EPJ Web of Conferences **214**, 05028 (2019) https://doi.org/10.1051/epjconf/201921405028
CHEP 2018

Sustainability of the Merlin++ particle tracking code

Scott Rowan[1,*], *Sam* Tygier[2,3], *Yuanfang* Cai[4], *Colin C.* Venters[1], *Robert B.* Appleby[2,3], and *Roger J.* Barlow[1]

[1]University of Huddersfield, School of Computing and Engineering, Huddersfield, HD1 3DH, U.K.
[2]University of Manchester, School of Physics & Astronomy, Manchester, M13 9PL, U.K.
[3]The Cockcroft Institute, Sci-Tech Daresbury, Keckwick Lane, Daresbury, WA4 4AD, U.K.
[4]Drexel University, Department of Computer Science, Philadelphia, PA 19104, U.S.A.

Abstract. Merlin++ is a C++ particle accelerator and particle tracking library originally developed at DESY for use in International Linear Collider (ILC) simulations. Merlin++ has more recently been adapted for High-Luminosity Large Hadron Collider (HL-LHC) collimation studies, utilizing advanced scattering physics. However, as is all too common in long-standing high-energy physics software, recent developments have focused on functional additions rather than code design and maintainability. This has resulted in usability issues for users and developers alike. The following presents recent improvements in adhering to modern software sustainability practices to address these issues. Quantifiable improvements in code complexity and maintainability are presented via appropriate test metrics and the evolution of the software architecture is analyzed. Experiences and conclusions of applying modern sustainability methodology to longstanding scientific software are discussed.

1 Introduction

As the lifetime of modern high-energy physics projects such as the Large Hadron Collider (LHC) and the Future Circular Collider (FCC) extend into the order of decades, the longevity of utilized software packages becomes a notable issue [1]. Software sustainability – a software package's capacity to endure in changing environments – is therefore crucial and must be taken into account in software design and implementation [2]. Notably, many software packages, including Merlin++, are long in development with functional additions to the code base over the years having predominately focused on results rather than maintainability. This article discusses current Merlin++ developer efforts to identify and address long-term usability and maintainability issues in accordance with modern software sustainability practices.

1.1 A brief history of Merlin++

Merlin++ is a multi-purpose C++ charged particle accelerator simulation and tracking library. Merlin++, formerly Merlin, has been in varying stages of development and use for over 15 years – originally developed at DESY, Germany, *circa* 2000, by Walker *et al.* for International Linear Collider (ILC) beam delivery system ground motion studies [3]. Merlin++'s functionality was extended to simulate the main linac and damping rings, including

*e-mail: s.rowan@hud.ac.uk

implementation of wakefield, collimation and synchrotron radiation processes. This was soon followed by the implementation of additional beam dynamics functionality, including Twiss parameter and lattice function calculations as well as symplectic integrators. Merlin++ then changed hands, *circa* 2009, and has been further developed by the University of Manchester/University of Huddersfield, UK, to allow for advanced scattering and LHC collimation studies [4–7].

1.2 Software sustainability

The UK Software Sustainability Institute formally defines primary criteria for sustainability such that the '*software you use today will be available – and continue to be improved and supported – in the future*' [8]. Other criteria do exist [2], however, it is widely accepted that overall sustainability can be evaluated from two distinct perspectives. The former is adherence to maintainable principles, practices and processes. The latter is a software's external influence on the socio-economic and environmental surroundings.

1.2.1 Sustainability metrics

Focusing on maintainability and usability, the UK Software Sustainability Institute identifies various test metrics and corresponding criteria to evaluate sustainability, see Table 1. This relatively concise list of metrics has been honed by the Software Sustainability Institute over approximately a decade of evaluating software in both industry and research [9]. This article uses these defined criteria as a baseline for determining overall sustainability.

Table 1: Software sustainability metrics outlined by the UK Software Sustainability Institute.

Usability	Sustainability & Maintainability
Understandability	Identity
Documentation	Copyright
Buildability	Licencing
Installability	Governance
Learnability	Community
	Accessibility
	Testability
	Portability
	Supportability
	Analysability
	Changeability
	Evolvability
	Interoperability

2 Evaluation and improvements

Merlin++ was evaluated against modern software sustainability practices. Various analyses were carried out looking at software package management as a whole as well as code quality and coding practices. The following details the evaluation processes and outcomes.

