objects. During the collection process, execution of the actual application code must be paused. This introduces an element of unpredictability into normal run-time operations, since the garbage collector is typically invoked when there is insufficient memory to allocate a new object, or when a certain amount of memory has been allocated. One possible solution to this problem is to explicitly reserve a time slice for garbage collection, thereby linking the execution of the garbage collector to a timed event rather than depending on the memory situation. This so-called "incremental" garbage collection can be employed in real-time systems, where the collection run will be terminated when the allotted time has elapsed.

Regardless of the method employed (pausing or incrementation), sweeping unreferenced memory will inevitably result in memory fragmentation. In order to mitigate this, garbage collectors will implement a "copy-pass," whereby objects will be relocated to other memory sections in order to densify the memory usage and create larger consecutive areas of memory to allocate large objects.

An alternative to garbage collection is the Automatic Reference Counting (ARC) method. This method is simpler than garbage collection, as it will add meta information to an object which counts the currently used references to that object. As soon as this counter will be set to zero, the object will be freed. In comparison to the "mark-and-sweep" method, this method does not need a "mark" run and will directly sweep objects that are no longer reachable. However, as with the garbage collection method, this method is also unsuitable for real-time systems. The sweeping of objects depends on the program flow, rather than a timed condition, which introduces unpredictable delays into the code execution. Although these delays are relatively brief, the algorithm is unable to fulfill the needs of a real-time system.

The third method of automatic memory management is the so-called "Variable Storage" implemented in the Espruino engine. The variables are stored in blocks of fixed size (usually 16 or 32 bytes), which can be linked if an object does not fit into a single block. These objects are sorted into two lists, which contain either free or used blocks. As in the ARC method, the blocks contain a reference counter as metadata, facilitating the sweeping of unused memory without the need for reference tracing of the objects. The garbage collector can then efficiently clean up the unused blocks by iterating over the used block list and moving blocks with a zero reference counter to the free list. The significant advantage of this method is that the sweeping can be called in a timed event, eliminating the need for indeterministic delays. Furthermore, as the memory is comprised of identical-sized blocks that can be linked in an arbitrary manner, this method is not susceptible to the problem of memory fragmentation.

V. Scripting language candidates

This section will present a selection of languages that have been identified for further discussion within the context of this paper. The languages that are considered in this paper have been selected according to a set of criteria derived from the technical and business use cases previously presented. The first criterion is the popularity of the language. The scripting language in its standard implementation shall be widely known to a large community and used for many applications. The use of a well-known scripting language will facilitate the entry of new developers into the field, as they will already be familiar with the syntax and only need to learn about the specific characteristics of the implementation. In order to assess popularity, the IEEE Spectrum Programming Language Ranking 2023 has been used [22]. The second criterion is the availability of languages for different real-time operating systems. Therefore, implementations that do not depend on a specific operating system must be chosen.

A. Candidates

This section will provide an overview of all languages and their implementations that are under consideration for this trade study. Three of these candidates will then be selected for further discussion in detail.

1) Python - CPython: Python is a widespread scripting language with a rich feature set. The current iteration of the language and its reference implementation is Python 3. Python is a scripting language with a strong typesystem and an extensive standard library which will ease application development. The CPython implementation is available as a standalone interpreter for POSIX and Windows Systems and can be used as a plugin interpreter in other applications.

2) Python - MicroPython: MicroPython is an embedded implementation of the Python language which maintains a high grade of compatibility to the Python 3 API. The main focus of the MicroPython implementation is on the compatibility to the standard Python implementation while running directly on a micrcontroller and achieving a small memory footprint. Many libraries, besides the standard library, that are available for the Python 3 reference implementation (CPython) will also work on MicroPython. MicroPython comes with a reference platform which consists of an ARM Cortex-M microcontroller and additional ports are available for the ESP8266 and ESP32 microcontrollers.

3) Lua - Reference Implementation: Lua is a very minimalistic Scripting language that is designed to be used as a plug-in scripting language. As a design concept the whole interpreter consists of a single library which makes it easy to integrate it into any application. The interpreter is written in ANSI C and therefore compatible to all platforms that have a compiler for this platform.

