Scripting Engine Design Approach 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

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

***Abstract*—Avionics software development is inherently challenging due to its complexity, high cost, and high demands for real-time performance and safety-critical certification. Traditional use of compiled languages results in longer development cycles and limited customization due to complex certification processes. The paper suggests using scripting languages, particularly Lua, to address these issues, potentially saving time and resources while maintaining compliance with safety standards. It identifies necessary modifications to the Lua run-time for DO-178C certification: altering the garbage collector for real-time behavior and adjusting the typing system for type safety. The paper also presents a technical implementation of the Lua scripting engine as a proof-of-concept, evaluating its stability and real-time capability compared to a C implementation. The results demonstrate the feasibility of the proposed implementation. The paper will also look at how such approach could be certified.**

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

I. Introduction

Developing software for avionics systems is a demanding task, characterized by intricate technical requirements, substantial development costs, and the necessity to meet rigorous real-time and safety certification standards. While compiled languages have long been the industry norm for delivering reliable and deterministic behavior, their use often entails long development cycles and adaptability constrains among other problems. An alternative approach is to use scripting languages to address these challenges. By leveraging scripting languages ability to increase abstraction and support rapid iteration, development complexity can be reduced and end-user customization can be enabled without the need for full system re-verification or repeated certification. This approach has the potential to save significant time and resources, while still maintaining compliance with critical safety standards and certification requirements.

Previous research showed that the Lua scripting language is the most suitable starting point for a scripting language that is certifiable by the DO-178C standard [1]. However it has been pointed out that the current reference implementation of Lua violates some of the requirements given by the standard. This paper will show the implications and the modifications needed to create a compliant version of the Lua language and runtime. These are two fold: modifications to the engine to make it real-time capable (this mostly concerns the garbage collector) and changes to the typing system for type safety. Afterwards, an implementation for the given use-case is implemented and results are shown. The paper will also look at how this approach could be certified according to the DO-178C up to criticality DAL-A.

The goal of this paper is to develop and present a proof of concept implementation of such an engine. The remainder of the paper is organized as follows: Section 2 provides an overview of related work, Section 3 presents the use-case to demonstrate the approach, Section 4 shows the implementation details to achieve a compliant engine, Section 5 presents the certification considerations in order to certify the given approach, Section 6 shows the results of the implemented proof of concept. Finally, Section 7 gives the conclusions and outlook to future work.

II. Related Work

Several studies highlight the growing interesting in adopting high-level languages in avionics and other safety-critical domains [2]. Although scripting languages are not always explicitly mentioned, the motivation for using them is to improve productivity and flexibility [3]. For example, Python is increasingly used in safety-related contexts for tasks such as requirements analysis [4] and testing [5], though still not fully in critical systems [6].

Language selection in safety-critical domains emphasizes robust certification and strict verification. Traditional guidance favors Ada, C/C++ and others compiled languages with predictable behavior and highly mature tool-chains [7]. C/C++ remains dominant due to its performance and hardware control [8]. Java has also had efforts to adapt it for real-time and safety-critical contexts through custom Virtual Machines and tightly restricted subsets [9].

Scripting engines are beginning to emerge in space and robotics applications. For example, JavaScript and MicroPython have been used for different space missions [10], [11]. In robotics, Lua has been employed within different real-time frameworks [12]. Despite these developments, there are no relevant comprehensive studies using scripting languages for safety-critical applications.

III. Use-Case

The use-case application used in this paper consists of the configuration of cockpit displays according to the ARINC 661 standard. The software framework integrates the reception of data from an ARINC 429 data bus, processes the information, and visualizes it on cockpit screens. APIs for interfacing with the data bus and display are provided by a vendor-supplied library. A wrapper enables scripting to utilize these APIs, bridging data acquisition with the display rendering. For certification and tooling requirements, scripts are managed independently from the primary software. While the interpreter running these scripts is part of the main software package and undergoes certification alongside it, the scripts themselves are certified separately. This strategy facilitates the creation of flight system scripts without dependence on the tooling package used for the broader software. The interpreter is tailored for execution on a Real-Time Operating System (RTOS) platform.

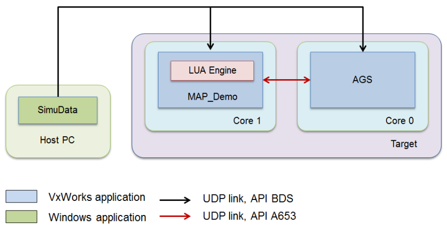


Fig. 1: Use-case system diagram

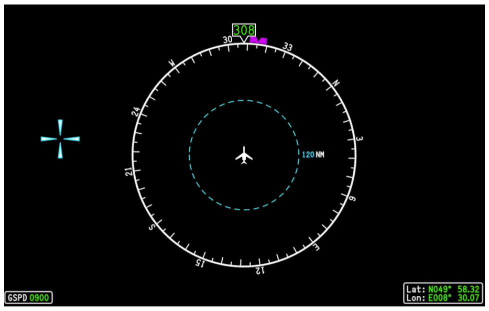


Fig. 2: Use-case application screenshot

The application is composed of three main modules as shown in Figure 1: SimuData, MAP_demo and AGS (Avionics Graphics Software). Of which two of them run on the target device running VxWorks and one run on an external Windows computer. The two module executing on VxWorks are the MAP_demo and the AGS. The MAP_demo acts as the main controller/intelligence of the applications and is where the LUA interpreter is located. The module decides what it needs to be displayed depending on the user input. The AGS acts as an interface between the display and the MAP_demo, providing it with simple functions to write to the display and informing it if certain events happend (e.g. the cursor is pressed). The AGS also handles some basic interactions like highlight a widget when the cursor is on top of it. The SimuData module on the other hand is located on an external computer and is the tool used by the user to interact and input data to the system.

