libmltl: A High-Performance Library for MLTL Parsing, AST Manipulation, and Formula Evaluation

Luke Marzen

May 5, 2024

Abstract

Mission-time Linear Temporal Logic (MLTL) is a finite interval bounded variant of Linear Temporal Logic (LTL). Recently, MLTL has seen high-profile deployments, such as in the verification efforts of NASA's Robonaut2 and Lunar Gateway Vehicle System Manager. Due to the novelty of MLTL, there is a lack of robust and high-performance tooling for parsing and evaluating MLTL formulas. This report details the development of `libmltl`, a high-performance framework for developing MLTL based tooling with both C++ and Python interfaces. In addition to providing MLTL parsing and evaluation capabilities, `libmltl` also provides full access to the Abstract Syntax Tree (AST). An AST interface is crucial for facilitating analysis and transformation of MLTL formulas without expensive and convoluted parsing operations.

1 Introduction

Mission-time Linear Temporal Logic (MLTL) is a finite interval bounded variant of *Linear Temporal Logic* (LTL) that has seen recent adoption [1]–[4]. Notably, MLTL has seen usage in the verification efforts of NASA's Robonaut2 [1] and Lunar Gateway Vehicle System Manager [2].

The syntax of MLTL formulas ϕ and ψ are defined recursively as

$$\phi, \psi := true \mid false \mid p \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \mathcal{F}_{[a,b]}\phi \mid \mathcal{G}_{[a,b]}\phi \mid \phi\,\mathcal{U}_{[a,b]}\psi \mid \phi\,\mathcal{R}_{[a,b]}\psi,$$

where $p \in \mathcal{AP}$, $\mathcal{AP}$ is a finite set of atomic propositions, and $a, b \in \mathbb{Z}$ such that $0 \leq a \leq b$ [1], [5]. The temporal operators, Future, Globally, Until, and Release, are denoted with the same symbols as in LTL, $\mathcal{F}, \mathcal{G}, \mathcal{U}$, and $\mathcal{R}$, respectively. A trace π represents truth assignments of atomic propositions at each discrete time step. We say $\pi \models \phi$ if π satisfies the MLTL formula ϕ, and we say $\pi \not\models \phi$ otherwise.

The objective of this project is to fill the missing gap in open-source MLTL tooling. Specifically this project aims to create a robust and extensible, free and open-source software for evaluating MLTL formulas. To this end `libmltl` was developed. `libmltl` is high-performance framework and library for developing MLTL based tooling with both C++ and Python interfaces. It provides MLTL parsing and evaluation capabilities, as well as, full access to the *Abstract Syntax Tree* (AST). AST interfaces are powerful tools for facilitating analysis and transformation on the structure of MLTL formulas.

The contributions of this project are as follows.

- A high-performance C++ library with Python APIs for MLTL parsing, AST manipulation, and formula evaluation.
- An MLTL testing suite consisting of 53,880 formulas and their evaluations for 4,096 traces.
- Validated `libmltl`, against the WEST[5] test suite of 1,662 formulas.

The code repository is publicly available at `https://github.com/lmarzen/libmltl`.

2 Implementation

`libmltl` follows modern design principles to achieve modularity and efficiency. This includes distinct frontend and backends. The frontend is responsible for parsing while the backend focuses purely on AST analysis and evaluation.

The frontend implements a recursive descent parser. Since the MLTL grammar is defined with left recursion backtracking is required when using a recursive decent parser. Due to the simplicity of the MLTL grammar a top-down parser was chosen despite the presence of left recursion. In the future, an LR parser could be explored as a possible avenue to achieve potentially better parser run time.

The WEST tool suite provides a basic MLTL Evaluator[5] which represents the current state of the art in terms of formula evaluation performance. `libmltl` implements the same algorithms for evaluation as the WEST MLTL Evaluator. Unlike the WEST evaluator, the `libmltl` evaluator operates on an AST and does not require a formula to be re-parsed for each additional trace evaluation. The WEST evaluator evaluates MLTL formulas as it parses it and does not construct an AST. This makes it hard to extend or modify the WEST evaluator without accidentally breaking parsing or evaluation. The WEST evaluator was likely implemented this way to save development time since an AST was not required for its initial purpose. Recursively evaluating on an AST makes `libmltl` especially performant when evaluating the same formula on many traces, such as in model checking applications. Significant effort has been invested in rewriting the evaluation functions to reduce memory allocations and overall memory usage.

