A Comparison of Maintainability Metrics of Two A+ Interpreters

Péter Gál and Ákos Kiss
Department of Software Engineering, University of Szeged, Dugonics tér 13., 6720 Szeged, Hungary

Keywords: Maintainability, Metrics, A+, .NET.

Abstract: Reports on reimplementing or porting legacy code to modern platforms are numerous in the literature. However, they focus on technical problems, functional equivalence, and performance. In the current paper, our goal is to pull maintainability into focus as well and we argue that it is (at least) of equal importance. We conducted source code analysis on two implementations of the runtime environment of the A+ language and computed maintainability-related metrics for both systems. In this paper, we present the results of their comparison.

1 INTRODUCTION

A+ is an array programming language (Morgan Stanley, 2008) inspired by APL. It was created more than 20 years ago to suite the needs of real-life financial computations. However, even nowadays, many critical applications are used in computationally-intensive business environments. Unfortunately, the original interpreter-based execution environment of A+ is implemented in C/C++ and is officially supported on Unix-like operating systems only.

In our previous work, we introduced the A+.NET project (Gál and Kiss, 2012), a clean room implementation of the A+ runtime for the .NET environment[1]. The goal of the .NET-based implementation was to allow the interoperability between A+ and .NET programs (calling .NET methods from A+ scripts, or executing A+ code from or even embedding – as a domain-specific language – into .NET programs), and the hosting of A+ processes on Windows systems. This may extend the lifetime of the existing A+ applications, unlock the existing financial routines to .NET developers, and make .NET class libraries available to A+ developers.

Reports on reimplementing or porting old legacy code bases are numerous in the literature (Wang et al., 2006; Sneed, 2010). However, similarly to our previous work, they usually focus on the technical problems that occur during the porting, on reaching functional equivalence, and on reporting performance data. In the current paper, our goal is to pull maintainability into focus as well. During the development of A+.NET we felt that interoperability is not the only benefit of the reimplementation but the resulting code is somewhat cleaner, easier to comprehend, and more maintainable. To justify our intuition, we conducted source code metric measurements on the two implementations of the A+ runtime, i.e., on the reference implementation and on the A+.NET engine. In this paper, we present the results of their comparison.

The rest of the paper is organized as follows. To make the paper self-contained, we give a glimpse of A+ in Section 2, focusing on the specialities. However, for the exact details of the language, the reader is refered to the Language Reference. In Section 3, we introduce the two A+ runtime implementations, show their differences, and suggest a method that still allows their unbiased comparison. In Section 4, we present the source code metric measurements of the two systems and the result of their comparison. In Section 5, we overview related work, and finally, in Section 6, we conclude the paper and give directions for future work.

2 THE A+ PROGRAMMING LANGUAGE

A+ derives from one of the first array programming languages, APL (Clayton et al., 2000). This legacy of A+ is one of the most notable differences compared to the more recent programming languages. While the operations in modern widespread programing languages usually work with scalar values, the data objects in A+ are arrays. Even a number or a character

[1]Project hosted at:
https://code.google.com/p/aplusdotnet/

Gál P. and Kiss Á..
A Comparison of Maintainability Metrics of Two A+ Interpreters.
DOI: 10.5220/0004597702920297
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-EA-2013), pages 292-297
ISBN: 978-989-8565-68-6

```
(+/a) , ×/a ← 1 + ⍳ 10
```

Listing 1: The computation of the sum and product of the first 10 natural numbers in A+.

is itself an array, of the most elementary kind. This approach allows the transparent generalization of operations even to higher dimensional arrays.

The other striking speciality of A+ is the vast number of (more than 60) built-in functions, usually called operators in other languages. These functions range from simple arithmetic functions to some very complex ones, like the matrix inverse or inner product calculations. Furthermore, most functions have special, non-ASCII symbols associated. This allows quasi-mathematical notations in the program source code, but may cause reading and writing A+ code to be a challenge for the untrained mind.

Finally, although being a language of mathematical origin, A+ has an unusual rule: all functions have equal precedence and every expression is evaluated from right to left.