The use-case intends to be simple but at the same time to test core functionalities. Figure 2 shows a screenshot of the application. This can be divided in three parts. Namely: map mode, debug mode and cursor widget. Each of them tries to test different functionalities.

- Map mode: Demonstrates an example of basic widget manipulation, such as modifying the value of a label or rotating a container. For the proposed use-case the following values will be set/modified: heading, latitude, longitude and gspd. The value of these will depend on the mapMode value. If it is "0" the value of these will be updated automatically. If it is different, those parameters will be set to the values sent by the SimuData tool on the external Windows computer.
- Debug mode: Shows an example of toggling a layer based on the value of debugMode. When set to a value different from "0", a debug grid is displayed.
- Cursor widget: Illustrates an example of toggling a layer (fuel remaining) based on a cursor event. Visibility of the "fuel left" box is toggled if the cursor was pressed inside the circle defined around the AC symbol.

```
while(true)
do
   -- Update basic widgets

   -- Update debug mode

   -- Update cursor widget

   -- Sleep for time t
end
```

Fig. 3: Use-case template Lua script

Figure 3 shows a template of how the Lua script for the use-case application can look like. The script consists of a main loop that runs indefinitely with a fixed sleep interval at the end of each iteration. This time determines the rate in which the different widgets are updated.

In each iteration of the loop the three parts of the application described before are updated. First, the map mode is updated. In order to do so, the value of the mapMode is obtained. Depending on it, the value of the head, lat, long and gsps will come from different sources. Once this is done, the widgets are updated accordingly. Second, the debug mode is updated. Depending on the value of debugMode (true or false) the debug grid will be displayed or not. Finally, the cursor widget is updated. Similarly to the debugMode, depending on the value of flagCursor, the fuel box will be shown or not. The flagCursor is toggled whenever the cursor is pressed inside the AC area. This parameter is set by the MAP_demo when an event is triggered by the AGS.

IV. Integration

The Lua language and runtime is designed specifically as a plug-in scripting engine for desktop applications. As a result to this design decision, the language and runtime itself is very minimal and easy to integrate into different environments. While being a minimal language, Lua supports extensions to the base language, which enables additional features, that might be beneficial for the certification process.

As most scripting languages, Lua has a very dynamic typing model. Variables inside the runtime will frequently change their type inherently and their size. This feature might be convenient from a programmers perspective but implies a complex memory management and issues regarding type safety as demanded by the DO-178 standard.

Beside the certification aspects of the scripts running inside the Lua runtime, the Lua runtime code itself has to be certified. As the Lua runtime is written in ANSI-C, the usual certification methods will apply to that code. Especially the modifications of the runtime have to be incorporated into this certification. These changes will be discussed later. For now, the focus is on a detailed view of the technical implications.

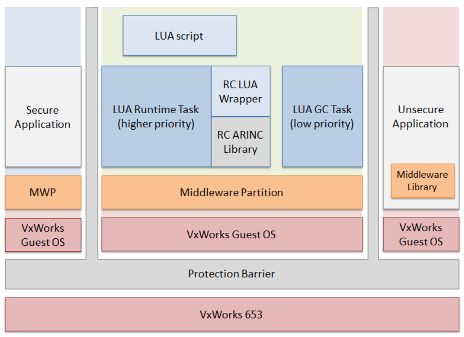


Fig. 4: Use-case architecture view

Figure 4 gives an overview of the setup of the different parts of the Lua runtime and the integration into the given operating system environment.

Integrating the Lua interpreter into a given environment is quite straightforward. However, it is necessary to consider necessary real-time modifications to the scripting language. Therefore some minor modifications of the scripting implementations have to be made.

The main idea to establish a real-time capable scripting engine is to detach the garbage collector from the rest of the engine. An essential concept of the operating system is the time and space partitioning according to the ARINC-653 standard to prevent error propagation through the software system. Therefore both parts (the scripting language and the garbage collector) will be split up into two different tasks. There, each Lua script needs its own OS task to define the timing of the script and enable the schedulability analysis. As the garbage collector needs access to the objects allocated by the interpreter, both parts of the engine have to share the same memory partition. The follow-up section about the time-triggered garbage collection concept will emphasize more on the modifications to the garbage collector.

The operating system used in the demonstrator hardware is VxWorks 653. A special version of VxWorks designed to be compliant with the ARINC 653 standard that implements an additional layer which allows having different partitions isolated both in time and space separated by a protection barrier. In this case, the two applications mentioned before running on VxWorks (MAP_Demo and AGS) will run in their own partitions. Looking deeper at the MAP_Demo, this contains the modified real-time capable LUA scripting engine, the vendor ARINC libraries, the wrappers for the interpreter to access these libraries as well as other low level functionalities.

To enable access to the ARINC-661 and ARINC-429 interfaces, the scripts need to call the driver functions provided by the vendor library. Therefore a Lua wrapper to this C library is needed and has to be integrated into the Lua runtime environment. Lua supports easy mapping of C functions to corresponding Lua functions by design, making wrappers easier to implement.

