MAUS: MICE Analysis User Software*

C. D. Tunnell†, John Adams Institute for Accelerator Science, University of Oxford, OX13RH, UK,
C. T. Rogers, ASTeC, Rutherford Appleton Laboratory, OX110QX, UK,
for the MICE collaboration.

Abstract

The Muon Ionization Cooling Experiment (MICE) is unique because it measures accelerator physics quantities using particle physics methods. It follows that the software that forms the theoretical model of MICE needs to have both accelerator physics and particle physics codes. MICE addresses this with the new software framework "MAUS". The diversity of challenges that MICE provides means that defining the software scope and layout is critical to the correctness and maintainability of the final accelerator physics analysis. MICE has restructured its code into a Map-Reduce framework to enable better parallelization while also testing the code with unit, functional, and integration tests to ensure code reliability and correctness. The experiences of this transformation are shared.

THE MICE EXPERIMENT

Muon beams have a wide range of potential applications in fields ranging from medicine to defense but the particle physics community is specifically interested in a Neutrino Factory (NF) or Muon Collider (MC) [1]. Cooling is advantageous for a NF because there are three effects that affect neutrino event statistics and physics sensitivities: the size of the far detector, the running time, and the number of muons within the storage ring that are aimed at the far detector. The latter is increased by reducing the transverse emittance such that the muons from pion decay fit within the dynamic aperture of the machine.

MICE will demonstrate the feasibility of reducing the transverse emittance of muon beams. This is called *cooling*. Beam cooling has previously been achieved with stochastic cooling, synchrotron radiation damping and electron cooling [2]. However, these techniques are too slow for muons whose lifetime is 2μs: a different method must be used.

MICE will demonstrate *ionization cooling* for the first time. A cooling cell contains two parts: the first reduces the muon's transverse and longitudinal momenta by ionization loss in material and the second accelerates the muon longitudinally with RF cavities, thereby reducing the transverse momentum while leaving the longitudinal momentum the same. This reduces transverse emittance. This reduces transverse emittance cell-after-cell down to an equilibrium determined by the strength of focusing, and the relative amount of energy loss and multiple scattering in the absorbers. The single cell tested in MICE can cool the transverse emittance by up to 15%.

Emittance reduction will be measured to a precision of 0.1% of the input emittance, significantly more precisely than conventional emittance measurement methods [3]. MICE measures the phase-space coordinates of each particle by using a wide range of detectors. Detectors up- and down-stream of the cooling section measure the position and momentum four vectors; these detectors have already been constructed and commissioned.

SOFTWARE REQUIREMENTS

The MICE analysis software must be both particle physics code and accelerator physics code. The ability to simulate electronics response and reconstruct tracks are examples of particle physics functionality. Accelerator physics requires the functionality to be able to compute transfer matrices and Twiss parameters. Both types of functionality require knowledge of the magnetic fields and geometry, so these codes fit within a single scope or package.

The requirements imposed upon the software were previously addressed by the G4MICE package [4], created in 2002. Test coverage and documentation were missing for the much of the code base making development, use and verification of the code challenging. This is a frequent problem in physics and industry. The extraction principle outlined in [5] was used to refactor the code, where *refactoring* is the systematic process of restructuring code to address changing specifications.

Since it was inefficient to attempt to understand that code, it was frozen and no changes were allowed. To improve the project, small pieces of code were gradually written to replace old functionality. These new codes had quality requirements: good comments, a style guide, and tests. This made it possible to slowly improve the quality and maintainability of the code base while retaining existing functionality.

MAUS

The MICE Analysis User Software (MAUS) has been official MICE software since 2010 and it is intended to move the software project from the proposal stage to the analysis of data [6]. The goal was to restructure the code into a Map-Reduce [7] data flow in order to simplify the interfaces that developers have to follow and aid running the code in parallel. The terms "map" and "reduce" are functional programming terms.

The basic unit of information is "spill-level", which corresponds to a single beam extraction. Spills are indepen-

* We would like to acknowledge the assistance of the Software Sustainability Institute. The work carried out by the SSI is supported by EPSRC through grant EP/H043160/1.

† Corresponding author: c.tunnell1@physics.ox.ac.uk

dent, thus simplifying parallelization. For example, each "map" should process a spill by converting the binary DAQ output to a processable data structure and then applying a track fitting routine. A similar thing can be done to Monte Carlo (MC) simulation of the apparatus. "Reduce" allows functionality that requires access to the entire data set of a single spill: evolution of parameters over time, making histograms, and so forth.

The JSON data structure is used to represent a spill in order to aid developers in extending it and users in understanding it. An example spill input is:

```
{
    "mc_particles": [
        {
            "primary": {
                "energy": 210.0,
                "momentum": {
                    "x": 0.0,
                    "y": 0.0,
                    "z": 1.0
                },
                "particle_id": 13,
                "position": {
                    "x": 0.0,
                    "y": -0.0,
                    "z": -5000.0
                },
                "random_seed": 10,
                "time": 0.0
            }
        }
    ]
}
```

and an example macro that controls MAUS is:

```
import MAUS

# File with particles to simulate
my_input = MAUS.InputJSON("evts.json")

# Create an empty array of maps, then
# populate it with the functionality you
# want to use.
my_map = MAUS.MapGroup()

# Add geant4 Monte Carlo simulation
my_map.append(MAUS.MapSimulation())

# Add electronics models
my_map.append(MAUS.MapTOFDigitization())
my_map.append(MAUS.MapTrackerDigitization())

# Create set of standard demo plots
my_reduce = MAUS.ReduceMakeDemoPlots()

# Where to save output?
filename = 'simulation.out'

# Create uncompressed file object.
# 'w' means write.
output_file = open(filename, 'w')

# Then construct a MAUS output component
my_output = MAUS.OutputJSON(output_file)

#  The Go() drives all the components you
#  pass in, then check the file defined
#  above for the output
MAUS.Go(my_input, my_map,
        my_reduce, my_output)
```

where dataflow in MAUS is illustrated in Fig. 1.