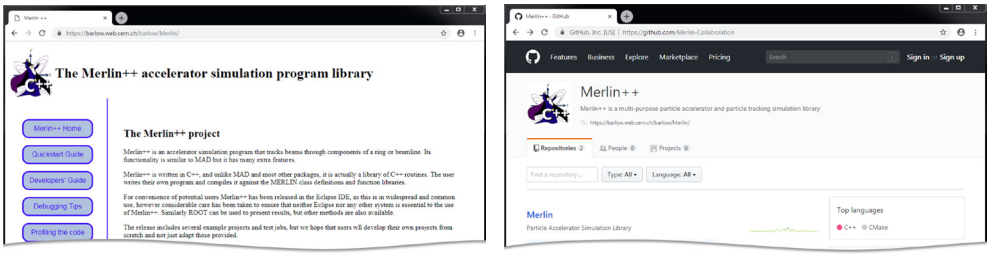
2.1 Software package management

Software package management was evaluated in accordance with the criteria-based assessment developed by the UK Software Sustainability Institute [8]. The assessment provides a number of criteria for each of the aforementioned sustainability metrics. Table 2 shows the assessment outcome before and after recent package management improvements.

Table 2: Merlin++ criteria-based assessment before and after recent improvements.

Sustainability Metric	Met Criteria	Initial Evaluation	Met Criteria	New Evaluation
Understandability	3/7	Unsatisfactory	6/7	Excellent
Documentation	3/19	Poor	14/19	Satisfactory
Buildability	3/9	Unsatisfactory	8/9	Excellent
Installability	7/14	Unsatisfactory	9/14	Satisfactory
Learnability	0/5	Poor	3/5	Satisfactory
Identity	3/7	Unsatisfactory	5/7	Satisfactory
Copyright	1/5	Poor	5/5	Excellent
Licencing	3/4	Satisfactory	4/4	Excellent
Governance	1/2	Unsatisfactory	2/2	Excellent
Community	1/11	Poor	6/11	Satisfactory
Accessibility	6/11	Satisfactory	8/11	Satisfactory
Testability	1/17	Poor	11/17	Satisfactory
Portability	10/16	Satisfactory	10/16	Satisfactory
Supportability	4/19	Poor	10/19	Satisfactory
Analysability	6/16	Unsatisfactory	13/16	Excellent
Changeability	3/10	Unsatisfactory	9/10	Excellent
Evolvability	0/3	Poor	2/3	Satisfactory
Interoperability	2/3	Satisfactory	3/3	Excellent

As shown, the Merlin++ software package management was initially found to be generally unsatisfactory or poor, requiring significant improvements throughout. The developers decided to focus on meeting criteria most likely to improve the new user-developer experience. With this philosophy in mind, a new website was designed and constructed, see Fig. 1(a), and the package source code was migrated from an old and unmaintained SVN repository to a clean and accessible public github repository, see Fig. 1(b) – meeting criteria in installability, community and accessibility (involved work totalled approximately 275 man-hours). Moreover, cmake scripts for simplified build management were developed and build and install guides covering various platforms/IDE's were created – meeting criteria in buildability, testability and analysability (approximately 225 man-hours). To meet learnability criteria a concise and easy-to-understand quick-start guide with the basic information on installation and basic use-cases was constructed (approximately 120 man-hours). Moreover, maintainability criteria led to the development of a developer/coding style guide with defined conventions adhering to modern software engineering practices as well as a formal name change from 'Merlin' to 'Merlin++' to conform to identity and searchability practices (approximately 40 man-hours). Finally, a detailed underlying physics and use guide is being drafted to establish a core source for software citation [10].



(a) New website (b) Github repository

Figure 1: Ensuring sustainable software package management has led to the development of (a) an accessible and user-friendly website (url: accelerators.manchester.ac.uk/merlin) and (b) a clean and public github repository.

Continuing to focus on the new user-developer experience, documentation is now accompanied by easy-to-follow examples and walk-through tutorials for both basic and advanced accelerator design use-cases – found in the quick-start guide. Figure 2 shows an example output plot of what is currently one of seven basic walk-through tutorials provided. The tutorial in question plots the LHC lattice beta and dispersion functions. Other tutorials focus on lattice construction and manipulation as well as particle bunch construction and tracking.