A. Callbacks

Since Lua has no direct support for callbacks, they had to be emulated. Our engine emulates them by using a queue that is filled with callback data by the C support code. Each element of the queue represents a callback and has the following members: id (int), name (string), value (float), active (bool) and register (bool). The queue has a size of 10 elements.

A Lua hook is set to run every 1 Lua bytecode instructions, being this the most low-level operations executed by the Lua engine. This hook function acts as the Lua callback handler. Each time the hook is activated, it will look for a different element of the queue in a consecutive order. If the element is registered and it is active the hook will call the Lua function with the name and pass the value to it. After that it will set active to false.

B. Wrappers

In order for the Lua scripts to be able to access the ARINC 661 and other low level functionalities required by the use case, a set of wrappers had to be created. It was decided to go with the lowest level of abstraction as possible to allow for a maximum of flexibility and support applications different from the demo use-case. A total of 6 function wrappers have been created. Apart from 6 ARINC 661 functions (SET_ROTATION_ANGLE, SET_STRING, SET_VISIBLE, SET_FILL_INDEX, SET_FILL_COLOR and SET_COLOR_INDEX) it has also been implemented: set_int_handler, which registers a callback, sleep_ms, which sleeps for a certain microseconds and time_ms, which gives the current time of the system in ticks of the system.

C. Memory Allocation

Lua does all its memory allocation and deallocation through one single allocation function. By default this functions uses the standard realloc and free functions from the C standard library. However, the VxWorks certified version, which we are using as Guest OS in both of our partitions, does not provide these functions (free and realloc). Nonetheless, Lua allows the user to provide a custom allocation function while creating the Lua state thus allowing us to define our own realloc and free functions. For this prototype implementation, a simplified custom memory pool was implemented. However, this implementation does not handle defragmentation, which could be needed for more complex scripts than the one used for the use-case, as well as other advanced features. Open source alternatives have been identified such as the FreeRTOS memory pool implementation. It implements five options with different levels of complexity and it is available under MIT license. There is also the SafeRTOS option available which provides certification evidence for different safety standards such as DO-178C. However, this option is not open source.

D. Runtime View

In this section we will look at the different calls that the MAP_Demo (main) needs to make to the Lua API from start until the script get executed. This is shown in Figure 5. The steps can be summarized as follows: create a new state, return the state, open the standard libraries, register the wrappers for the ARINC 661 and other low level functionalities, set the hook for the real-time garbage collector and the callbacks, load the script file and execute the script.

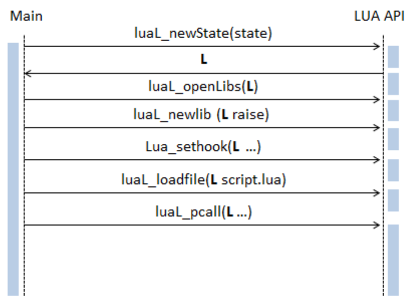


Fig. 5: Runtime view initialization of Lua engine

Once it is running, the scripting engine will run the garbage collector interleaved with the interpreter. In Lua 5.2, on which our engine is based, every time the interpreter allocates some amount of memory, the collector will run a small step. However, since we want the garbage collector to be deterministic, in our implementation it will not run when memory is allocated but instead at predefined times intervals (cycle time) and with a maximum duration (cycle length). More information can be found in the following subsections.

E. Time-Triggered Garbage Collection

The most challenging task from a technical perspective is the modification of the garbage collector to achieve a real-time behavior of the scripting engine. Implemented as per specification, the Mark-and-Sweep algorithm used by the Lua runtime will introduce unpredictable delays into the execution of scripts. The reason for this is the coupling of garbage collector runs to memory allocation calls. This means for each allocation, the garbage collector will check if enough memory is available and free unused objects if not. Using this coupling will guarantee that, if there is enough free memory available, an allocation will never fail due to floating memory (not referenced but not yet freed memory). The problem with this technique is that for non-trivial programs the memory situation is not predictable and therefore the duration of a garbage collector run can not always be predicted.

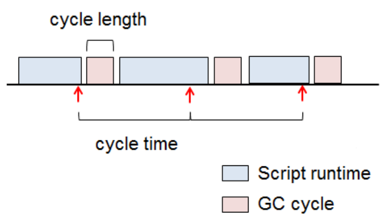


Fig. 6: Interleaving of runtime and GC cycle

The implementation of the Lua garbage collector however offers approaches to mitigate this problem. As the methods to control the garbage collector are exposed to both, the Lua language and the C library interface, it is possible to start, stop and resume the garbage collector at any time. This enables the possibility to decouple the garbage collector runs from memory allocation calls.

The OS integration has to be designed in such way that the garbage collector can be put into its own task. With this method, the garbage collector is no longer invoked by memory allocations or the state of memory, but is triggered at specific times. This makes the call predictable, and the deadline set by the task slot constrains the execution time of a garbage collector run.

The decoupling of the garbage collector runs from the allocation calls brings up a problem with the memory management. It is now possible that allocations can fail due to unreachable objects, which have not been freed yet. This might happen because the garbage collector will not be called whenever new memory might be needed. Additionally the garbage collector is limited to a certain time frame and might not have enough time to clean up the necessary amount of memory.

