Student Conference Proceedings
Vol 2, No 1, 2026, Article ID 2555, 4 Pages

Student Conference Lübeck

Proceedings Article

ucumate: A Java Library for Units of Measure

Felix Naumann[a*] · Joshua Wiedekopf[b*] · Michael Lawley[c]

[a]Student of Medical Informatics, Universität zu Lübeck, Lübeck, Germany
[b]Institute of Medical Informatics, Universität zu Lübeck, Lübeck, Germany
[c]Australian e-Health Research Centre, Brisbane, Queensland, Australia
*Corresponding author, email: felix.naumann@student.uni-luebeck.de; j.wiedekopf@uni-luebeck.de

Received 21 January 2026; Accepted 17 June 2026; Published online 10 July 2026

Abstract

Healthcare informatics relies on standards for interoperability. Fast Healthcare Interoperability Resources (FHIR) is the standard for data exchange and it recommends the usage of the Unified Code for Units of Measure (UCUM) standard for representing units of quantities. Yet, existing parsing libraries suffer from incomplete edge case handling and performance limitations, creating reliability issues for production applications. This work developed *ucumate*, a standalone modern Java UCUM library, integrated into the terminology server *Ontoserver*. The implementation employed ANTLR4 for grammar-based parsing, a data-oriented programming paradigm for type safety, and comprehensive testing against both standard and custom test suites covering previously unhandled edge cases. The resulting library demonstrates 40x faster canonicalization and conversion operations compared to existing solutions, achieves full compliance with UCUM specifications, and successfully integrates with Ontoserver's terminology processing pipeline. These improvements establish a robust foundation for UCUM processing in production healthcare environments.

I. Introduction

The Unified Code for Units of Measure (UCUM) is a standardized code system designed to facilitate unambiguous electronic communication of quantities and their units in science, engineering, and healthcare [1]. The specification provides a complete, conflict-free coding system with computational semantics by defining a compositional grammar to form expressions. It defines the syntax for composing UCUM expressions by listing individual units and their potential prefixes along with potential exponents. For example, the expression *m* describes the unit *meter*, *cm* describes *centimeter*, and *cm+2* describes *centimeter*^{2}. Multiple units can be combined to form complex expressions such as *kg.m/s2*, which is the definition for *Newton* (N).

UCUM has been adopted by major healthcare standards including Fast Healthcare Interoperability Resources (FHIR) for reliable unit representation in electronic data interchange. FHIR is a Health Level 7 (HL7) standard that defines how healthcare information can be exchanged between different computer systems [2]. The standard encourages that quantities in FHIR resources use UCUM for unit representation to ensure consistent interpretation across systems. Ontoserver is a FHIR-compliant terminology server that manages clinical terminologies, code systems, and value sets for healthcare applications [3]. As a terminology server processing FHIR resources, Ontoserver requires robust UCUM parsing capabilities to validate and process quantity data types. Currently, a handful of UCUM-aware libraries exist. Most notably, *ucum-lhc*, a JavaScript library for parsing and converting UCUM expressions, and *Ucum-java*, a Java library for parsing and converting UCUM expressions [4], [5]. The latter is used by HAPI FHIR, a popular HL7 FHIR server implementation [6]. The *Ucum-java* library, while functional, exhibits several significant limitations. Specifically, it demonstrates inadequate handling of cer-

tain edge cases within the UCUM specification and suffers from suboptimal performance during unit conversion operations. This work presents *ucumate*, a modern Java implementation of the UCUM specification that addresses these limitations by providing comprehensive edge case support, improved performance, and enhanced functionality while maintaining compatibility as a drop-in replacement for existing solutions.

II. Methods and Materials

The development of the UCUM parsing library involved two main components: the core library implementation and its integration into Ontoserver. The library implementation encompasses creating a parser for the UCUM grammar, a data-oriented programming approach for robust type safety, and comprehensive testing to ensure UCUM specification compliance. The integration work involved designing a flexible plugin architecture for Ontoserver and implementing the UCUM library as the first plugin to demonstrate the system's extensibility.