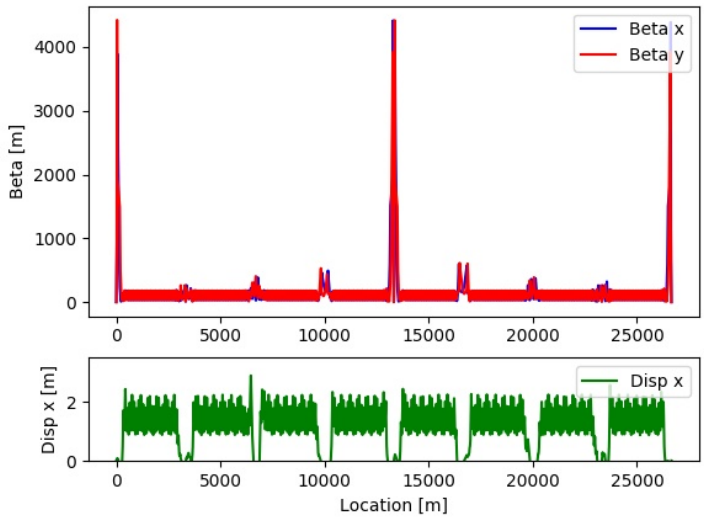


Figure 2: New walk-through tutorials allow users to easily run complex simulations and produce meaningful plots such as the LHC lattice functions shown.

2.2 Code quality

Regarding evaluation of the quality of the code base itself, the developers identified the following sustainability metrics to be the most practically relevant (in no particular order): understandability, learnability, changeability and evolvability. These metrics were identified to

more impactful than others due their known correlation with software adoption in the long-term [11]. This correlation is not only well-known in industry but also aligned with the developers' personal experiences. To quantify these metrics, Merlin++ was evaluated utilizing various static and dynamic analysis tools and methodologies, investigating complexity, dependencies, technical debt and optimization across various system architectures.

2.2.1 Conventions and practices

Standardization of coding and formatting styles is crucial for maintability and readability of a software. For formatting uncrustify [12] was chosen as it is able to work well with both legacy code and new code developed within the eclipse IDE. Other standards and improvements were made, including: the addition of pre-commit hooks to ensure adherence to code styles; refactoring of class and member function names for coherency and understandability, *e.g.* `PointInside()` was renamed `CheckWithinApertureBoundaries()`; standardization of copyright and licensing (GPL2+); development of an extensive practical test suite, including nightly build scripts with results stored online via cdash; formulation of an API/class library documentation using doxygen. All conventions and coding practices are now also outlined in a developer guide which accompanies the source code. Establishing suitable conventions totalled approximately 360 man-hours.

2.2.2 Code Complexity

Cyclomatic complexity – or McCabe value – is a quantitative measure of code complexity which has been shown to correlate with code understandability, learnability and changeability [13]. An example of complex code, identified by a high McCabe value, would be an algorithm containing an excessive amount of *if* and/or *switch* statements. Figure 3 shows the top 50 most complex class member functions within Merlin++ identified by the Metriculator metric analysis tool [14], both before and after complexity reduction refactoring. Note that strict and lenient limits for acceptable levels of complexity are defined to be 15 and 20, respectively. Identification and resolution of complexity hotspots totalled approximately 375 man-hours.

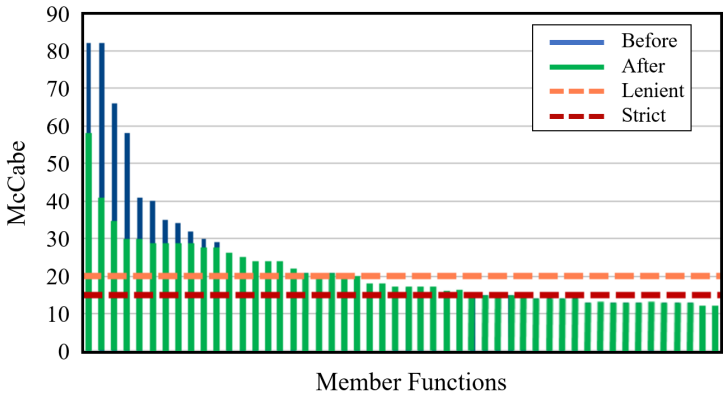


Figure 3: Plot of the 50 most complex class member functions in Merlin++ identified by Metriculator before and after complexity reduction refactoring.

As shown, a significant number of member functions initially far exceeded even lenient complexity thresholds. Investigation led to the identification of a number of common code smells [15], including: Long Methods, Large Classes, Long Parameter Lists, Switch Statements, Alternative Classes With Different Interfaces, Parallel Inheritance Hierarchies, Duplicate Code, Dead code and Middle Man classes. Refactoring was prioritized by identifying class member functions which both exceed lenient complexity limits and exhibit multiple code smells. As a primary example, the Aperture class and related derived classes are not type defined until runtime. The class structure was collectively redesigned to adhere with SOLID class methodology by implementing the factory method design pattern, preventing the requirement for use of switch statements. Moreover, Valgrind [16] dynamic analysis showed that there was a corresponding reduction in speculation misses where switch statements were removed. It is the developer's intention to continue and eventually reduce all member functions below lenient thresholds.