The macro language is Python but components can be written in either Python or C++. SWIG [8] is used to make Python bindings to C++ code which are created automatically.

APPLYING LESSONS FROM INDUSTRY

Knowledge gained within industry can be applied and enable the project to run more smoothly. Various industry procedures were tested in developing MAUS.

Project Management and Issue Tracker

A paradigm shift in how the code was written came naturally with the change in how the project was managed. Before any code was written, a project management website was set up that included a wiki and issue tracker. This allowed people to keep track of task assignment, current bugs, and feature requests. It is a vital tool for establishing the status of various blocks of work while simultaneously providing a useful historical record and institutional memory.

Code Reviews

Code reviews are standard practice in industry but rare within physics. New code requires an hour of review before entering the trunk with other communal code. In addition to tracking down bugs, the review process also helps spread knowledge of the project between developers. It helps people learn from one another while simultaneously decreasing the reliance on specific developers.

Static Code Analysis

Static code analyzers such as Coverity [9] were used. The purpose of this type of tool is to inspect code to determine if there are conditions the code can enter that may lead to unexpected behavior. For example, if a variable is not initialized and there is a way for the variable to be used that would lead to a segmentation fault, this tool will alert the user.

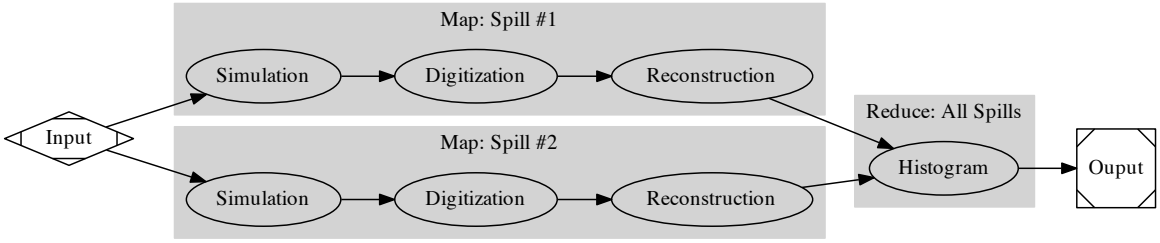

Figure 1: An visual representation of a MAUS control macro that illustrates the data flow.

The static code analyzer finds problems of varying degrees of severity and human intervention is required to categorize them. Given that, it is inefficient to use static code analyzers on legacy code since it takes an unrealistic amount of time to process the wide range of errors. The optimal use case is only to rectify problems observed in new code, thus incrementally improving the code base.

Unit Tests

Unit tests are small pieces of code that test other small pieces of code. They are meant to be granular, deterministic, and repeatable. Their purpose is to allow the person developer to know if they have broken preexisting code. These tests aid in creating releases since one can verify that the code is still functional. If bugs are found, new tests are added to make sure the bug never resurfaces. This type of development also allows one to quickly narrow down the source of a problem.

Integration Tests

The entire system is checked by integration tests that execute at the application level. For example, a large statistics simulation could be used to verify that physics quantities have not changed within statistical uncertainties.

Continuous Integration

Jenkins [10] performs continuous integration tests of the code. This tool runs the test suite in a number of different installation environments every time code is committed. A distributed version control system called Bazaar [11] is used and code from every user is tested before it becomes communal.

These tools have been vital to the project since developers are alerted to broken code. Jenkins is able to try to compile and test the code on a wide range of Linux and Mac platforms to ensure that the code can be deployed to any system. Continuous integration complements unit tests because the frequent running of unit tests allows code developers to know instantly where and when a problem was introduced into the code base.

Release Cycle

Code that has been tested as described above is periodically released. Major releases occur every few months and minor releases are biweekly. The limiting factor on the timescale for minor releases is how long it takes to develop and test new code. This quick release cycle means that bugs are quickly resolved.

FUTURE

The MAUS effort within the MICE experiment has proven to be a successful collaboration physicists and software engineers. There are currently about ten active developers working as a team and using a wide range of tools and methods. A wealth of knowledge and experience exists in the software engineering community and taking advantage of that knowledge has helped MICE.

REFERENCES

[1] S. Geer, “Muon Colliders and Neutrino Factories”, Annu. Rev. Nucl. Part. Sci. (2009)

[2] H. Shropper, “Advances of accelerator physics and technologies”, p. 359 (1993)

[3] Zagel *et. al.*, Complementary Methods of Transverse Emmitance Measurement, BIW (2008)

[4] C. Rogers *et. al.*, ”Simulation of MICE using G4MICE”, EPAC (2006)

[5] M. Fowler, K. Beck, John Brant, W. Opdyke and D.Roberts, “Refactoring: Improving the Design of Existing Code” (1999)

[6] `http://maus.rl.ac.uk`

[7] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, Google Public Note, OSDI (2004)

[8] `http://www.swig.org/`

[9] `http://www.coverity.com`

[10] `http://jenkins-ci.org/`

[11] `http://bazaar.canonical.com/`