II.I. Implementation

ANother Tool for Language Recognition (ANTLR4) is a toolkit to generate lexers and parsers from a grammar language [7]. ANTLR4 provides a powerful framework to build and walk parse trees which hides the complexity of managing parse tree traversals by hand. It was chosen because of its popularity and maturity in the field, especially in the Java ecosystem. An ANTLR4 grammar was defined for the UCUM grammar to capture UCUM expressions. The model structure for the parsed and validated UCUM expressions follows the data-oriented programming (DOP) principle [8]. By providing an immutable data model with strong validation, this paradigm prohibits illegal model states, resulting in strong guarantees that business logic rules are always enforced. The traditional approach would be to have a nullable `prefix` field in the `Unit` class as seen in Listing 1. This requires always checking for the presence of this value to avoid unexpected behavior. By contrast, Listing 2 illustrates the DOP design. A `Unit` always consists of a UCUM unit (e.g. `m`) but optionally has a prefix (e.g. `cm`). In the data-oriented approach, the `Unit` is split into different subclasses (records for readability and immutability): `PrefixUnit` and `NoPrefixUnit`. Sealed interfaces, which were introduced in Java Version 14, and the enhanced switch pattern matching from Java 21 allow for a more concise and thus more readable syntax than traditional equivalents (Listing 2).

Required operations like validation, canonicalization, and conversion are implemented as separate functions that operate on the immutable data structures, with caching layers for expensive operations.

```
class Unit {
  private Prefix prefix;
  private UCUMUnit ucumUnit;
  Unit(Prefix prefix, UCUMUnit ucumUnit) {
    this.prefix = prefix;
    this.ucumUnit = ucumUnit;
  }
}
if(prefix != null) // unit with a prefix
else // unit without a prefix
```

Listing 1: Illustrative traditional programming example for UCUM units with potential null problems.

```
sealed interface Unit {}
record PrefixUnit(prefix, unit) implements Unit {}
record NoPrefixUnit(unit) implements Unit {}
switch (unit)
  case PrefixUnit(prefix, unit) -> {/*...*/}
  case NoPrefixUnit(unit) -> {/*...*/}
```

Listing 2: Illustrative data-oriented programming example for UCUM units enforcing non-null and exhaustiveness.

The actual data (base units, prefixes, etc.) is distributed by UCUM through an XML file (*ucum-essence.xml*). This is read and parsed into the immutable data structure as well.

Testing included validation against the existing UCUM test suite to ensure specification compliance. Beyond that, an additional test suite with more than 2000 test cases for validation and conversion has been created. The test cases are sourced from common publicly available UCUM lists and from manual edge case testing.

II.II. UCUM Plugin for Ontoserver

Ontoserver works with FHIR resources and supports arbitrary terminologies that can be loaded as finite code lists. However, it provides additional specialized capabilities for certain large terminologies like Logical Observation Identifiers Names and Codes (LOINC) and Systematized Nomenclature of Medicine (SNOMED CT) [9], [10]. It uses ad hoc checks throughout the code base to detect these terminologies and apply optimizations via Lucene, an indexing and search engine.

This ad hoc approach does not scale when adding support for terminologies with compositional grammars like UCUM. A compositional grammar allows composing an infinite number of valid expressions according to grammatical rules, similar to how natural language works. Such grammars require specialized parsers that understand the grammar rules in order to parse and validate expressions, which is fundamentally different from loading and searching finite code lists.

To address this, a plugin architecture was developed that allows Ontoserver to handle both traditional termi-

DOI 10.18416/SCP.2026.2555

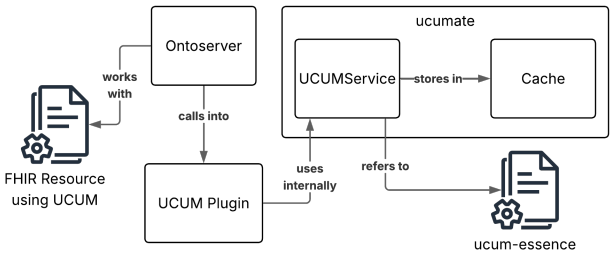

Figure 1: Schematic use case of *ucumate* in Ontoserver through the UCUM plugin.

nologies and those with compositional grammars in a unified and extensible manner, rather than continuing to add ad hoc checks for each special case.