The work of Robertz et al. [13] addresses this problem in describing an approach of a time-triggered garbage collector. The work describes a technique to run an incremental garbage collector in a fixed-priority task. As Lua implements an incremental garbage collector since version 5.1 and VxWorks incorporates a fixed priority scheduler, this technique is suitable for the given environment. The basic concept behind the approach of Robertz et al. is the periodic call of the garbage collector. For the time frame between two calls of the garbage collector, the maximum memory consumption of the mutator (in our case the Lua interpreter application) has to be determined. This value can be asserted by static analysis of the scripts or by unit tests which reports the memory consumption of the interpreter for the test run. After the memory consumption has been determined this data will be used to calculate the cycle time of the garbage collector. The referenced paper defines the calculation for determining the cycle time T_{GC} of the garbage collector as follows:

$$T_{GC} \leq \frac{\frac{H-L_{max}}{2} - \sum_{j\in P} a_j}{\sum_{j\in P} f_j a_j} \qquad (1)$$

With the following inputs:

H	The total heap size for the Lua runtime, which can be configured when the runtime environment is initialized. This is mostly limited by the size of the memory partition.
L_{max}	The maximum amount of live (reachable) memory all of the script will take.
P	The set of all processes or in this context scripts.
a_j	The maximum amount of memory the script j can allocate per invocation.
f_j	The frequency of the invocation of script j.

After calculating the cycle time, the worst-case execution time of the garbage collector can be derived from that calculation. In general the garbage collector will have its longest execution time when all scripts acquire their maximum amount of memory and leave all objects floating (dereference them) before the next cycle. In this case the garbage collector has to clean all objects and their memory, respectively, that have been allocated in that cycle. Therefore the worst-case execution time for the garbage collector can be determined by the following equation

$$M_{max} = \sum_{j\in P} a_j i_j \,. \qquad (2)$$

Where the maximum memory for the garbage collector is the sum of all memory that can be acquired in one cycle. In detail M_{max} is the maximum amount of memory that each script j in the set of scripts P can allocate per invocation (a_j) multiplied by the maximum amount of invocations per garbage collection cycle i_j. This way the garbage collector can be integrated into the system as a normal task and the timings can be used for the schedulability analysis.

F. Script Integration Process

To integrate new or modified scripts into the system, some work has to be done to ensure the correct operations of the garbage collector. Figure 7 gives an overview of the proposed process to integrate scripts into the system.

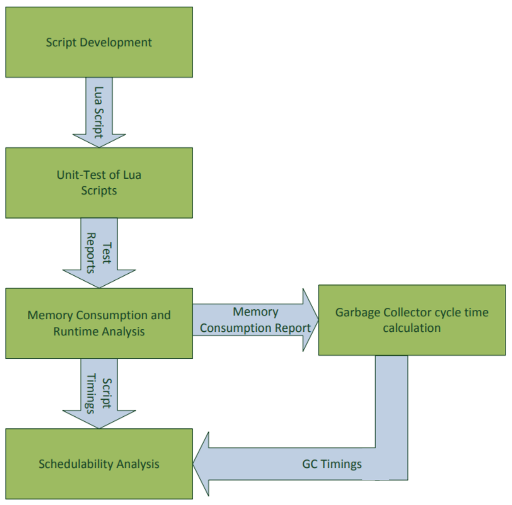


Fig. 7: Script integration process

This process needs a fixed worst-case execution time (WCET) of the Lua garbage collector. The garbage collector implementation supports this process by allowing incremental steps to be performed each run. With the amount of steps, the WCET can be calculated as well as memory that can be freed in that time. These values will than be used for the cycle time of the garbage collector. The following process step will now determine the cycle time of the garbage collector task and validate the schedulability with the given period.

The first step in the process after the script has been finished is the unit testing of this script. The unit tests will run in an environment where the garbage collector is disabled. When running the unit tests, the testing framework needs to keep track of the memory consumption of the test run and the execution time. As described in the previous sections with the information about memory usage of all scripts for a given time frame, the cycle time and the worst-case execution time of the garbage collector can be calculated. The last step uses these calculation for schedulability analysis to check if the real-time constrains can be met with the current setup.

G. Error Mitigation

"Traditional" garbage collectors which invocations are associated to the allocation of new objects can guarantee that allocation will never fail if there is enough memory available. The reason for that is, if the allocation would fail, the garbage collector will free and compact the floating memory until the allocation can succeed. The same strategy applies to the time-triggered garbage collector as long as the values provided for the maximum memory usage and the maximum frequency are correct. This means, there shall be no situation where a script can allocate more memory or run more often per garbage collection cycle as specified in the calculations. However this can be hard to ensure if scripts depend on external inputs or other side-effects, like being triggered from a peripheral interface and processing data of arbitrary length. For complex applications it can not be guarantee that the garbage collector will meet its deadline in every possible situation.

Therefore it is important to have at first a monitoring mechanism for the memory allocation to report the memory situation. This monitor can support the integration testing as memory shortages can be detected early.

To avoid crashing of the software due to memory allocation errors, the garbage collector shall have the possibility of "emergency runs" which will occur out of order in case an allocation of an object will fail. In case of an emergency run, the garbage collector will behave like a non real-time implementation and free memory as long as the allocation can be carried out. This call would probably break the real-time capability of the system, but it will avoid the scripts to crash and causing undefined behavior. This vulnerability is later addressed in section V-A4.

