EPJ Web of Conferences **295**, 03004 (2024) https://doi.org/10.1051/epjconf/202429503004
CHEP 2023

A dimensions aware evaluator for High Energy Physics applications

Ben Couturier[1,*] and *Marco* Clemencic[1,**]
[1]CERN

Abstract. The LHCb software stack is developed in C++ and uses the Gaudi framework for event processing and DD4hep for the detector description. Numerical computations are done either directly in the C++ code or by an evaluator used to process the expressions embedded in the XML describing the detector geometry. The current system relies on conventions for the physical units used (identical as what is used in the Geant4 simulation framework) and it is up to the developers to ensure that the correct factors are applied to the values entered. Physical units are not primary entities in the framework, it is therefore not possible to check the dimensional consistency of the computation performed. In this paper we investigate the possibilities to add physical units and dimensions to the existing evaluator or to replace it by a more suitable system, and how this would integrate with the possible tools to express units in C++ code (such as boost::units).

1 Introduction

As with any physics related software, the software of High Energy Physics experiments has to deal with physical quantities in numerical computations maintaining consistency and coherency. The solution adopted by LHCb was to use the CLHEP system of units constants which allow to easily convert quantities to a unit of reference and to the numerical value in another unit. For example one can convert from *mm* to *m* with something like

```
length_in_m = ( length_in_mm * unit::mm ) / unit::m;
```

where `unit::mm` and `unit::m` are just numerical conversion factors to a reference unit (for example `unit::mm = 1` and `unit::m = 1000`).

While this works very nicely in most cases it is also error prone as there is no way to prevent mistakes like in

```
time_in_s = ( length_in_mm * unit::mm ) / unit::kg;
```

Such errors can be avoided using one of the many C++ libraries that provide helper classes to enforce at compile time that computations are dimensionally correct and values are correctly scaled to the same unit.

The situation becomes worse when we have to pass some quantity to our C++ applications from, for example, a configuration file. In this case the quantity is often expressed as a

*e-mail: ben.couturier@cern.ch
**e-mail: marco.clemencic@cern.ch

mathematical formula in some persistent format (for example a text file) that has to be parsed and evaluated at run time by a C++ function that can only return a pure number. One can imagine a configuration file describing some selection parameters like

```
energy_threshold = 100 * MeV
```

To make it work we have to use the same selection of reference units in the text representation of the quantity as in the C++ layer that is meant to use the values. The chances of mistakes increase because no C++ library can enforce that the run time computation was performed in the correct system of units. In the example above we have to make sure that the value of `MeV` used by the expression evaluator that translates the string `100 * MeV` matches the value that C++ sees for `unit::MeV`.

In this paper we describe a technique that can be used to enforce dimensional and units consistency when passing values from run time computations to C++ compile time checked physical units libraries.

2 Requirements

For this project we need

- a C++ library that ensures compile time validation of the operations on physical quantities and correct scaling between different units
- an expression evaluator that performs (at run time) the same validation and scaling performed by the C++ library

To bridge the gap between compile time and run time checks and computations we need

- a way to convert a C++ unit to an expression (on units) that the evaluator can understand

3 Libraries

For C++ there are a number of options like Boost.Units[1] and the standardization proposal P1935[2]. Although mp-units[3] (the reference implementation of P1935) is very interesting as it might become part of the standard C++, it only supports C++20 which we cannot use yet, so we have chosen Boost.Units for this project as it is anyway very mature and it supports conversion of C++ units to strings, which is a fundamental requirement for this project.

For the evaluator we had three basic options

- modify and extend an existing evaluator (e.g. CLHEP Evaluator[4], which is already used in LHCb software)
- implement a new evaluator with integrated support for physics quantities
- leverage on an embedded programming language

Extending an existing library or implementing an evaluator from scratch was more complicated and time consuming than what we could allocate to the development of the proof of concept we had in mind, so we decided to reuse what was already available in embedded programming languages and we opted for Lua[5] with the library lua-physical[6]. Similar results can be achieved with Python[7] too.

4 Implementation

The source code of the proof of concept implementation of the evaluator library described here is available on demand. Here we describe some technical aspects of the implementation.

```cpp
class PhysicsEvaluator {
public:
  double eval(std::string_view expression,
              std::string_view result_unit);

  template <auto unit>  // requires C++17
  auto eval(const std::string_view expression) {
    const auto value = eval(expression,
                            details::unit_repr<unit>());
    return value * unit;
  }
  // constructor, destructor and data members are omitted
};
```

Figure 1. Declaration of the C++ class used to bridge physical units checks between compile time (Boost.Units) and run time (Lua + lua-physical). See Sect. 4.3 for details.

4.1 Embedding Lua + lua-physical

Being Lua a programming language developed explicitly for being embeddable, it is straight forward to write a C++ class that wraps initialization and finalization of the Lua interpreter and that exposes a function that uses such interpreter to evaluate numerical expressions.

What is slightly less easy is to embed some third party modules, like lua-physical, in the library so that all the necessary code is provided in one simple binary. For that we embedded as text in C++ the relevant parts of lua-physical so that they could be imported during the initialization of the library.