4) Lua - eLua: eLua is a alternative implementation of the Lua language. It is designed as a stand-alone implementation without the need for an underlying operating system. Therefore this engine implements features as file I/O and access to several microcontroller peripherals. Code written in eLua that is not depending on platform-specific methods is compatible to the Lua Reference Implementation.

5) JavaScript - Espruino: JavaScript is a scripting language that is mostly used for client-side scripting of webbrowsers. Espruino is implementing an JavaScript interpreter for embedded microcontrollers. Unlike other implementations presented here, the Espruino platform includes a special method for automatic memory management called Variable Store. Espruino comes with several implementations for ARM Cortex-M and ESP8266 microcontrollers.

6) Ruby - CRuby: Ruby is a popular scripting language which is developed as an all-purpose programming language and mostly used in server-side web applications. One major feature of Ruby is the consistent object orientation that considers even primitive data types like integers and floating point numbers as objects. The standard CRuby implementation is available for Windows and POSIX platforms.

7) Perl - Reference Implementation: Perl is a language primarily designed for processing text files but can be used as a multi-purpose language. An extensive amount of libraries is available and Perl has very good support for regular expressions. The reference implementation is available as a single library and supports POSIX Operating systems as well as Windows.

B. Pre-selection of Candidates

While all of the candidates would generally be suitable for the use case, a preliminary evaluation can help to identify some as less suitable. The initial decision can be made between the microcontroller implementations and the standard implementation of Lua and Python. The standard implementation is more suitable for our use case as they are designed to be a plugin language rather than eLua and MicroPython, which are designed as standalone interpreters. A standalone interpreter for embedded systems is typically equipped with pre-built extensions for the use with microcontrollers, such as a hardware abstraction layer. In the context of avionics systems, however, the language is typically employed on top of an RTOS, rendering these extensions unnecessary. Consequently, the standalone version must be modified to function as a plugin container. Another candidate that can be excluded at this stage is CRuby. This language is highly reliant on object orientation, treating all the elements as objects, which produces more overhead, performance issues, and more memory consumption, particularly in the case of a scripting language. Furthermore, having object orientation throughout all language elements will add additional effort for the porting of the engine to DO-178C. As more rules apply to object-oriented languages according to the DO-178C standard, the scripts will be more difficult to certify. The last candidate to be excluded is Perl. Although Perl is a widely used language, its design decisions are contrary to the requirements of safety-critical systems. Perl is in generally considered difficult to read due to the high amount of "syntactic sugar" that enables multiple ways to write the same expression. After eliminating these unsuitable candidates, the remaining three will be discussed in detail in the following sections.

VI. Methodology

The candidates are evaluated according to different criteria which are related to the concepts discussed in the previous section. The following criteria have been selected and the methodology of the evaluation will be described in detail.

A. Language Complexity

Language complexity is a key factor in determining the adaptability of a language and the effort required to train developers, and is therefore important in the selection process. For this study, language complexity is measured by the number of keywords in a language. The more keywords a language has, the more concepts are implemented and the more complex the implementation has to be to cover these concepts. In the context of the use case, a less complex language is preferred because it reduces the effort to adapt the languages to meet the safety-critical requirements.

B. Implementation Complexity

This criterion evaluates the complexity of the codebase for the language implementations. This is especially important for adapting the implementation to the requirements described in the use case. The metric for this is the code size of the implementations, with fewer lines of code and fewer modules the better.

C. Interpretation Technique

The method used to interpret the scripting code is important when evaluating runtime behavior and memory consumption. However, there is not much difference between the techniques presented (direct interpretation and JIT compilation) when determining the worst-case execution time. However, for the proposed use case, a minimum set of tools should be used to enable a rapid prototyping process. Therefore, it is preferable to translate the script from its textual representation rather than from an intermediate byte-code representation.