H. Unit Testing

A key requirement of safety critical software is the testing and the code coverage of the tests. The DO-178 standard covers these requirements with the objectives 6.4.4.a-d. Lua, as most programming languages, has several unit test frameworks readily available. The test frameworks are capable of automatically running tests of given scripts and report the coverage of the tested code. Some frameworks like LuaTest publish the reports in commonly used formats like the JUnit format for use in continuous integration environments.

As stated before, the unit tests can also be used to estimate the worst-case execution time for a given script. For this purpose, the test framework has to be modified to calculate the execution time of each test case. It has to be ensured that test cases are available that results in the worst possible execution time.

For the measurement of the execution time, the actual unit tests have to run on the target system, as the worst-case execution time is depending on the environment the script is running on. That implies the need for a test environment to run scripts on the target system. This system could be as minimal as the RTOS running the Lua interpreter with the unit-test script.

Using this technique it is possible to determine meaningful worst-case execution times. Additionally to that, the unit tests will become more representative for the use in the final setup.

I. Type Safety

Type safety in the context of reliable systems is a challenge to implement with the Lua language. As explained previously, Lua only uses implicit casts which are often not obvious to recognize. The requirement 6.3.4.f and 6.7.1. of the DO-178C standard however require a consistent use of types and clarifications on the conversion between types. To fulfill these requirements, the implicit conversions have to be marked and the intent shall be stated. This can be achieved by creating coding standards that demands comments on every type conversions to explain the intent. To support the developers and the following certification review, tools should be developed to check the coding standards that are stated below.

J. Type Checks

The type system of Lua evaluates the types of variables at run-time and converts the types if necessary. The conversion itself is not checked implicitly and can lead to a propagation of invalid values or types. A way to prevent the use of invalid types and unintended conversion is the defensive programming technique. Lua supports the techniques with the "assert" keyword which allows to make general assertions at runtime. This includes basic range checks of variables as well as type checks using the string comparison of the example.

An example of using asserts for type checks is listed in the following snippet:

```
function abs(x)
    assert(type(x) == "number",
    "abs expects a number")
    return x >= 0 and x or −x
end
```

When using asserts to check for the parameters it can be ensured that the variables are of the expected type. Requiring defensive programming techniques to be used in the script language will ensure the type safety and can be further used for sanitizing the function parameters.

The coding rules for safety critical Lua scripts shall implement this technique and check every parameter of each function at least for it's type to fulfill the DO-178C requirements. The assertions however can only be used in the development phase.

K. Type Conversions

Lua uses implicit casts to convert variables which are not suitable for certain operations. These implicit casts can be convenient but in the same time confusing when it comes to debugging and checking for the right type of variables.

To clarify type casts the internal conversion functions of the Lua runtime should be used even if they are not necessary. The following listing shows an example with implicit casts:

```
a = "40";
b = a + 2;
```

The standard Lua behavior of the addition operation will implicitly convert the first operand in the second statement to a number. However the type conversion at the second line is not clear and can be confusing in further code as it appears as if the type of a is a number. This confusion gets worse with a larger code base where the uses of variables are further apart.

Lua provides several internal functions to enforce the cast of the variable and clarify the types being used. These functions can be used to highlight a type conversion. The code from the previous listing should therefore rewritten to the following:

```
a = "40";
b = tonumber(a) + 2;
```

This way it is clear that *a* is originally not a number and has to be converted. Future work will evaluate if the implicit type conversion can be disabled in the interpreter to ensure explicit type conversion.

V. Certification Considerations

The aviation domain is safety-critical. As such, development of aircraft is strictly regulated. The de-facto standard for the software development of aircraft is DO-178C ("Software considerations…") [14], as well as the related documents suite, DO-330 ("Software tool qualification considerations…") [15], and DO-332 (Object-oriented technology … Supplement") [16]. In the context of this project the Lua scripting engine has to be considered as flight software. This is due to the fact that the scripting engine will be used on board the aircraft and so the software will actually "fly". To further underline the general maturity of this scripting engine, we refer to a paper "Lua-Based Virtual Machine Platform for Spacecraft On-Board Control Software" [17] that targets using Lua for use as a platform for spacecraft onboard control software. When discussing the certification of a scripting language, the following steps need to be considered: (1) Development of a Lua runtime engine in full compliance of DO-178C software considerations and object oriented technologies supplement DO-332. (2) Development of a Lua program scripts in full compliance of DO-178C software considerations and object oriented technologies supplement DO-332. (3) As an optional trade-off step: Development of a Lua Byte-code compiler in full effect of DO-330 tool qualification compliance.

A. Certifiable Lua Scripting Engine

1) Certification Consideration of DO-332: The DO-178C standard clearly states, that a supplement document has to be used, if it is applicable for a specific technique used during development (see: DO-178C [14], 1.4 How to use this document). As a result, although DO-178C is the basis of the certification approach, it is required to comply to DO-332 for the use of object-oriented techniques. Since Lua is an interpreted language, in this case DO-178C dictates the additional application of DO-332 for the use of scripting languages. The main challenges regarding the use of scripting languages in a certification context is the virtualization, the dynamic memory management and the type safety.