Throughout the development of `libmltl`, a number of optimizations were employed that successfully reduce execution time of the regression suite from 2 minutes down to 5 seconds. The regression suite was written before attempting to implement more error-prone optimizations. This was done so that the regression suite could be rerun when new optimizations were added. The most profitable optimizations involved the elimination of copies of sub traces. Evaluating temporal operators necessitates evaluation between specific time bounds in a trace, which we call a sub trace. Eliminating copying of these sub traces required careful use of multiple pointers and offsets to track the start and end of a sub trace and relative positions within a trace. This optimization alone reduced memory allocations by 33% on evaluation of traces with length 6 as measured by valgrind.

Abstract classes are heavily relied on by `libmltl` in order to reduce code duplication and improve extensibility. To improve accessibility to researchers, both C++ and Python interfaces have been implemented. pybind11 [6] is a header-only library that was used to create Python bindings for the C++

source code of `libmltl`.

2.1 Syntax

Table 1 lists the precedence and associativity of MLTL operators as interpreted by `libmltl`. Operators are listed top to bottom, in descending precedence.

Table 1: Operator Precedence

Precedence	Operator	Description	Associativity
1	`!` `~`	Logical negation	Right-to-left
	`F[a,b]`	Temporal finally / eventually	
	`G[a,b]`	Temporal globally / always	
2	`U[a,b]`	Temporal (strong) until	Left-to-right
	`R[a,b]`	Temporal (weak) release	
3	`&`	Logical AND	Left-to-right
4	`^`	Logical XOR (exclusive or)	Left-to-right
5	`|`	Logical OR (inclusive or)	Left-to-right
6	`->`	Logical implication	Left-to-right
7	`<->` `=`	Logical equivalence	Left-to-right

Note: Parentheses can be used to clarify the intended evaluation of a complex compound expression, even if they are not strictly required.

Propositional variables are defined by pN, where $N \geq 0$. *ex:* `p0`, `p1`, `p123`. Propositional constants, $true$ and $false$, are specified using `t`, `tt`, or `true` and `f`, `ff`, or `false`, respectively. Temporal bounds are specified with square brackets, $[a, b]$, such that $0 \leq a \leq b$. *ex:* `G[0,10] p0`, `(p0 & p1) R[3,6] p1`.

The WEST Evaluator can require an excessively verbose use of parentheses even when operator precedence could be used to determine correct evaluation order. In contrast, `libmltl` removes the need for redundant parentheses and can correctly evaluate order of operations.

Precise and descriptive error messages are important to help users quickly fix syntax errors. Special care was taken to provide descriptive diagnostic messages for `libmltl`. Error messages follow in the style of Clang's diagnostic messages where all diagnostics provide column indicators and underlines. Example error messages can be seen in Listing 1.

Listing 1: Example `libmltl` diagnostic messages.

```
>>> parse("G[11,10]~p1")
error: illegal temporal operator bounds subscript
  G[11,10]~p1
   ^~~~~~~~

>>> parse("p1|p2)R[1,10]~p1")
error: unbalanced parentheses, expected '('
  p1|p2)R[1,10]~p1
  ~~~~~^