2.2.3 Dependencies & Technical Debt

Technical debt, *i.e.* the associated cost with adding and/or amending the code base, was evaluated utilizing the ArchDia DV8 design structure matrix analysis tool suite [17]. Two principal metrics are identified: MacCormack *et al.*'s Propagation Cost [18] – the amount of additional code which must be changed for any given amendment (lower is better) – and Y.Cai *et al.*'s Decoupling Level [19] – the ability to separate the code into independent submodules (higher is better). Figure 2 shows the Merlin++ dependency analysis across its version history.

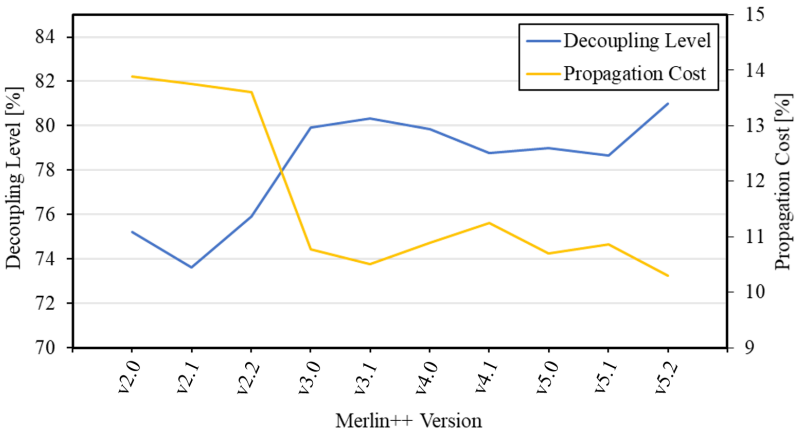


Figure 4: Plot of the evolution of the decoupling level and propagation cost of Merlin++.

With a relatively high decoupling level of 78 % and a propagation cost as low as 0.11, the Merlin++ architecture is shown to be generally good. However, looking at the evolution across previous versions, one can see a significant improvement between versions 2.0 and 3.0, followed by a general decline – a trend mitigated by recent removal of dependency issues, such as unhealthy inheritance and package cycling, also identified using the ArchDia DV8 analysis tool. Identification and resolution of dependency bottlenecks totalled approximately 550 man-hours across 6 months. Note that dependency issues were only removed when correlated with other performance issues/hotspots. It is the developers' intention to eventually remove all identified dependency issues. Adherence to coding practices detailed in the aforementioned developer's guide is expected to minimize future technical debt.

2.2.4 Performance and portability

A crucial aspect of sustainability is user adoption. Alongside functionality, user adoption relies on architecture compatibility and overall performance. Dynamic analysis tools, Valgrind [16] and Intel Parallel Studio XE [20], were utilized to identify runtime hotspots across various different system architectures. Hotspots were identified in the aperture checking algorithms and in the CPU cache utilization. Aperture boundary algorithms were improved via implementation of additional simplified geometric boundary checks which focus on the aperture centre – faster due to the beam being predominately centred. CPU cache usage was improved by implementing additional checks preventing unnecessary bunch copies being made during tracking if no particles were lost or scattered. Figure 5 shows the increase in simulated particles per second on CPUs of varying cache limits. Identification and resolution of the main performance and portability bottlenecks totalled approximately 170 man-hours.

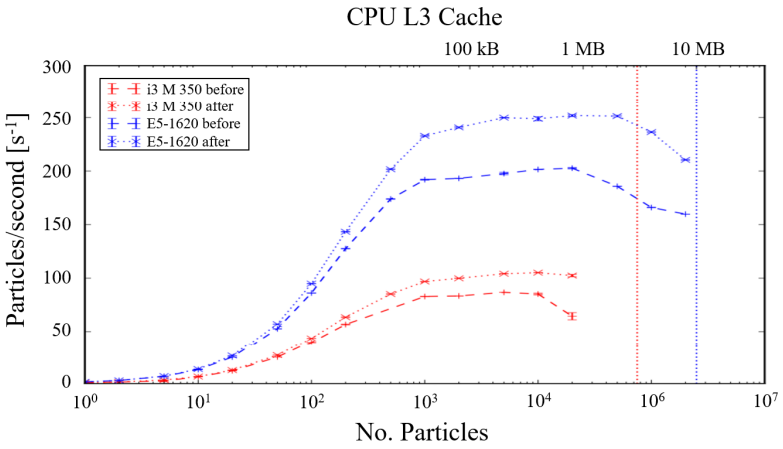


Figure 5: Plot of the evolution of the decoupling level and propagation cost of Merlin++. Vertical lines correspond to CPU L3 cache sizes.