D. Automatic Memory Management

As mentioned earlier, the algorithm used for the garbage collector has a large impact on the real-time performance. The requirement for automatic memory management is predictable behavior in terms of when the collection is called and how long a run will take. This criterion has a moderate impact on the selection process, since all algorithms can potentially be modified to meet real-time constraints.

E. Typing Concept

Since all languages are dynamically typed, this criterion considers both strong and weak typing concepts. Since weak typing is a potential source of programming errors, the selected language should implement a strong typing concept. This criterion is of medium relevance because the problems associated with weak typing and implicit type casting can be limited by applying coding rules that restrict certain expressions in the language. However, applying extensive coding rules may limit the usability of the language, and users of the standard implementation will have to learn how to comply with these rules. In addition to usability, coding rules will require tools such as static code analyzers to help enforce the rules. These tools will also grow in complexity as the rules become more extensive. Therefore, it is not impossible to use weakly typed languages, but more strongly typed languages are preferred.

F. Portability

The portability criterion evaluates the effort required to port the given implementation to any given operating system. In general, all proposed candidates are compatible with the POSIX interface. Therefore, this criterion analyzes the dependencies needed to build a minimal scripting environment. These dependencies include standard libraries and dependencies on operating system libraries.

G. Integrated Language Features

Many scripting languages provide a high level of abstraction from the underlying platform to the application developer. These abstractions implement commonly used data structures such as lists, queues, stacks, etc., and operating system features such as threading. These abstractions are achieved through a standard library that must be ported to each platform. In general, these features are desirable because they are ready to use and the developer can focus more on solving the actual problem. However, in the context of the use cases, many features are not needed (such as threading) and will significantly increase the effort to port the language to another platform. Therefore, the relevance of this criterion is low.

H. Community Support

All languages and their implementations proposed for modification are open source projects. This criterion takes into account the state of the community (documentation, development activity, etc.). Since modifications or at least extensions need to be implemented, an active community and good documentation would be helpful. This criterion is considered low because modifying the scripting engine results in a fork of the original implementation and is not maintained by the original project community.

VII. Comparison of Candidates

Table II shows a comparison of the different approaches with respect to the criteria stated in the previous section.

Considering the sum of all criteria, the Lua language is the most suitable solution for our use case. Although some parts of the language and the implementation need to be modified, the concept of Lua as a very minimal language is very advantageous. In terms of complexity, Lua is the least complex when it comes to the language itself and the implementation of the standard scripting engine. Having fewer keywords than Python or JavaScript will make it easier to implement code checking tools. The smaller code base will make it easier to modify the scripting engine. Another benefit of Lua's minimal and extensible design is that a basic setup requires only a single library written in ANSI C code. This increases the portability of the scripting language, and the only task to be performed when porting the language to a new platform is the implementation of extensions to access platform-specific features. Compared to Espruino, the core features do not need to be modified, as they do not depend on any hardware. Also, Lua does not have a standard library like Python that provides basic language runtime functions that would otherwise have to be ported. The interpretation technique of Lua is a direct textual interpretation, as it is for Espruino. This technique is more suitable for use in real-time systems, as it behaves more deterministically and the just-in-time compilation used in MicroPython does not bring any performance advantage. The mark-and-sweep garbage collector is not the best choice compared to the variable store implemented in the Espruino engine. Although the garbage collector is not deterministic when running at arbitrary times, the implementations used in Lua and Python can be disabled and controlled manually. This allows the explicit invocation of garbage collection runs at specific time slots, making it possible to use these implementations with little modification. Another important criterion for evaluating the candidates is the typing system. Python is the best candidate because it supports a stronger type system than the other two candidates. All other languages use a weak typing system that makes extensive use of implicit type conversions. This disadvantage can be mitigated by code analysis tools that check for implicit type casts and other restricted language constructs. While a

