The Suitability of Java for Satellite Applications

Michael Dorin
Information Technology For Aerospace
Universitt Wrzburg
Wrzburg, Germany
mike.dorin@stthomas.edu

Dr. Sergio Montenegro
Information Technology For Aerospace
Universitt Wrzburg
Wrzburg, Germany
sergio.montenegro@uni-wuerzburg.de

Juan Manuel Machuca
Facultad de Ingeniera y Arquitectura
Universidad de Lima
Lima, Peru
jmachuca@ulima.edu.pe

Abstract—**The Java programming language has become widely accepted and popular in many application areas. It is likely that features Java provides allow for the quick and efficient development of applications for use in CubeSats. Java also has many independently designed scientific and data processing libraries available making it worth consideration in the aerospace industry. However, software used for controlling satellite applications should meet established coding standards such as those published by the Jet Propulsion Laboratory (JPL) of NASA. Since Java is an interpreted language, if an application written in Java is to be considered acceptable, both the application and the Java Virtual Machine (JVM) which runs it must be evaluated. Knowing the level of compliance of a Java Virtual Machine is essential. This paper presents a method of establishing the suitability of software for space applications. This process was created using a coding survey directed to software engineers as well as a thorough review of the JPL coding standard documentation. Multiple existing open source Java virtual machines are evaluated, and the suitability of each JVM is presented. This research determined it is possible to use current JVM technology in CubeSats, but not without accepting a certain amount of risk. Understanding how to evaluate a Java Virtual Machine correctly allows creators of satellite applications to make informed decisions on using the Java programming language in their next mission.**

Index Terms—**Java, JVM, Complicated Software, CubeSat Application in Java, Programming, Java Virtual Machine, Open Source, Metric, Level of compliance, java and risk**

I. Introduction

There are many examples of Java use in CubeSat applications. For example, the Ham Radio community has telemetry software written in Java [1]. There are programs and libraries written in Java to create simulations of orbit and orbital decay [2]. Though there are many examples of Java used in ground software, there are few examples of Java used in orbital applications. As of August 2018, GitHub hosts more than sixty Java Virtual Machine (JVM) projects [3]. Wikipedia shows a similar count of active and inactive open source JVM projects. Based on the popularity of Java in aerospace applications and the number of JVMs available, it makes sense to consider using Java in embedded, mission-critical applications. This paper endeavors to answer the question of whether any JVM is suitable for use in space flight. The first step is to define criteria to evaluate the suitability of source code for use in mission-critical applications. Then, various open source Java virtual machines are evaluated for suitability. Finally, a report of findings is presented.

II. Justification for Java

The C programming language has long been accepted as the programming language of choice for aerospace and mission critical applications. Java is now more than 23 years old and should not simply be dismissed for this type of work. In addition to hosting many JVM projects, GitHub also hosts more than four hundred math and science libraries written in Java [3]. Many universities teach Java as the introductory programming language meaning large numbers of Java programmers are available in an academic setting [4]. Professionally, Java is just as popular with programmers as C [5].

III. Criteria for Java Virtual Machine Selection

Many Java Virtual Machines exist so criteria were defined to select which JVMs would be evaluated in this project. First and foremost, the source code needed to be readily available. For this study, the source code must had to available on GitHub [3] or SourceForge [6]. If the source code was too challenging to acquire, it would be of no use to those interested in the project. The next criteria are that the JVM source code must be written in C or C++ and must be simple to build. This is because complicated builds are harder to port to embedded applications. As such complicated builds and builds that required special tools were dropped from consideration. There are also example Java Virtual Machines written in Java, Python, and other interpreted languages. At this point, having a layered JVM is not practical for satellites, so none of these were selected. Ideally, the selected JVMs have some sort of open source license, but many projects have not established any license. However, any project that specifically limited type of use was dropped from consideration.

IV. Criteria for Satellite Software

Care is essential when selecting software that for inclusion in a satellite mission as a failure of the software can mean failure of the mission. This section defines essential characteristics for software used in satellite and other mission-critical applications. The criteria presented here are based on two foundations. The first and most important foundation in determining if a piece of software is space-worthy is whether the source code is too complicated for humans to understand when reviewing it.

If project code is too complicated for a proper review, it is likely to contain faults that go unnoticed. Recommendations included here are based on a recent study demonstrating that software engineers can very quickly visually spot code that is too complicated for review. In this study, programmers were asked to visually scan C and C++ source code and immediately indicate if they felt the code would be pleasant to review or unpleasant to review [7]. Upon completion of the survey, the most important stylistic failures were identified and listed in Table I [7].