A plugin API mechanism was developed to provide an abstraction layer for the *validate-code*, *expand*, and *lookup* operations. The Ontoserver software was then refactored to also conform to this new API layer. The UCUM plugin is the first external plugin that Ontoserver supports. It is a layer that implements the plugin API and depends on the developed UCUM core library to provide rich UCUM code support (Fig. 1). The *expand* operation is the most challenging because it is impossible to expand an infinite number of expressions. Instead, the plugin expands with all UCUM expressions it knows at that time. This includes all codes defined by the UCUM specification itself as well as all codes the plugin encountered in its lifecycle.

III. Results and Discussion

The implementation of *ucumate* demonstrates significant improvements in UCUM parsing capabilities through comprehensive edge case handling, enhanced performance, and successful integration with healthcare terminology systems. The library achieves full specification compliance while providing extensible architecture for modular terminology processing in Ontoserver. While limitations around precision handling remain, *ucumate* offers substantial improvements over existing solutions and establishes a robust foundation for production UCUM processing in healthcare environments.

III.I. Compliance

The *Ucum-java* library maintains a test suite with validation and conversion tests. *ucumate* is almost fully conformant with the test suite. The failing test cases relate to complex issues with rounding that are discussed in Section III.IV. *ucumate* also passes all tests in its own test suite which has more extensive edge case testing. *Ucum-java* fails in some of the tests because the UCUM specification has some ambiguity for certain edge cases. For example, the specification is unclear whether integer units may be any positive integer number or just positive one-digit numbers. Both cases are mentioned in the specification. UCUM also has problems with the *mole* unit. UCUM defines the unit as dimensionless; it is just the Avogadro constant. In the real world, it is sometimes desirable to convert *mole* to a mass unit like *gram* with a molar mass coefficient instead. This is supported by various UCUM tools including *ucum-lhc* and *Ucum-java*. Therefore, *ucumate* also optionally supports converting *mole* to a mass unit if desired.

III.II. UCUM Plugin

The UCUM plugin is fully integrated into Ontoserver and provides seamless interaction. When a resource that uses UCUM is processed by Ontoserver, the plugin is triggered and returns resources as expected. It supports UCUM versions 2.1 and 2.2. The *lookup* operation returns the common properties *code*, *display*, and *version*. Furthermore, it supports a custom property *metric* that is true if all units in the UCUM expression are metric. Since by definition the *display* for expressions is always the code itself, the plugin adds a human-readable name to the designation attribute.

ValueSets with *include* or *exclude* filters are optimized before expansion by reducing the filters to a single filter. This significantly reduces expansion time for ValueSets with many exclusions, such as including all UCUM terms but excluding all except those in a specific dimension.

The *validate-code* operation sometimes requires not only checking the validity of the provided code, but also checking if the code is in the ValueSet. Therefore, the filter optimization is also used for *validate-code*. Checking if a code is in the ValueSet requires a definition of equality between two expressions. Syntactic equality is trivial, but semantic equality is more complex. For example, the two expressions *1/1* and *2/2* both evaluate to the expression *1*. However, it is unclear whether they should be considered equal. *ucumate* defines two types of equality: syntactic equality and semantic equality. Semantic equality is checked by canonicalizing, flattening, and ordering the expression and then performing a syntactic equality check on the expression and comparing the resulting canonicalization factors. *Flattening* here refers to the process of modifying the expression to only contain multiplication and exponents and then canceling out redundant parts. For example, *m/s2* is flattened to *m.s-2*. Similarly, *2/2* is flattened to *2.2-1* and then the parts cancel out to *1*.

III.III. Performance

Performance benchmarks show that *ucumate* is approximately 6x slower than *Ucum-java* for initial validation but 40x faster for canonicalization and conversion oper-

DOI 10.18416/SCP.2026.2555

ations. The integrated caching layer provides significant performance improvements for subsequent validation operations. With caching, validation is about 4x faster and conversion is up to 130x faster than *Ucum-java*. The benchmarks were performed on a collection of common codes with random access. Detailed benchmark results and methodology are available in the project repository.

III.IV. Limitations

Compliance with the UCUM standard is important, and therefore *ucumate* cannot work around the following limitations with its own definitive solution. Instead, it provides a modular and extensible software architecture that allows users to easily adapt the behavior to their needs.