Criteria	**Relevance**	**Lua**	**Python**	**Espruino**
Language complexity	High	Low (21)	Medium (33)	High (64)
Implementation complexity	High	Minimal	High	Medium
Portability	High	High (only little changes and extensions)	Medium (standard library has to be adapted)	Medium (some core features need modifications)
Interpretation technique	High	Direct interpretaton (Textual)	Just-in-time compilation (Textual)	Direct interpretation (Textual)
Automatic memory management	Medium	Mark & Sweep GC	Mark & Sweep GC	Variable Store
Typing concept	Medium	Weak (no explicit casts)	Strong	Weak
Integrated language features	Low	Few	Many	Few
Community support	Low	High	High	Medium

TABLE II: Comparison of scripting language candidates

very good choice in all other categories, Lua's typing system is rather disadvantageous for use in safety-critical systems. Lua only supports implicit casts, which is a potential source of programming errors, and tools must be provided to ensure consistent conversion of variables. Another minor drawback of Lua is related to the minimal concept of the language. Since it contains only the minimal set of language features, some convenient features of Python such as libraries of commonly used containers, threading support, etc. are missing. There are several libraries that implement some of these features, but the engine itself does not contain a standard library. When using Lua, it may be necessary to create a library of commonly used data structures to support the rapid prototyping use case.

VIII. Discussion

The comparison of the three candidates showed that the concept of the Lua languages best fits our requirements, although there are some major drawbacks, especially in the context of the typing system. Therefore, it is recommended to use the standard Lua implementation as a reference for further development. However, the standard implementation is not directly usable and needs to be modified to meet the requirements of safety-critical systems. To enable the use of common software engineering tools (such as unit test frameworks, code coverage analyzers, etc.), modifications to the Lua language itself should be as limited as possible. In this way, it will be possible to use currently available tools for the standard implementation with little or no modification. In order to make the engine and scripts certifiable for the DO-178C standard, several modifications have to be made.

A. Modifications to the Garbage Collector

As mentioned earlier, the automatic, arbitrary calls to the garbage collector will break the real-time requirements of a safety-critical system. To meet real-time requirements for script execution, the automatic garbage collector must be turned off and called manually at an appropriate time. Lua already provides functions to manually control the garbage collector from within the scripts. These functions need to be externalized so that they can be executed by a dedicated task of the operating system. Also, the language-internal functions for controlling the garbage collector need to be removed to ensure that it is not inadvertently enabled. This especially affects worst-case execution time and memory consumption.

B. Modifications to the Type System

The main part of the change is Lua's typing system. Lua only supports implicit casts, which is not generally forbidden, but the DO-178C standard requires that these type conversions are "safe and the implications understood". Therefore, a mechanism for tracing these casts and ensuring safe casts must be established, where developers can add the information of the intended conversion. This can be achieved with extensions to the Lua language through so-called annotations, which tag variables and conversions with the intention of that conversion. These annotations do not need to be parsed by the interpreter, but can be used in code verification tools. These tools can then be used to check the implicit conversions against the intended conversions given by the annotations and warn the developer if there is a mismatch.

C. Code Verification

For use in production, certified aircraft systems, coding rules must be established that cover all applicable DO-178C requirements. To verify these rules, code verification tools are needed to support the development and certification process. These code verification tools can be implemented in a model-based manner, where the script itself is mapped to a model. Depending on the set of rules to be established, annotations can be added to the language to provide additional information to the model. This model can then be processed by various model verification methods to check compliance with the coding rules. Any tools used for the purpose of verifying specific DO-178C objectives must comply with the Standard for Software Tool Qualification Considerations Supplement DO-330.

IX. Conclusion and Outlook

Lua is the most suitable candidate according to the characteristics used for comparison. This trade study showed that Lua is a suitable candidate for a scripting language in the area of safety-critical systems. Since this scripting engine cannot be used directly, some modifications have to be implemented, mainly in the context of automatic memory management. While the engine that will execute the Lua code can be certified by these modifications, the features of the Lua language itself have to be limited. Therefore, the next step in making Lua scripts certifiable is to create coding rules that map to the applicable requirements of the DO-178C standard. These coding rules will ensure that the scripts comply with the requirements. To support the enforcement of these rules, code verification tools must be created. These tools will need to comply with the additional tool qualification supplement to DO-178C: DO-330 Software Tool Qualification Considerations. In future work, we will take the results of this trade study and work on a design approach that will lead to a certifiable scripting language based on the Lua implementation presented in this work.