TABLE I
MOST UNPLEASANT TO REVIEW STYLES

Style Name
There should be space around operators
Do not write over 120 columns per line
Average length of functions should be short
Indent blocks inside of a function
Put matching braces in same column
Use less than 5 parameters in function
Do not use the question keyword
Avoid deeply nested blocks
Use braces for even one statement

The complexity study also discovered that programs with a high cyclomatic complexity value were considered unpleasant to review [7]. Cyclomatic complexity measures the number of independent paths through a portion of code [8].

The second foundation is based on coding recommendations from NASAs Jet Propulsion Laboratory (JPL) [9]. These standards were summarized through recommendations compiled by Gerald Holzmann from his paper, the Power of Ten-Rules for Developing Safety Critical Code. These are listed in Table II [10].

TABLE II
POWER OF TEN-RULES

Rule Name
Use simple control structures including avoiding goto
Know how long control will remain in a loop
Do not use dynamic memory
Keep function length short
Use assertions to check for conditions that should never happen
Use smallest scope possible for variables and methods
Check return codes from function calls
Do not use preprocessor directives
Limit pointers to only one level of dereferencing
Do not ignore compilation warnings

Since uncomplicated code is an essential aspect of this study, in this section we are going to map how these recommendations are connected to uncomplicated code. For example simple control structures are recommended for use when programming. Though it is superficially easy to search for C goto statements, to be utterly confident that software has simple enough control structures, the code must be easy to understand and review.

Likewise with the upper bounds of loops. It is possible to do running time analysis on loops to determine upper bounds, but understanding the code is indeed the best way to be sure. Limiting pointers to only one level of dereferencing makes understandability to a human reviewer easier as well.

Assertions are essential for checking conditions that should never happen. Uncomplicated to understand source code is also very important in this matter. If a human is unable to understand code, they likely cannot wholly understand conditions that should never happen to place assertions.

An often overlooked recommendation is when calling a function, the return code should be checked rather than assuming success. Intuition tells us that it is easier to determine proper error handling when working with less complicated code. Straightforward things to verify are the absence of preprocessor directives such as #ifdef, the absence of compilation warnings, the inclusion of asserts, and a small total number of lines in a function or method.

From knowing the characteristics of complicated code and the ideal characteristics of software for space applications, a methodology can be created to identify JVMs that are far from compliance.

No dynamic memory is recommended, and in C programming this means that malloc system calls are not allowed. An important feature of Java is garbage collection, which implies dynamic memory. However, it is theoretically possible to eliminate garbage collection and require developers to understand how much memory they need.

V. METHODOLOGY

More than sixty virtual machines are available and manually reviewing each machine was impractical. Built on the understanding of what characteristics are important in satellite software, a mostly automatic process was created with the goal of eliminating JVMs which were very far from meeting the recommendations. The process of evaluation was carried out over several steps.

A. Step 1

As mentioned, only JVMs with readily available source code were used for this study. JVMs were downloaded from GitHub.com [3] and SourceForge [6]. License files for each of the JVMs were reviewed, and those that explicitly forbade certain usage were eliminated. JVMs without a license were not eliminated, but the lack of license is an important consideration from a legal perspective.

B. Step 2

In this step, an attempt was made to build the virtual machine in a Unix environment. A complicated build process is likely too risky, so if it too much effort was required to make the build or if the build could not be completed, the JVM was dropped from the study.

C. Step 3

In this step, the surviving machines were evaluated for software quality. Using a tool called nsiqcppstyle, a review quality report based on the unpleasant to review styles shown Table 1 was generated for each machine [11]. The review quality report calculated a quality percent based on the number of lines in a project and the number of style infractions found in the source code. No machines were eliminated from the study based on this report, but having the value is an important consideration.

D. Step 4

Remaining JVMs were then checked for C constructs such as #if, #ifdef, and malloc. They were also checked for usage of assert and the absence of C goto statements. The results of these actions was logged.

E. Step 5

A small test program was compiled using the OpenJDK [12] environment and run. The rationale for this step is that if we were unable to get a small application running a more complicated application would be impossible. The application is shown in Listing 1. Note the test here is not meant to be an exhaustive test of all of Java's capabilities, only to exercise very basic Java capabilities.