The above mentioned specialities are exemplified in Listing 1, which shows how the computation of the sum and product of the first 10 natural numbers can be formalized in A+. The expression `⍳ 10` (using the symbol *iota*) generates a 10-element array containing values from 0 to 9, while `1 +` increments each element by one, which is then assigned to the variable `a`. The operator `×/` computes the product of the vector, while `+/` computes the sum. The concatenation function is denoted by comma, which finally results the two-element array `55 3628800`. Note the parentheses around the sum, whithout which concatenation would be applied to variable `a` and the product, and summation would be applied only afterwards because of the right-to-left order of evaluation.

As the above code shows, quite complex computations can be easily expressed in a very compact form in A+. The Language Reference gives further examples, mostly from financial applications, e.g., how to compute present value at various interest rates in a single line (Morgan Stanley, 2008, page 62).

3 TWO A+ EXECUTION ENGINES

In this paper, our goal is to compare the two A+ execution engines: the reference interpreter and our A+.NET system. However, there are several difficulties we have to handle. First, although both implementations aim at doing the same thing, i.e., executing A+ scripts according to the language specification, their architecture and their internal logic is completely different.

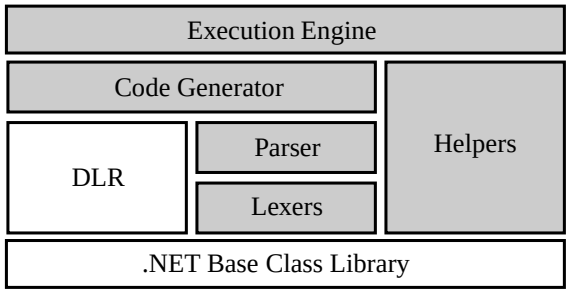


Figure 1: Architecture of the A+.NET runtime. (White boxes denote components provided by the .NET framework, shadowed boxes are modules of A+.NET).

The modules of the .NET implementation, their interrelation, and their behaviour are known and documented (Gál and Kiss, 2012). Figure 1 depicts the main components of the system, and how they build on each other, and on the BCL and DLR (Chiles and Turner, 2009) libraries of the .NET framework. However, the reference implementation has no available documentation other than the source code itself. The only meaningfully deducible model for modules is the directory layout of the source code files.

Other difficulties include the difference in the languages of implementation: the reference interpreter is written in C/C++, while A+.NET in C#. And finally, we admit that the .NET-based version is far from being as functionally rich as the reference implementation. These issues prevent matching lines, functions, classes, or even files directly to each other in the two code bases. Thus, we had to find a way to make our analysis unbiased. We decided to determine the functionally equivalent parts of the two implementations and perform the comparison on those parts.

Determinig the functionally equivalent parts required a reasonably large and diverse A+ code base to drive the engines and a way to track which parts of the engines were exercised by the test inputs. Fortunately, during the development of A+.NET, tests were written for almost every implemented functionality. At the time of writing this paper, this means 1778 test cases, of which 1719 tests (consisting of about 2300 lines of A+ code) could be used as input to both A+ execution engines.

For the reference implementation, we used the instrumentation support of GCC. We recompiled the interpreter and instructed GCC to instrument every function at its entry point with a call to a routine that determines (with the help of the libunwind library (Mosberger, 2013)) the name of the called function at execution time and dumps it out into a log file. Then, this instrumented interpreter was used to

Table 1: Modules of the A+ reference interpreter, the size of each module (given as NF – number of functions, LOC – lines of code, and NOS – number of statements), and the size and ratio of those code parts that are exercised by the test suite. (The granularity of the code coverage information is functions).