2.2.5 Parallelism and vectorization

Modern computer systems are moving towards multi-core and multi-threaded architecture solutions. As a result, if Merlin++ is expected to be utilized long-term, it is vital that the tracker must take full advantage of all available levels of concurrency. Note that, due to a lack of intra-bunch coupling, some simulations, such as collimation losses, can be inherently parallelized via message passing interface (MPI) technologies – simulating 1 particle per CPU and subsequently accumulating the final loss location of each. This is currently practised using HTCondor [7]. However, multi-threading and vectorization of single particle tracking functions could provide significant performance boosts. Unfortunately, due to the nature of the code, auto-vectorization and basic threading techniques provide minimal improvements, sometimes even being detrimental to performance. It is, therefore, the intention of the developers to focus on implementing explicit low-level methods for threading and vectorization for a future Merlin++ release.

3 Summary and Conclusions

Merlin++ has been thoroughly profiled and analyzed for long-term sustainability of the software package. Significant improvements have been made to the user-developer experience

following a criteria-based assessment constructed by the UK Software Sustainability Institute. Code quality was assessed by numerous static, dynamic and architectural analysis tools. Maintainability and dependency issues were removed when correlated with other performance and/or usability issues, such as runtime bottlenecks or modularity. The potential of multi-threading and vectorization was also investigated and will now be pursued by the developers for a future release. In conclusion, Merlin++ may now be considered a sustainable software package and can be utilized for long-term particle tracking studies.

References

[1] G. Apollinari, I. Béjar Alonso, O. Brüning, P. Fessia, M. Lamont, L. Rossi, L. Tavian, *High-Luminosity Large Hadron Collider (HL-LHC) : Technical Design Report V. 0.1* (CERN, 2017)

[2] C. C. Venters, C. Jay, L. M. S, Lau, M. K. Griffiths, V. Holmes, R. R. Ward, J. Austin, C. E. Dibsdale, J. Xu, *CEUR Workshop Proceedings*, (White Rose Research Online, 2015), RE4SuSy

[3] F. Poirier, D. Krücker, N. Walker, *Proceedings of EPAC 2006* (JACoW, 2006), MOPLS065

[4] J. Molson, H. Owen, A. M. Toader, R. J. Barlow, *Proceedings of IPAC'10* (JACoW, 2010), TUPEC057

[5] R. B. Appleby, R. J. Barlow, J. G. Molson, M. Serluca, A. Toader, Eur. Phys. J. C, **76**, 520 (2016)

[6] H. Rafique, *Doctoral Thesis* (University of Huddersfield, U.K., 2017)

[7] S. Tygier, R. B. Appleby, R. J. Barlow, J. Molson, H. Rafique, S. Rowan, *Proceedings of IPAC2017* (JACoW, 2017), MOPAB013

[8] Software Sustainability Institute, https://software.ac.uk/ [accessed 2018-10-17]

[9] Software Sustainability Institute, https://software.ac.uk/publications [accessed 2018-10-17]

[10] R. B. Appleby, R. J. Barlow, S. Rowan, S. Tygier, Comput. Phys. Commun, (to be submitted)

[11] D. Fontdevila, M. Genero, A. Oliveros, *Proceedings of PROFES 2017*, (Springer,2017), pp. 137-145.

[12] Uncrustify Developer, *Uncrustify*, Release v0.54, http://uncrustify.sourceforge.net/

[13] T. J. McCabe, IEEE Trans. Softw. Eng., **SE-2**, 4, 308-320 (1976)

[14] U. Kunz, *Metriculator* [software], v1.10.2.0, https://github.com/ideadapt/metriculator [accessed 2018-10-17]

[15] M. Fowler, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999)

[16] Valgrind Development Team, *Valgrind* [software], Release v3.14.0, http://valgrind.org/ [accessed 2018-10-17]

[17] ArchDia Inc., *dv8* [software], Release v1.0.1rc2, https://www.archdia.net/ [accessed 2018-11-19]

[18] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Manag. Sci., **52(7)**, 1015–1030 (July 2006).

[19] R. Mo, Y. Cai, R. Kazman, L. Xiao, Q. Feng, *Proceedings of the 38th International Conference on Software Engineering*, (ACM,USA,2016), pp. 499-510.

[20] Intel Corporation, *Intel Parallel Studio XE* [software], Release v2019, https://software.intel.com/en-us/parallel-studio-xe [accessed 2018-11-19]