Listing 1. Simple Math Test Program

```
public class HelloJVM {
 public static int math(int a, int b)
 {
   int returnVal;
   returnVal = a * b;
   return returnVal;
 }
 public static void main(String args[])
 {
   int a=2;
   int b=6;
   int answer;
   answer=math(b,a);
   System.out.println("Hello␣"+ answer);
 }
}
```

VI. Results

None of the virtual machines were able met all of the requirements perfectly. This does not necessarily mean that Java should not be considered for satellite applications. However it does preset difficulties that may require extra work to get Java ready for such applications. The results presented are broken up into three sections. First, the standard GNU JVM for Linux and the standard Oracle JVM are discussed. Then the results from analyzing the open source embeddable machines are presented. Finally, two machines which did not make the final selection but were regarded as interesting are shown.

A. Standard Linux JVMs

1) Oracle: The intent of this paper was to discuss open source JVMs due to their review-ability. During the investigation, it became apparent that many of the open-source JVMs are not ready for mission-critical applications and it seemed wise to consider the Oracle JVM. Except for the absence of source code to review, it cannot automatically be considered a bad choice. It is well maintained and is available for different target platforms. Oracle provides good support for their product, and it is easy to install in Linux environments. The Oracle JVM had no trouble executing the code from Listing 1. Oracle presently provides an embedded version of the software which can run in even in Arm environments [13]. If a CubeSat has the resources to support the Oracle JVM and the need exists for Java, it may be the proper choice.

2) OpenJDK and HotSpot: Open JDK is generally thought of as the standard set of Java tools for Unix. Open JDK uses HotSpot JVM, which is the self-proclaimed best JVM on the planet [14]. This is likely the most complete and the most stable of the open source JVMs evaluated in this project. Installing Open JDK on various Linux distributions is easy. HotSpot makes regular use of assert, as recommended by the JPL summary. However, it is not an easy JVM to independently review, based on its size and having a review quality report of only 30.53 percent. HotSpot source also has an average cyclomatic complexity of 2.7, which is very good, but it has some functions which go higher. Hotspot does use the C goto statement various places and there are thousands of preprocessor directives. HotSpot may not easily run on small target platforms or specialized real-time operating systems. HotSpot did properly run the test program in Listing 1. However, on systems that run Linux and have sufficient resources, this could also be a good choice.

B. Embeddable Java Virtual Machines

None of the finalist JVM candidates for embedded systems met the requirements for mission-critical applications. None of these machines properly used asserts to identify conditions which should never happen. However, the JVMs included in this section built and ran Java class files, meaning they may be good candidates for future updating.

1) sJVM by Jakub Veverka [15]: This seems to be a student project, and no license is provided on GitHub. The project has an excellent review quality report of 71.79 percent and it has a cyclomatic complexity of 3.7, though some of the functions included go significantly higher. At present it only has twenty preprocessor directives and it uses no C goto statements. sJVM also is somewhat functional and does operate. sJVM is also very easy to build on Linux and even Macintosh environments and the build generates no warnings. However, it is not a complete Java implementation as it will not properly run all class files created with current Java build tools and did not properly run the code shown in Listing 1.

2) JVM by Arthur Emidio [16]: This also seems to be a student project with no license provided, but the complete source code is available on GitHub. The virtual machine put forth by Arthur Emidio has a good review quality report of 55.56 percent and it has a cyclomatic complexity of 2.6. It does have a considerable number of preprocessor directives but it does not use C goto statements. It also calls malloc twenty-seven times in the code. Though an incomplete JVM, it is somewhat functional. This JVM was very easy to build on Linux but required cmake and had only 2 build warnings. sJVM could run its own test classes but unfortunately was not successfully able to run the test program (Listing 1).

3) Simple Java Virtual Machine by ntu-android [17]: The virtual machine put forth by ntuAndroid has a low review quality report of only 20 percent but it has some features that make it attractive. Reviewing at the code, one can see many obvious stylistic violations which are easily correctable. This JVM also is somewhat functional and operates better than most when tested with compiled Java code. It has an average cyclomatic complexity of 3.1. It uses more than 50 preprocessor directives but uses no C goto statements. It also calls malloc 36 times. The JVM was very easy to build on both Linux and on Macintosh environments, but does generate 62 warnings at this time. It was able to run a very simple hello world program, but not the code from Listing 1. It is packaged with the GNU Public License which makes its use unrestricted.