Module	Size			Coverage		
	NF	LOC	NOS	NF	LOC	NOS
a	819	7716	12578	576 (70.33%)	5083 (65.88%)	8546 (67.94%)
cxb	26	480	484	1 (3.85%)	18 (3.75%)	15 (3.10%)
cxc	66	1073	877	2 (3.03%)	38 (3.54%)	32 (3.65%)
cxs	1	13	8	0 (0.00%)	0 (0.00%)	0 (0.00%)
cxsys	82	1802	1501	5 (6.10%)	123 (6.83%)	108 (7.20%)
dap	227	4094	2466	18 (7.93%)	344 (8.40%)	208 (8.43%)
esf	201	3690	3494	16 (7.96%)	155 (4.20%)	132 (3.78%)
main	17	425	322	15 (88.24%)	272 (64.00%)	209 (64.91%)
AplusGUI	3328	23761	17041	72 (2.16%)	1768 (7.44%)	1576 (9.25%)
IPC	303	2680	2358	12 (3.96%)	52 (1.94%)	47 (1.99%)
MSGUI	8938	78268	44858	16 (0.18%)	110 (0.14%)	53 (0.12%)
MSIPC	399	2298	1344	28 (7.02%)	261 (11.36%)	156 (11.61%)
MSTypes	4574	27690	15593	100 (2.19%)	431 (1.56%)	179 (1.15%)
TOTAL	18981	153990	102924	861 (4.54%)	8655 (5.62%)	11261 (10.94%)

Table 2: Modules of A+.NET, the size of each module (given as NF – number of functions, LOC – lines of code, and NOS – number of statements), and the size and ratio of those code parts that are exercised by the test suite. (The granularity of the code coverage information is functions. The table does not include data on those parts of the lexers and the parser that are generated).

Module	Size			Coverage		
	NF	LOC	NOS	NF	LOC	NOS
Code Generator	385	5008	1706	217 (56.36%)	4003 (79.93%)	1231 (72.16%)
Execution Engine	87	639	284	55 (63.22%)	355 (54.56%)	176 (61.97%)
Helpers	1213	11396	5020	909 (74.94%)	9171 (80.48%)	4183 (83.33%)
Lexers	5	68	41	3 (60.00%)	48 (70.59%)	29 (70.73%)
Parser	34	255	104	22 (64.71%)	227 (89.02%)	98 (94.23%)
TOTAL	1724	17366	7155	1206 (69.95%)	13804 (79.49%)	5717 (79.90%)

execute the A+ test scripts and by analyzing the log files, we were able to determine which functions were called. For the .NET version, we used the built-in functionality of Visual Studio to gather these code coverage results.

Tables 1 and 2 show the modularization of both systems and data about the size and the test coverage ratio of each module. The reference interpreter (version 4.22) has quite a large code base of C and C++ files. It consists of 153,990 lines of code and 102,924 statements in 18,981 functions (class methods and global functions included). This means that its code is 8.87-14.38x larger than the code base of A+.NET (revision 232), depending on which code size metrics we compare. It is also visible from the tables that a big portion of the reference interpreter is not covered by the tests. This is not a surprise, since the scripts were written as tests for the A+.NET system to cover its functionality. However, if we consider that the major part of the functionality of all the modules of A+.NET correspond to module 'a' of the reference interpreter and only parts of the 'Helpers' module contain code that match other modules of the reference implementation, the coverage results become much closer to each other: the function-level coverage ratio is cca. 70% for both the whole A+.NET system and module 'a' of the reference interpreter. Moreover, the size metrics of the covered code do not differ as much as they do in the case of the whole code bases: in the reference implementation, 8,655 lines of code and 11,261 statements were covered by the tests in 861 functions, while the same data for A+.NET is 13,804 lines, 5,717 statements in 1,206 functions. This means that we managed to identify two function sets, one in each system, of comparable size and of equivalent functionality, which can form the basis of our further investigations.

4 MAINTAINABILITY METRICS

Once we identified the functionally equivalent parts of the two A+ execution environments, their comparison became possible. We used the Columbus tool chain (Ferenc et al., 2002) to analyze the sources of the two systems and to compute such maintainability-

Table 3: Maintainability-related metrics measured for the A+ reference interpreter and A+.NET.

Metric	Reference Interpreter min / avg / max	A+.NET min / avg / max
LOC	1 / 10.07 / 375	1 / 11.46 / 275
NOS	0 / 13.08 / 372	0 / 4.73 / 71
McCC	1 / 4.45 / 123	1 / 2.38 / 71
NLE	0 / 1.04 / 6	0 / 0.59 / 5

related metrics for the covered functions, which were defined for both implementation languages. As a result, we got two size metrics – (executable) lines of code (LOC) and number of statements (NOS), those that were already presented in Tables 1 and 2 – and two complexity metrics – McCabe's cyclomatic complexity (McCC) and nesting level (NLE) – for each function.