>>> parse("(p1|p2)U[1,10]bad")
error: unexpected token
  (p1|p2)U[1,10]bad
                ^~~
```

3 Validation

To validate the correctness a cross validation was conducted between `libmltl` and the WEST tool [5]. WEST is a tool for generating regular expressions that describe all satisfying traces for a given MLTL formula. WEST has previously been extensively validated including cross-validation against AllSAT via BDD[7], R2U2+C2PO [8], [9], and FPROGG[10]. Thanks to the help of one of the primary authors of WEST, Zili Wang, we were able to successfully validate that `libmltl` and WEST produce the same satisfying traces for each formula in the WEST test suite.

To further ensure that this project is thoroughly validated with future updates and to assist with the validation of future MLTL tools, we have developed regression tests. The regression tests include evaluating 53,880 formulas over 4,096 traces. The formulas were generated by recursively building

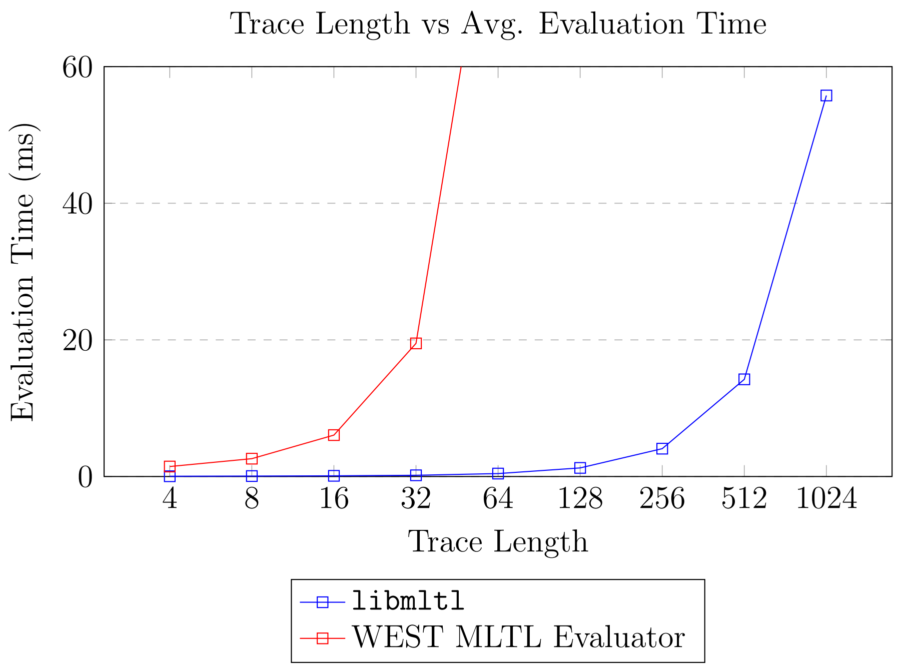

Figure 1: Average MLTL formula evaluation time in milliseconds for traces of increasing length.

all possible formulas to a depth of 3 operators with bounds between 0 and 2 inclusively. The traces are the result of generating all possible traces of 2 variables with length 6. In total, this amounts to 220,692,480 evaluations. These regression tests can also be used as a performance benchmark for future comparisons. Valgrind memcheck was used to verify that `libmltl` does not contain any memory leaks, illegal memory accesses, or reads of uninitialized data.

4 Evaluation

`libmltl` was evaluated against the MLTL Evaluator[5] included as part of the WEST suite. One of the key differences between MLTL Evaluator and `libmltl` is that `libmltl` has a distinct parser and evaluator and can evaluate traces directly on an AST. So `libmltl` must only parse a function once and

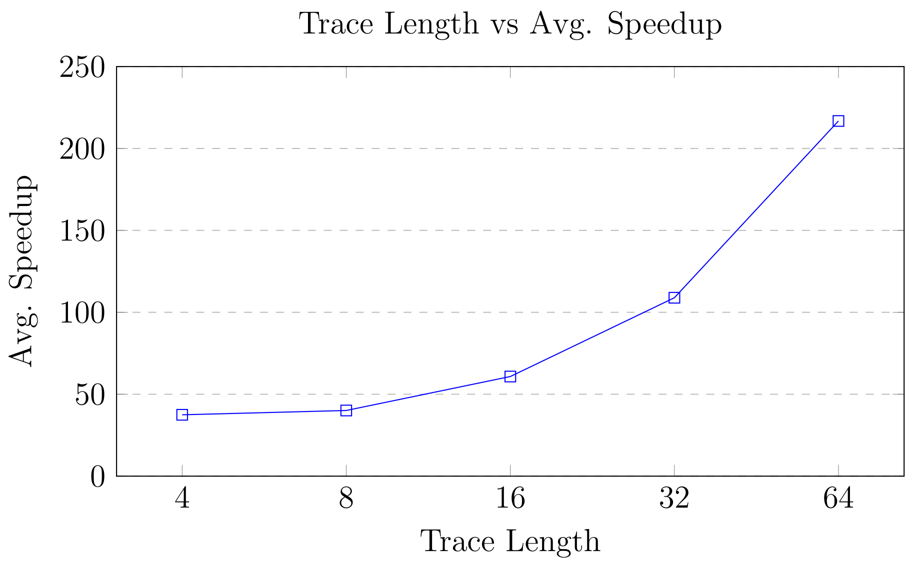

Figure 2: Average speed up of `libmltl` over the WEST MLTL Evaluator.

can evaluate as many traces as necessary, while MLTL Evaluator performs its evaluation while it parses a formula. Both programs were benchmarks on the WEST verification suite of 1,662 formulas with 2048 randomly generated traces. Trace length has a significant impact on evaluation performance, so each benchmark was evaluated with increasingly long traces. With each doubling of trace length, the bounds of temporal operators were also doubled. The following graph compares trace length and average evaluation time of 1662 Formulas on 2048 randomly generated traces for `libmltl` and the WEST MLTL Evaluator.

As shown in Figure 1, the gap between `libmltl` and MLTL Evaluator widens significantly when trace length is increased. This is primarily due to the reduction of memory allocations and copies over MLTL Evaluator. With traces of length 16 `libmltl` offers a speedup of 60×. In the same period of time, `libmltl` is able evaluate traces 16× longer than MLTL Evaluator. Calculating speedups beyond trace length 64 was not feasible due to the prohibitively expensive compute time requirements. Figure 2 shows that `libmltl` achieves a speedup of 37–217× over the previous state-of-the-art when evaluating traces between length 4 and 64.

5 Conclusion