2) Considerations for using Virtualization (Scripting Engine): The main difference of using a scripting language is the fact that the language by design is interpreted by a language engine. This technique is called virtualization in the context of software certification. The available guidance on the virtualization topic, given by DO-332, is given below and summarized in the following:

- OO.1.6.2.7 Virtualization Techniques
- OO.4.0 Software Planning Process
  - OO.4.2 Software Planning Process Activities
  - OO.4.4 Software Life Cycle Environment Planning
- OO.D.1.7 Virtualization

With the statements in [16] OO.1.6.2.7 it is detailed what has to be considered as virtualization, i.e., using abstraction of a set of resources at a higher level. Here, interpreters including programming language interpreters are listed as examples of this technique. Furthermore, [16] OO.4.0 mentions that virtualization software also should comply with DO-178C. As a result, the Lua engine must be interpreted as virtualization software and has to comply with DO-178C and applicable supplements.

With objective [16] OO.4.2.m it is detailed what has to be considered as executable code, i.e. if data is interpreted as instructions providing flow control. Scripts must be interpreted as executable code and all applicable objectives must be satisfied. With this objective [16] OO.4.4 it has to be assured that virtualization software is included in the software planning process and that it is included in the target environment.

The [16] OO.D.1.7 description of the vulnerabilities of virtualization highlights the importance of categorizing scripts as executable code and not as data.

The [16] OO.D.2.4.3.3 description highlights the impact on virtualization for the scheduling and consequently the worst-case execution timing. The virtualization has to be considered when performing worst-case execution timing analysis.

3) Considerations for Type Safety and Type Conversion: Type safety is important for object-oriented languages. In this context it has to be ensured that uses of classes and superclasses are safe. In the following, the guidance regarding type safety and type conversion is detailed and a summary is given below.

- [16] OO.1.6.1.2 Types and Type Safety
- [16] OO.D.1.4 Type Conversion

The Lua scripting language is dynamically typed and there are eight basic types: nil, boolean, number, string, function, userdata, thread, and table. It should be noted that the number type represents real (double-precision floating-point) numbers. Furthermore, Object-oriented programming is based on tables. However, it is not necessary to use object-oriented programming techniques while using Lua. As such, with the focus on this research project, the use of object-oriented programming is not further analyzed. Type safety between classes and superclasses are therefore not relevant in this analysis.

With [16] OO.D.1.4.3.a and [16] OO.D.1.4.3.c it has to be assured that type conversions are safe, implications understood and that narrowing type conversions are made explicit. Since Lua has only a single type for numbers, this is basically automatically assured. In context of software certification a coding rule should be enforced to only allow explicit type conversions between strings and numbers. With [16] OO.D.1.4.3.b and [16] OO.D.1.4.3.d it has to be assured that upcasting and downcasting is safe. This refers to object-oriented programming, which is not necessary to use and will not be further analyzed.

Lua is dynamically typed. As such, programming errors can lead to runtime errors. However, the use of a single number type in Lua should enable safe number arithmetics. For type conversion between types, i.e. strings and numbers, a coding rule should enforce explicit conversion.

4) Considerations for Dynamic Memory Management: The guidance on the topic of dynamic memory management, given by DO-332, is given below and summarized in the following:

- [16] OO.1.6.2.6 Dynamic memory management

- [16] OO.6.8 Dynamic memory management verification
- [16] OO.D.1.6 Dynamic memory management
- [16] OO.D.2.4 Resource analysis

In general, the DO-332 Object-oriented Technology Supplement [16] OO.1.6.2.6 characterizes four main techniques for managing dynamic memory. These techniques are briefly described in the standard.

With [16] OO.1.6.2.6.1 Object Pooling an initial set of unused objects is provided. Only the initial number of objects can be used during runtime. However, certification requirements show that for this approach most verification activities for memory management need to be applied on the application layer, see Table I. This can increase the certification efforts, especially in the context of the scripting language. [16] OO.1.6.2.6.2 Activation Frame Management refers to the technique of automated object allocation with reference to the execution stack (stack allocation) and scope (scope allocation). This technique specifically limits the usability of dynamic memory use for example in a given method and its called methods. The traditional memory management approach used for C programming is referred to as [16] OO.1.6.2.6.3 Manual Heap Management. However, it is error-prone and it is actually discouraged by the standard to manage dynamic memory manually. Furthermore, the application developer has to ensure that fragmentations do not cause allocations to fail. Finally, garbage collection is mentioned as the standard form of [16] OO.1.6.2.6.4 Automated Heap Management. The benefits for this technique is that most verification activities, except activity [16] OO.6.8.2.d, can be performed once for the garbage collector, instead of on the application layer (individual scripts), see Table I. This means that if the verification activities are performed for the scripting engine, there is no need to verify these activities for individual scripts. For additional details on these techniques, refer to the standard document. Each of the verification activities mapped to a corresponding vulnerability of the dynamic memory management feature (activities paraphrased and shortened).

- a. Verify exclusivity
- b. Verify allocation success
- c. Verify unused memory reclamation
- d. Verify sufficiency for maximum memory
- e. Verify reference consistency
- f. Verify atomic object moves
- g. Verify memory management time bounds

Furthermore, in section [16] OO.D.2.4.2.2.2 Automatic Reclamation Of Memory the standard acknowledges 3 approaches for dynamic memory management (additional approaches might be possible): Periodic, Slack-based, Work-based. The periodic approach uses a separate thread, which is run periodically with the highest priority. The separate thread assures that no deadlock can occur and the high-priority ensures that the garbage collector is run periodically and thus the memory is freed before the program execution continues. The Slack-based approach also uses a separate thread; however the garbage collector is run at low priority, it is run in slack-time, i.e. idle time. With this approach a worst case execution time analysis has to ensure that the program execution leaves enough idle time for the garbage collector to run periodically. Finally, the Work-based approach uses no separate thread. However, a so-called collector is part of the dynamic memory management that allocates new memory and in this case also automatically tracks the allocations and the allocation rate. Therefore it has global knowledge of the memory resources and can therefore decide if a garbage collector run has to be made every time new memory is being allocated.