Two out of the four metrics above are traditional and well-known: LOC is one of the easiest metric to compute (but often still very informative) and McCC (McCabe, 1976) is also often used. For the sake of completeness however, NOS and NLE is explained below. NOS counts the number of control structures (e.g., if, for, while), unconditional control transfer instructions (e.g., break, goto, return), and top level expressions in a function. This definition makes NOS capture a more syntax-oriented concept of size than LOC (at least for languages where the concepts of line and statement are not related). The NLE metric determines the maximum number of the control structure depth in a function.

Table 3 presents the aggregated metrics for both systems. The averages of NOS, McCC, and NLE, and the maximums of all metrics show that the size and the complexity of the functions in the reference implementation are higher (sometimes significantly higher, see NOS) than in A+.NET, which is usually an accepted mark of lower maintainability. The only outlier is avg(LOC), where the reference implementation produces a lower (i.e., better) metric. However, the difference in the case of this metric is much smaller (a factor of 1.14 only) than for the others.

In addition to the aggregated results, Figure 2 shows the histograms of the computed metrics. (For the size metrics, the histograms use exponentially growing intervals with numbers on the horizontal axis denoting the upper bound of the interval. For the complexity metrics, the intervals are of equal size. The vertical axis denotes the percentage of functions falling in a given interval.) The histogram of LOC explains why the aggregations of the metric do not give a conclusive result. Whether the relative number of functions falling into a size category (interval) is larger in the reference implementation or in A+.NET is almost alternating. However, the histograms of the other three metrics, especially NOS and McCC, strengthen the hypothesis that the reference interpreter is larger and more complex. For A+.NET, a larger portion of functions fall into the small metric ranges than for the reference implementation, while in

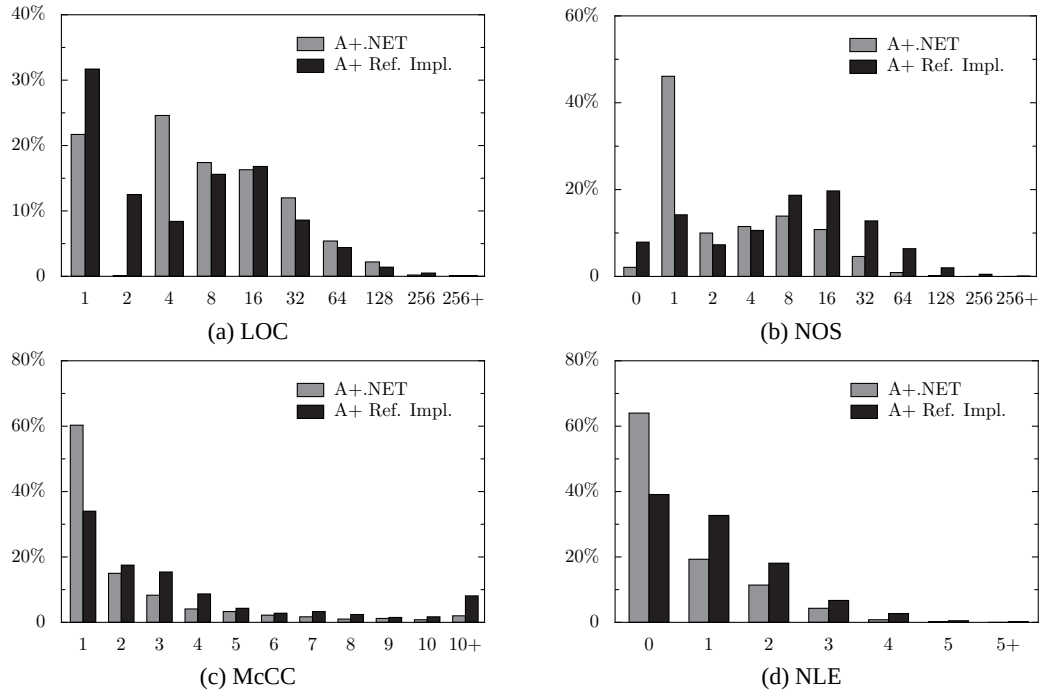
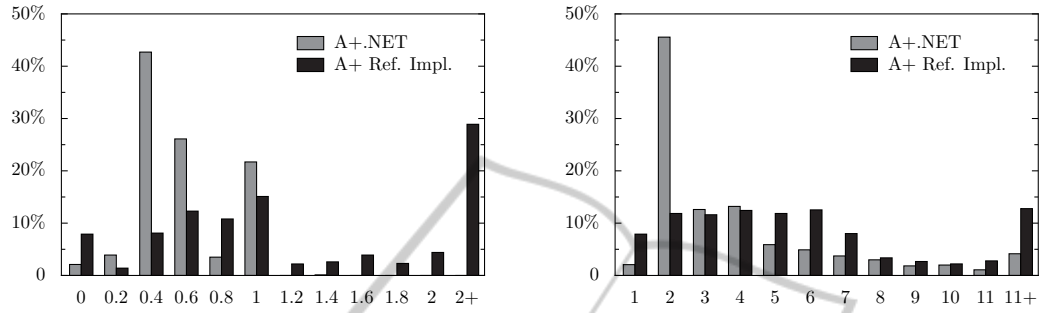