C. Honorable Mentions

1) Tiny JVM by Julian Offenhuser [18]: During the JVM evaluation period, it was not possible get this virtual machine running correctly, but it is mentioned here because of its size and simplicity. It will build on Linux, but a license has yet to be defined. Executing the included automatic test did not work on the Ubuntu system. That said, it would be very straightforward to port this JVM to different processors and operating systems. Based on a single file, the build process cannot be simpler. It has an relatively high average cyclomatic complexity of 11.2. The parseClass function has a large cyclomatic complexity and is comprised of 190 lines of code. It uses very few preprocessor directives and it has no goto statements. At this time, it uses malloc in nine locations. The size and scope of this one file JVM is very interesting and compelling as a candidate for updating and porting.

2) Nano VM by Tharbaum [18]: This virtual machine was not tested on Linux as it has been made for the Atmel AVR family. It is a good implementation and was actively maintained for years. It has a reasonable review quality report of 45 percent and it cyclomatic complexity of 3.3, though some of the functions go considerably higher. There are 141 preprocessor directives and it has no C goto statements. It does use malloc, but in only one place making it ideal for handling dynamic memory concerns. Many of the complaints in the quality report are very easy to fix, such as adding braces even for one statement and adding spaces around operators. It was released under the GNU public license so has flexibility for use.

VII. Conclusion

As part of the preparation for this paper, more than 60 Java Virtual Machines were reviewed for suitability for satellite and mission-critical applications. The results show that it is possible to use Java for a satellite application, but presently, it is not a perfect choice. There is no single Java Virtual Machine that can meet all the requirements for an ideal application. The recommendation from this review is, for now, use a standard JVM from Oracle or one provided by the OpenJDK project. Unless a CubeSat software team has the background and the time to do extra work, an embeddable JVM should not be considered. Many of the embedded JVMs do not properly build or are simply too complicated to adapt to an embedded target smoothly. It is unfortunate that a JVM suitable for satellite embedded systems was not identified during this study. Future work should be done taking an existing embeddable JVM such as Nano VM and bringing it up to a mission-critical software coding standard. During this process, an abstraction layer should be created allowing the adapted JVM to be ported to different real-time operating systems and processors.

References

[1] "Foxtelem software for windows, mac, and linux," August 2018. [Online]. Available: https://www.amsat.org/foxtelem-software-for-windows-mac-linux/

[2] A. Platzer, "Orbital library," February 2017. [Online]. Available: http://symbolaris.com/orbital/

[3] "Github," August 2018. [Online]. Available: https://www.github.com

[4] P. Guo, "10 most popular programming languages in 2018: Learn to code," July 2014. [Online]. Available: https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext

[5] M. Priyadarshini, "10 most popular programming languages in 2018: Learn to code," June 2018. [Online]. Available: https://fossbytes.com/most-popular-programming-languages

[6] "Sourceforge," August 2018. [Online]. Available: https://www.sourceforge.com

[7] M. A. Dorin, "Coding for inspections and reviews," in *XP '18 Companion, May 2125, 2018, Porto, Portugal*, ACM. New York, NY, USA: Association for Computing Machinery, 2018.

[8] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976. [Online]. Available: http://dx.doi.org/10.1109/TSE.1976.233837

[9] J. P. Laboratory, *JPL Institutional Coding Standard for the C Programming Language*. 800 Oak Grove Dr, Pasadena, CA 91109: Jet Propulsion Laboratory, 2009.

[10] G. J. Holzmann, "The power of ten–rules for developing safety critical code1," *Software Technology: 10 Years of Innovation in IEEE Computer*, 2018.

[11] J. Yoon and K. Tyagi, "Nsiqcppstyle," November 2017. [Online]. Available: https://github.com/kunaltyagi/nsiqcppstyle

[12] "Openjdk," August 2018. [Online]. Available: http://openjdk.java.net/

[13] Oracle, "Oracle java embedded," July 2018. [Online]. Available: http://www.oracle.com/technetwork/java/embedded/overview/index.html

[14] T. H. Group, "Hotspot," 2018. [Online]. Available: http://openjdk.java.net/groups/hotspot/

[15] J. Veverka, "Jakubveverka sjvm," December 2015. [Online]. Available: https://github.com/jakubveverka/sJVM

[16] A. Emidio, "Arthur emidio jvm," March 2016. [Online]. Available: https://github.com/ArthurEmidio/jvm

[17] ntuAndroid, "Simple vm," March 2014. [Online]. Available: https://github.com/ntu-android/simple_vm

[18] J. Offenhuser, "Tiny jvm," July 2018. [Online]. Available: https://github.com/metalvoidzz/TinyJVM