We have shown that `libmltl` is a powerful framework for building MLTL tooling. It provides robust and extensible interfaces for both C++ and Python that enable MLTL parsing, AST manipulation and evaluation capabilities. We successfully validated `libmltl` to a high degree of confidence against WEST and created a regression test suite to address the risk of introducing bugs in future versions. `libmltl` out performs the previous state-of-the-art by a significant speedup of $37\times$ for short traces and more than $217\times$ for traces of lengths longer than 64.

`libmltl` eases the barrier to entry to developing MLTL tooling and serves as the new baseline for future novel MLTL evaluation techniques and represents optimized state-of-the-art performance. The AST provided by `libmltl` will enable the development of sophisticated MLTL analysis and transformation techniques.

References

[1] B. Kempa, P. Zhang, P. H. Jones, J. Zambreno, and K. Y. Rozier, "Embedding Online Runtime Verification for Fault Disambiguation on Robonaut2," in *Proceedings of the 18th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, ser. Lecture Notes in Computer Science (LNCS), vol. 12288, Vienna, Austria: Springer, Sep. 2020, pp. 196–214. DOI: 10.1007/978-3-030-57628-8_12.

[2] J. B. Dabney, J. M. Badger, and P. Rajagopal, "Adding a verification view for an autonomous real-time system architecture," in *AIAA Scitech 2021 Forum*, 2021, p. 0566.

[3] N. Okubo, *Using R2U2 in JAXA program*, Electronic correspondence, Series of emails and zoom call from JAXA with technical questions about embedding MLTL formula monitoring into an autonomous satellite mission with a provable memory bound of 200KB, Nov. 2020.

[4] A. Hammer, M. Cauwels, B. Hertz, P. Jones, and K. Y. Rozier, "Integrating Runtime Verification into an Automated UAS Traffic Management System," *Innovations in Systems and Software Engineering: A NASA Journal*, Jul. 2021. DOI: 10.1007/s11334-021-00407-5.

[5] J. Elwing, L. Gamboa-Guzman, J. Sorkin, C. Travesset, Z. Wang, and K. Y. Rozier, "Mission-time LTL (MLTL) formula validation via regular expressions," in *iFM 2023*, P. Herber and A. Wijs, Eds., Cham: Springer Nature Switzerland, 2024, pp. 279–301, ISBN: 978-3-031-47705-8.

[6] W. Jakob, J. Rhinelander, and D. Moldovan, *pybind11 — Seamless operability between C++11 and Python*, 2016. [Online]. Available: https://github.com/pybind/pybind11.

[7] G. Hariharan, P. H. Jones, K. Y. Rozier, and T. Wongpiromsarn, "Maximum satisfiability of mission-time linear temporal logic," in *Formal Modeling and Analysis of Timed Systems*, L. Petrucci and J. Sproston, Eds., Cham: Springer Nature Switzerland, 2023, pp. 86–104, ISBN: 978-3-031-42626-1.

[8] K. Y. Rozier and J. Schumann, "R2u2: Tool overview," in *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, G. Reger and K. Havelund, Eds., ser. Kalpa Publications in Computing, vol. 3, EasyChair, 2017, pp. 138–156. DOI: 10.29007/5pch. [Online]. Available: https://easychair.org/publications/paper/Vncw.

[9] C. Johannsen, P. Jones, B. Kempa, K. Rozier, and P. Zhang, "R2u2 version 3.0: Re-imagining a toolchain for specification, resource estimation, and optimized observer generation for runtime verification in hardware and software," in Jul. 2023, pp. 483–497, ISBN: 978-3-031-37708-2. DOI: 10.1007/978-3-031-37709-9_23.

[10] A. Rosentrater and K. Y. Rozier, *FPROGG: A Formula Progression-Based MLTL Benchmark Generator*, under submission, 2024.