TABLE I: [16] OO.D.1.6.3: Techniques for dynamic memory management and verification activities

Technique	**Verification activities from OO.6.8.2**						
	a	**b**	**c**	**d**	**e**	**f**	**g**
Obj. pooling	AC	AC	AC	AC	AC	N/A	MMI
Stack allocation	AC	MMI	MMI	AC	AC	N/A	MMI
Scope allocation	MMI	MMI	MMI	AC	AC	MMI	MMI
Man. heap allocation	AC	AC*	AC	AC	AC	N/A	MMI
Aut. heap allocation	MMI	MMI	MMI	AC	MMI	MMI	MMI

AC = application, MMI = memory-management infrastructure, N/A = not applicable, * = difficult to ensure by either application or MMI.

As a result of this analysis, automated garbage collection is the most promising technique for handling dynamic memory for our script engine. The implementation used for this use-case employs a periodic garbage collector solution. To ensure safety, the standard requires analyzing of the amount of free memory and the allocation rate to ensure that objects are reclaimed quickly enough.

The vulnerabilities of dynamic memory management are fully addressed by the verification activities of the standard.

B. Memory Pool

The use of a garbage collector also requires an implementation of a memory pool. For the purpose of the prototype implementation, this memory pool was implemented in a simplified form. Specifically the implementation does not handle defragmentation, one of the vulnerabilities of dynamic memory management, as discussed earlier. The full implementation of a memory pool can get very complex and cannot be implemented in the scope of this research project. However, from a certification point of view this is not a concern, since there exist known implementations for memory pools that are certifiable.

C. Certifiable Lua Program Scripts

Some of the objectives and activities have to be performed on the layer of the scripting engine and others have to be performed on the layer of the script. With the used approach of garbage collection as the implementation for the dynamic memory management, the following verification activity has to be performed on the script layer.

- OO.6.8.2 d. Verify sufficiency for maximum memory

This has been shown by a worst-case execution timing/memory usage analysis as well as test cases for the script. Additionally, a combination of coding guidelines as well as reviews should assure safe usage of language features.

D. *Trade-off on Tool Qualification*

Lua makes it possible to write scripts and then compiling the scripts into object code. This is done by the Lua compiler "luac". This software would generate software that is executed on the embedded hardware. Therefore a tool qualification process would be necessary for the object code compiler. The step of compiling the script into object code before execution is optional, but improves certain aspects of execution, since this transformation would not be needed at runtime. On one hand, this might seem as an unnecessary step, introducing an additional tool qualification requirement into the process. On the other hand, this could possibly reduce the certification effort of the runtime engine, since this part of the software otherwise would need to undergo the DO-178C certification process. This trade-off between DO-178C certification process and DO-330 tool qualification would be subject to further research.

E. *Certification Summary*

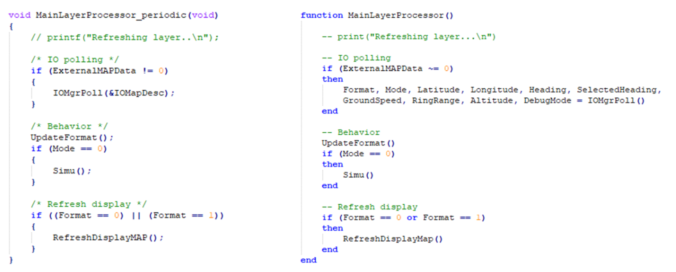
In summary the analysis shows that the approach seems feasible for certification, even up to a level “A” assurance level. Given the prototype implementation, it can be assumed that there exist a combination of approaches and processes to assure the safety of the used concepts of virtualization, the typing system and dynamic memory management. The efforts for developing a certifiable version of the scripting language however can still be significant, but would be comparable to the certification efforts of software of similar complexity and/or size. To further emphasize the efforts required for a certifiable scripting language solution: The same requirements for level “A” software apply. For instance, the scripting engine would require full MCDC test coverage based on the requirements. However, implementing, executing, and handling full certifiable MCDC testing is not possible in this work. This work can only evaluate the necessary modifications and assess the effort required for certification.

VI. Results

The following section states some benchmarks performed comparing the C and Lua implementation of the use-case as well as certain metrics of the implemented Lua engine.

Table II shows the mean time to perform a refresh cycle (shown in Figure 9) for the C and Lua implementation. From a user point of view the performance of the C and Lua implementation are perceived equal and it is difficult to find any difference for both implementations of the use-case.