X. Acknowledgment

This work is partly supported by a Luftfahrtforschungsprogramm (LuFo) grant with funds provided by the Federal Ministry for Economic Affairs and Energy (BMWi) with reference 20H1507A / MFDCS. We would like to thank Mr Jason Myren for inspiring the work carried out in this paper.

XI. Bibliography

[1] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *Information*, vol. 11, no. 4, p. 193, 2020.

[2] A. Nandy and M. Biswas, *Reinforcement Learning: With Open AI, TensorFlow and Keras Using Python*. Apress, 2017.

[3] A. C. Müller and S. Guido, *Introduction to machine learning with Python: a guide for data scientists*. "O'Reilly Media, Inc.", 2016.

[4] Radio Technical Commission for Aeronautics, *DO-178C/ED-12C Software Considerations in Airborne Systems and Equipment Certification*. Washington, D.C.: RTCA, 2011.

[5] ——, *DO-332/ED-217 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A*. Washington, D.C.: RTCA, 2011.

[6] L. Hatton, "Safer language subsets: an overview and a case history, MISRA C," *Information and Software Technology*, vol. 46, no. 7, pp. 465–472, 2004.

[7] R. Bagnara, A. Bagnara, and P. M. Hill, "The MISRA C coding standard and its role in the development and analysis of safety-and security-critical embedded software," in *International Static Analysis Symposium*. Springer, 2018, pp. 5–23.

[8] N. Valot, P. Vidal, and L. Fabre, "Increase avionics software development productivity using Micropython and Jupyter notebooks," in *ERTS 2018*, ser. 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, Jan. 2018.

[9] B. Annighoefer, M. Halle, A. Schweiger, M. Reich, C. Watkins, S. H. VanderLeest, S. Harwarth, and P. Deiber, "Challenges and Ways Forward for Avionics Platforms and their Development in 2019," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, Sep. 2019, pp. 1–10, ISSN: 2155-7209.

[10] A. Kanavin, "An overview of scripting languages," *Lappeenranta University of Technology. Finland*, p. 10, 2002.

[11] L. Prechelt, "Are scripting languages any good? a validation of perl, python, rexx, and tcl against C, C++, and Java." *Adv. Comput.*, vol. 57, pp. 205–270, 2003.

[12] A. Patel, R. Cerveny, T. Shibamoto, and J. Murphy, "Collins aerospace artificially intelligent requirement analysis tool."

[13] L. Li, Y. Cai, D. Qiao, X. Zhang, Z. Wang, T. Qi, and H. Ji, "Research and Design of Automatic Test Language for Control System Software," in *Signal and Information Processing, Networking and Computers*, ser. Lecture Notes in Electrical Engineering, Y. Wang, L. Xu, Y. Yan, and J. Zou, Eds. Singapore: Springer, 2021, pp. 552–559.

[14] W. Cullyer, S. Goodenough, and B. A. Wichmann, "The choice of computer languages for use in safety-critical systems," *Software Engineering Journal*, vol. 6, no. 2, pp. 51–58, 1991.

[15] W. A. Halang and J. Zalewski, "Programming languages for use in safety-related applications," *Annual Reviews in Control*, vol. 27, no. 1, pp. 39–45, 2003.

[16] A. Kornecki and J. Zalewski, "Certification of software for real-time safety-critical systems: state of the art," *Innovations in Systems and Software Engineering*, vol. 5, no. 2, pp. 149–161, Jun. 2009.

[17] R. Dove, W. Schindel, and R. W. Hartney, "Case study: Agile hardware/firmware/software product line engineering at Rockwell Collins," in *2017 Annual IEEE International Systems Conference (SysCon)*, Apr. 2017, pp. 1–8, iSSN: 2472-9647.