Figure 2: Histograms of the computed maintainability-related metrics.

Table 4: Derived metrics computed for the A+ reference interpreter and A+.NET.

Metric	Reference Interpreter min / avg / max	A+.NET min / avg / max
NOS/LOC	0 / 2.94 / 37	0 / 0.50 / 1.40
McCC + ln(1 + NOS)	1 / 6.42 / 128.36	1 / 3.72 / 75.28



(a) NOS/LOC (b) McCC + ln(1 + NOS)

Figure 3: Histograms of the derived metrics.

the large ranges the reference implementation dominates. We can observe that in A+.NET, cca. 60% of the functions have 2 statements at the maximum, while in the reference interpreter, on the contrary, cca. 60% of the functions have more than 4 statements. For the McCabe complexity, we can see that while in A+.NET only 2% of the functions have a metric value higher than 10, in the reference interpreter 8% of the functions is so complex.

Additionally to the metrics that are directly computed by the Columbus tool chain from the sources, we experimented with derived metrics as well. Our original plan was to compute the Maintainability Index (MI) (Oman and Hagemeister, 1992) for both systems but unfortunately, the current version of the tool chain does not compute the Halstead Volume metric that is a component of MI. Thus, the analysis of MI is left for future work. In this paper, we compute two simpler formulae that grasp some key differences between the reference implementation and A+.NET. The first formula is NOS/LOC: this derived metric tells the average number of statements written in a single line. Guidelines normally suggest one statement per line to keep the code readable and comprehensible. The second formula is McCC + ln(1 + NOS): combining the complexity and the size of a function into a single number. This metric is motivated by the Maintainability Index but the currently unavailable Halstead Volume component is left out and the scale is inverted: now higher values represent larger code size and/or higher complexity, thus – presumably – lower maintainability.

The above discussed derived metrics were also computed for each investigated function, and their aggregated data is shown in Table 4. For all averages and maximums, A+.NET scores considerably lower than the reference interpreter. Moreover, the statements per line metric of the reference interpreter is stunning: the average number of statements in every executable line of source code is about 3, and the most 'crowded' function contains 37 top-level statements in a line on average! Manual investigation revealed that this extreme function consisted of a single line only. However, multi-line functions with a high NOS/LOC ratio are not uncommon either. Actually, cca. 30% of the investigated functions of the reference interpreter have more than two statements on a line on average, as depicted in Figure 3.

5 RELATED WORK

The porting and reengineering of legacy systems have long been the focus of research. However, most authors concentrate on estimating the cost of the reengineering work (Sneed, 2005) or on using automated tools to transform the legacy system from the source language to another target language (Sneed, 2010). These papers focus mainly on the tools developed for the transformation explaining that a great amount of time and effort is required to create such software. In the case of A+.NET, such an automatic transformation was not viable since a completely different architecture – the use of the DLR – was envisioned to allow interoperability with other .NET routines and applications.