The results below show that the C implementation achieves superior performance and timing consistency, with loop iterations completing in approximately 26 microseconds and minimal variation. The Lua implementation, while slower with an average time around 600 microseconds per iteration, offers a flexible and scriptable environment suitable for rapid development. Periodic timing spikes in Lua are attributed to garbage collection, but since the GC execution is explicitly controlled in this setup, its impact on performance can be managed and confined to non-critical moments. This allows Lua to maintain predictable timing behavior during normal execution, making it a viable option even for performance-aware applications that benefit from its ease of use and runtime adaptability.



```c
void MainLayerProcessor_periodic(void)
{
    // printf("Refreshing layer..\n");

    /* IO polling */
    if (ExternalMAPData != 0)
    {
        IOMgrPoll(&IOMapDesc);
    }

    /* Behavior */
    UpdateFormat();
    if (Mode == 0)
    {
        Simu();
    }

    /* Refresh display */
    if ((Format == 0) || (Format == 1))
    {
        RefreshDisplayMAP();
    }
}
```

```lua
function MainLayerProcessor()

    -- print("Refreshing layer...\n")

    -- IO polling
    if (ExternalMAPData ~= 0)
    then
        Format, Mode, Latitude, Longitude, Heading, SelectedHeading,
        GroundSpeed, RingRange, Altitude, DebugMode = IOMgrPoll()
    end

    -- Behavior
    UpdateFormat()
    if (Mode == 0)
    then
        Simu()
    end

    -- Refresh display
    if (Format == 0 or Format == 1)
    then
        RefreshDisplayMap()
    end
end
```

Fig. 8: C and Lua implementations

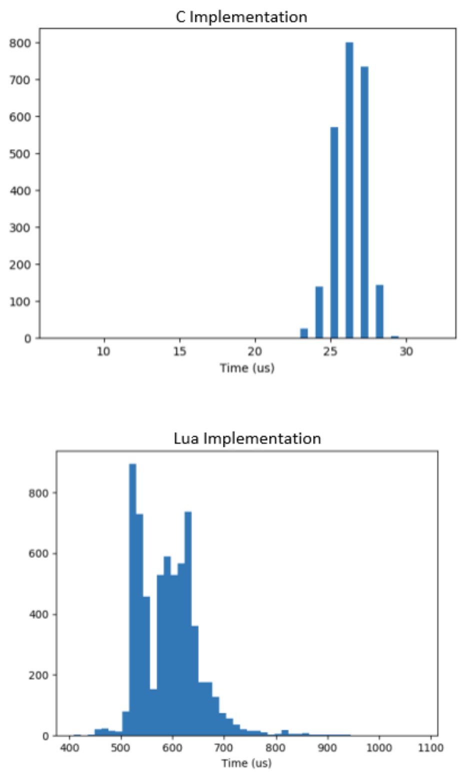


Fig. 9: Execution time histograms for C and Lua

VII. Conclusion and Outlook

This paper showed the cornerstones of a certifiable scripting engine. In general it is feasible to modify the Lua runtime in order to fulfill the requirements of the DO-178C standard.

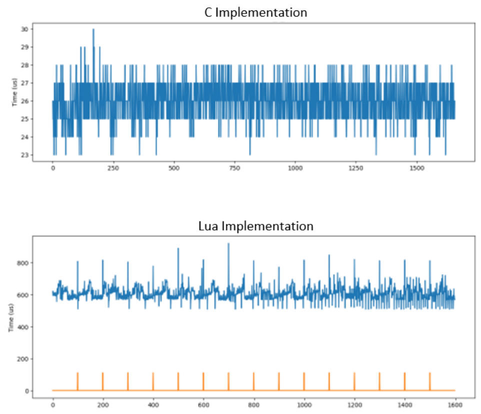


Fig. 10: Per-iteration timing with visible Lua GC spikes

	C Impl	**Lua Impl**
Mean time iteration	26.039 us	591.404 us
Standard deviation	1.136 us	59.889 us

TABLE II: Statistics for C and Lua implementations

It turns out that the reference implementation of Lua already supports the use in real-time systems by implementing features like the incremental garbage collector. The main task for the use in real-time system is therefore a correct integration of the scripting engine.

The elements of the language and the runtime itself do not need to be modified. Only the garbage collector has to be put in a separate process and the interfaces have to be provided. Also, some mechanisms to observe the runtime behavior need to be developed to achieve a safety net if the WCET was not determined correctly due to external inputs. However, with this approach the Lua scripts are compatible to the commonly used tools for testing and extensions to increase type safety. The effort to get scripts certified can therefore be focused on the certification process instead of adapting and developing new tools and extensions.

The paper also presented a proof of concept implementation, results of this and how such approach could be certified up to DAL-A. The efforts for a real-world certification in compliance to DO-178C is vast and cannot be comprised in this work. As a result, this work can be utilized as a basis for a future certification project and give guidance on the overall necessary modifications and efforts.

VIII. 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.

IX. Bibliography

[1] A. Prat, C. Torens, B. Carrick, F.-M. Adolf, F. Giljohann, and D. Lüdtke, "Trade study of scripting languages for avionics systems," in *2024 AIAA DATC/IEEE 43rd Digital Avionics Systems Conference (DASC)*. IEEE, 2024, pp. 01–09.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] S. G. Robertz and R. Henriksson, "Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems," in *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, 2003, pp. 93–102.

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

[15] ——, *DO-330/ED-215 Software Tool Qualification Considerations*. Washington, D.C.: RTCA, 2011.

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

[17] S. Park, H. Kim, S.-Y. Kang, C. H. Koo, and H. Joe, "Lua-based virtual machine platform for spacecraft on-board control software," in *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, Oct 2015, pp. 44–51.