[18] A. Malik, H. Park, M. Nadeem, and Z. Salcic, "Memory management of safety-critical hard real-time systems designed in SystemJ," *Microprocessors and Microsystems*, vol. 64, pp. 101–119, Feb. 2019.

[19] I. Schagaev and T. Kaegi-Trachsel, "Programming Language for Safety Critical Systems," in *Software Design for Resilient Computer Systems*, I. Schagaev and T. Kaegi-Trachsel, Eds. Cham: Springer International Publishing, 2016, pp. 159–182.

[20] M. Nahas and A. Maaita, "Choosing appropriate programming language to implement software for real-time resource-constrained embedded systems," *Embedded Systems-Theory and Design Methodology*, 2012.

[21] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *Computer*, vol. 31, no. 3, pp. 23–30, 1998.

[22] S. Cass, "The top programming languages 2023," https://spectrum.ieee.org/the-top-programming-languages-2023, 06 2023.

[23] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek, "Java for safety-critical applications," 2009.

[24] A. Walter and J. J. Hunt, *Java in Safety Critical Systems*, 2009.

[25] M. Schoeberl and J. Vitek, "Garbage collection for safety critical Java," in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, ser. JTRES '07. Vienna, Austria: Association for Computing Machinery, Sep. 2007, pp. 85–93.

[26] J.-M. Dautelle, "Validating java (tm) for safety-critical applications," p. 6812, 2005.

[27] H. Sartaj, M. Z. Iqbal, and M. U. Khan, "Cdst: A toolkit for testing cockpit display systems of avionics," *arXiv preprint arXiv:2001.07869*, 2020.

[28] S. G. Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov, "Using real-time java for industrial robot control," in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, 2007, pp. 104–110.

[29] J. Baker, A. Cunei, C. Flack, F. Pizlo, M. Prochazka, J. Vitek, A. Armbruster, E. Pla, and D. Holmes, "A Real-time Java Virtual Machine for Avionics - An Experience Report," in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, Apr. 2006, pp. 384–396, iSSN: 1545-3421.

[30] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek, "A real-time Java virtual machine with applications in avionics," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 1, pp. 5:1–5:49, Dec. 2007.

[31] R. Mahmoud Almostafa, R. Nabila, B. Hicham, and R. Mounir, "Python in real time application for mobile robot," *Smart Application and Data Analysis for Smart Cities (SADASC'18)*, 2018.

[32] M. Chabroux, "Vxworks now has a pet snake," https://blogs.windriver.com/wind_river_blog/2019/09/vxworks-7-now-has-a-pet-snake, 09 2019.

[33] V. Balzano and D. Zak, "Event-driven James Webb space telescope operations using on-board javascripts," in *Advanced Software and Control for Astronomy*, vol. 6274. International Society for Optics and Photonics, 2006, p. 62740A.

[34] F. Pasian, J. Hoar, M. Sauvage, C. Dabin, M. Poncet, and O. Mansutti, "Science ground segment for the ESA Euclid mission," in *Software and Cyberinfrastructure for Astronomy II*, vol. 8451. SPIE, 2012, pp. 21–32.

[35] S. Plamauer and M. Langer, "Evaluation of micropython as application layer programming language on cubesats," in *ARCS 2017; 30th International Conference on Architecture of Computing Systems*. VDE, 2017, pp. 1–9.

[36] D. George, D. Sanchez, and T. Jorge, "Porting of micropython to leon platforms," *Data Systems in Aerospace*, 2016.

[37] M. Klotzbücher, P. Soetens, and H. Bruyninckx, "Orocos rtt-lua: an execution environment for building real-time robotic domain specific languages," in *International Workshop on Dynamic languages for Robotic and Sensors*, vol. 8, 2010.

[38] J. E. Cooling, "Languages for the programming of real-time embedded systems a survey and comparison," *Microprocessors and Microsystems*, vol. 20, no. 2, pp. 67–77, 1996.

[39] M. D. Masood and A. Wijdan, "Comparison of programming languages in game development," *preprint*, 2020.