Since UCUM works with physical units, tools for it have to deal with conversion factors. UCUM itself defines canonicalization factors in the specification, but it does not specify the number of significant digits or whether a factor is exact or measured. For some units, the conversion factor is defined by the SI as a precise factor with no measurement uncertainty, e.g., the factor for converting from the imperial inch to centimeters is defined as exactly 2.54. On the other hand, there are factors derived from physical constants like the Newtonian constant of gravitation, which is defined in UCUM as 6.67430×10^{-11}, limited by measurement precision to 6 significant digits. UCUM provides no way of distinguishing between exact definitions and measured constants. Therefore, a lot of manual effort has to be put in to mark units as having conversion factors with limited precision.

UCUM explicitly mentions that human-readable displays are not part of the specification. However, in many situations it is beneficial to have a human-readable display. *Ucum-java* attempts to define a standard way, but it has some problems. For example, parentheses for operator ordering are removed from printing, which can lead to different expressions. As of now, there is no standard way of printing human-readable expressions.

IV. Conclusion

This work successfully addressed critical limitations in existing UCUM parsing libraries through the development of *ucumate*, a modern Java implementation that leverages data-oriented programming principles for enhanced type safety and maintainability. The library demonstrates substantial performance improvements while achieving full compliance with UCUM specifications and comprehensive edge case support that surpasses existing solutions.

The implementation of a plugin architecture for Ontoserver validates the feasibility of modular terminology processing in healthcare systems. This extensible framework not only enables robust UCUM integration but also provides a foundation for supporting additional compositional grammars beyond UCUM, addressing the scalability challenges faced by terminology servers managing diverse healthcare code systems.

While challenges regarding precision handling and human-readable display remain, these represent opportunities for broader UCUM specification improvements rather than fundamental implementation limitations. The data-oriented approach proves particularly well-suited for parser development, offering compile-time guarantees that eliminate common runtime errors in production healthcare environments.

The *ucumate* library is available as open source software at https://github.com/fhnaumann/ucumate. The UCUM plugin is anticipated to become a standard inclusion in Ontoserver.

Acknowledgments

The work has been carried out at the Australian e-Health Research Centre (CSIRO), Brisbane, and was supervised by the Institute of Medical Informatics, Universität zu Lübeck.

Author's statement

Conflict of interest: Authors state no conflict of interest.

References

[1] G. Schadow and C. J. McDonald, The Unified Code for Units of Measure (UCUM), [Online; accessed 19-08-2025], Regenstrief Institute, 2009. URL: https://ucum.org/ucum.

[2] HL7 International, FHIR R5: Fast Healthcare Interoperability Resources, [Online; accessed 19-08-2025], 2025. URL: http://hl7.org/fhir/.

[3] A. Metke-Jimenez, J. Steel et al. Ontoserver: a syndicated terminology server. *Journal of biomedical semantics*, 9(1):24, 2018.

[4] National Library of Medicine, UCUM-LHC: A Unified Code for Units of Measure Software Library, [Online; accessed 19-08-2025], 2025. URL: https://ucum.nlm.nih.gov/ucum-lhc/.

[5] G. Grieve, Ucum-java: Java UCUM Library, [Online; accessed 19-08-2025], 2025. URL: https://github.com/FHIR/Ucum-java.

[6] University Health Network, HAPI FHIR: Java API for HL7 FHIR Clients and Servers, [Online; accessed 19-08-2025], 2025. URL: https://hapifhir.io/.

[7] T. Parr, The Definitive ANTLR 4 Reference. The Pragmatic Bookshelf, 2013, ISBN: 978-1934356999.

[8] N. Parlog, Data-Oriented Programming in Java - Version 1.1, [Online; accessed 19-08-2025], Oracle, 2024. URL: https://inside.java/2024/05/23/dop-v1-1-introduction/.

[9] Regenstrief Institute, LOINC: Logical Observation Identifiers Names and Codes, [Online; accessed 20-09-2025], 2025. URL: https://loinc.org/.

[10] SNOMED International, SNOMED CT: Systematized Nomenclature of Medicine Clinical Terms, [Online; accessed 20-09-2025], 2025. URL: https://www.snomed.org/.

DOI 10.18416/SCP.2026.2555