As a script language, A+ was not the first to be integrated into the .NET framework. Another successful work was the porting of the Python language,

which resulted in the IronPython project (Hugunin, 2004). The project is still under active development and it is regularly compared to the reference implementation (Fugate, 2010). Most importantly, the DLR also grew out from this project (Hugunin, 2007).

6 SUMMARY AND FUTURE WORK

In this paper, we argued that technical difficulties and performance are not the only important aspects of porting legacy code to modern platforms, but maintainability is (at least) of equal importance. Thus, we took two implementations of the A+ runtime, one that had been developed for 20 years in C/C++ and a clean-room implementation in C#, and investigated them. We described how we managed to identify the functionally equivalent parts of the two systems and then we presented the analysis of maintainability-related source code metrics, as well as some simple derived metrics. The results indicate that the reimplemented version is less complex and thus – presumably – more maintainable.

For the future, we have several plans for improvements. First of all, we would like to increase the functionality of the A+.NET runtime to support more features of the reference implementation. This would result in larger functionally equivalent code bases that can be compared. (And this would also help moving more A+ applications to the .NET-based environment.) Then, we would like to compute and analyze more maintainability-related metrics for the two implementations. Most importantly, we would like to compute the Maintainability Index for C/C++ and C# as well. Furthermore, we would like to build a broad benchmark set that allows the evaluation of the performance of A+ environments. This would allow the evaluation of performance and maintainability side-by-side. Unfortunately, such an A+ benchmark does not exist yet, or it is not publicly available at least.

Although we plan more analysis and comparison for the future, we hope that our current findings can already give guidance in assessing the costs and the gains of porting legacy systems.

ACKNOWLEDGEMENTS

The authors would like to thank Péter Siket and Péter Hegedűs for their help with the Columbus tool chain.

REFERENCES

Chiles, B. and Turner, A. (2009). *Dynamic Language Runtime*. http://dlr.codeplex.com/.

Clayton, L., Eklof, M. D., and McDonnell, E. (2000). *ISO/IEC 13751:2000(E): Programming Language APL, Extended*. Internation Standards Organization.

Ferenc, R., Beszédes, Á., Tarkiainen, M., and Gyimóthy, T. (2002). Columbus – reverse engineering tool and schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181, Montréal, Canada. IEEE.

Fugate, D. (2010). IronPython performance comparisons. http://ironpython.codeplex.com/wikipage?title=IronPython%20Performance.

Gál, P. and Kiss, Á. (2012). Implementation of an A+ interpreter for .NET. In *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*, pages 297–302, Rome, Italy. SciTePress.

Hugunin, J. (2004). IronPython: A fast Python implementation for .NET and Mono. In *PyCON 2004*, Washington, DC, USA.

Hugunin, J. (2007). A dynamic language runtime (DLR). http://blogs.msdn.com/b/hugunin/archive/2007/04/30/a-dynamic-language-runtime-dlr.aspx.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320.

Morgan Stanley (1995–2008). *A+ Language Reference*. http://www.aplusdev.org/Documentation/.

Mosberger, D. (2002–2013). *The libunwind project*. http://www.nongnu.org/libunwind/index.html.

Oman, P. and Hagemeister, J. (1992). Metrics for assessing a software system's maintainability. In *Proceedings of the 1992 IEEE Conference on Software Maintenance*, pages 337–344, Orlando, FL, USA. IEEE.

Sneed, H. M. (2005). Estimating the costs of a reengineering project. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005)*, pages 111–119, Pittsburgh, PA, USA. IEEE.

Sneed, H. M. (2010). Migrating from COBOL to Java. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–7, Timisoara, Romania. IEEE.

Wang, X., Sun, J., Yang, X., Huang, C., He, Z., and Maddineni, S. R. (2006). Reengineering standalone C++ legacy systems into the J2EE partition distributed environment. In *Proceedings of the 28th International Conference on Software Engineering*, pages 525–533. ACM.