One particular issue that we had to overcome in order to use lua-physical is the fact that the developer decided to prefix units names by an underscore (e.g. `_mm` or `_s`), so we had to patch the library to have the names we use in our configuration files (such as `mm` or `s`).

4.2 Converting Boost units to strings

One key aspect of being able to validate at run time that a give expression is consistent with the units requested at compile time is to communicate such units to the evaluator. As the evaluator operate on strings this means having a way to convert a C++ unit (like `km / h`) to a string that is a valid expression understood by the evaluator.

The Boost.Units library provides means to convert a unit to a human readable string, but those are not valid expressions (for example `m kg / s` where it should be `m * kg / s`). It is possible to implement a reliable unit-to-string conversion, but for a proof of concept implementation it was enough to post-process the string emitted by Boost.Unit standard conversion to string.

4.3 Connecting run time and compile time

With all the bits in place we need minimal glue code to bridge the gap between the physical units required in the C++ compiled code and the units used in the expression evaluated at run time.

In Fig. 1, the method `eval(std::string_view,std::string_view)` takes the expression string and a string representation of the unit the result should be converted to and

```
TEST_CASE("examples") {
  using namespace boost::units::si;

  PhysicsEvaluator e;

  auto some_speed = l.eval<meter / second>(
    "(10 * m + 1000 * mm) / min"
  );
  CHECK(some_speed.value() == Approx(0.183333));
  CHECK(e.eval<km>("1 * au").value() == 149597870.700);
  CHECK(e.eval<km / h>("1 * km/h").value() == 1.);
  CHECK(e.eval<metre / second>("1 * mi/h").value()
        == Approx(0.44704));

  CHECK_THROWS(e.eval<metre>("10 * h"));
}
```

Figure 2. Examples of use of `PhysicsEvaluator` extracted from actual tests in the package.

returns the result of the expression in the desired unit, raising an exception if the expression is invalid or the desired unit is incompatible with the result of the expression. The method `eval<unit>(std::string_view)` uses a helper function to convert the compile time unit to a string suitable for the other `eval` method and attaches the unit to the numerical result of the evaluation.

The code in Fig. 1 is generic and the same interface could be used for different C++ physical units libraries, as long as we can provide an implementation of the helper function `details::unit_repr<unit>()` or different evaluation engines, hidden behind the `eval(std::string_view,std::string_view)` method.

Fig. 2 shows some examples, extracted from the library unit tests, of how our proof of concept implementation of the units aware evaluator can be used.

4.4 Limitations

The proof of concept implementation described here has a few limitations that prevent it to be used in production.

The Lua library used to add support for physics units in the run time evaluator lacks functions like `sqrt` which is very practical when the expressions we want to evaluate live in the domain of geometry descriptions, like when using Pythagoras' theorem:

```
auto hypotenuse = e.eval<mm>("sqrt((3*cm)^2 + (4*cm)^2)");
```

To overcome the limitation we can implement the missing functions in Lua, or think of other evaluation engine options that do not have such limitations, like Python paired with Pint[8] and NumPy[9]. We prototyped this solution and used it to validate the consistency of the units in all the expressions used within the LHCb detector description. While this solution proved functional, it is not as lightweight and easy to embed in an application as Lua.

The naïve implementation of the unit to string conversion functions used for the proof of concept is fragile and does not cover all possible combinations of units. For a production grade library we will have to implement a more reliable and precise conversion.

5 Conclusion

We described how one can implement an expression evaluator that correctly takes into account physics units at run time and at compile time and we produced a proof of concept implementation based on the library Boost.Units and the Lua language interpreter.

The proof of concept implementation is only the first step. We now intend to evaluate the cost of introducing this kind of evaluator in LHCb software projects, starting from the detector description library. If we decide to migrate the code we plan to improve the presented implementation to overcome limitations such lack of mathematical functions (`sqrt`, `sin`...) or the fragile conversion from C++ unit expression to the evaluator equivalent.

With the proposed evaluator integrated in the LHCb software stack we can slowly transition towards a more correct software setup, where mistakes like forgetting to express the units of a quantity are detected and properly reported.

References

[1] *Boost.units library*, [software], `https://www.boost.org/doc/libs/1_82_0/doc/html/boost_units.html`
[2] M. Pusz, *A C++ approach to physical units*, `https://wg21.link/P1935`
[3] *mp-units - a physical quantities and units library for C++*, [software], `https://github.com/mpusz/mp-units`
[4] L. Lonnblad, Comput. Phys. Commun. **84**, 307 (1994)
[5] *Lua Programming Language*, [software], `https://www.lua.org/`
[6] *The lua-physical library*, [software], `https://github.com/tjenni/lua-physical`
[7] *Python Programming Language*, [software], `https://www.python.org/`
[8] *Pint: makes units easy*, [software], `http://pint.readthedocs.org/`
[9] *Numpy: the fundamental package for scientific computing with python*, [software], `http://www.numpy.org/`