Software Quality Control at Belle II

IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 072029 doi:10.1088/1742-6596/898/7/072029

Software Quality Control at Belle II

M Ritter[1], T Kuhr[1], T Hauth[2], T Gebard[2], M Kristof[3] and C Pulvermacher[4] for the Belle II Software Group

[1]Ludwig-Maximilians University Munich, Excellence Cluster Universe, Boltzmannstr. 2, 85748 Garching, Germany
[2]Karlsruhe Institute of Technology, Institut für Experimentelle Kernphysik, Wolfgang-Gaede-Str. 1, 76131 Karlsruhe, Germany
[3]Charles University, Ovocný trh 3-5, Prague 1, 116 36, Czech Republic
[4]High Energy Accelerator Research Organization (KEK), 1-1 Oho, Tsukuba 305-0801, Ibaraki, Japan

E-mail: Martin.Ritter@lmu.de, Thomas.Kuhr@lmu.de, Thomas.Hauth@kit.edu, Christian.Pulvermacher@kek.jp

Abstract. Over the last seven years the software stack of the next generation B factory experiment Belle II has grown to over one million lines of C++ and Python code, counting only the part included in offline software releases. There are several thousand commits to the central repository by about 100 individual developers per year. To keep a coherent software stack of high quality that it can be sustained and used efficiently for data acquisition, simulation, reconstruction, and analysis over the lifetime of the Belle II experiment is a challenge.

A set of tools is employed to monitor the quality of the software and provide fast feedback to the developers. They are integrated in a machinery that is controlled by a buildbot master and automates the quality checks. The tools include different compilers, cppcheck, the clang static analyzer, valgrind memcheck, doxygen, a geometry overlap checker, a check for missing or extra library links, unit tests, steering file level tests, a sophisticated high-level validation suite, and an issue tracker. The technological development infrastructure is complemented by organizational means to coordinate the development.

1. Introduction

The Belle II detector [1] and the SuperKEKB accelerator are currently under construction at the KEK laboratory in Tsukuba, Japan. The aim of this next generation B factory experiment is to collect 50 times more data than its predecessor Belle [2] and to use this data to search for new physics in a variety of B meson, charm hadron, or τ lepton decays with unprecedented precision. The higher luminosity at the SuperKEKB accelerator leads to an increase in background radiation and higher event rates and requires a major upgrade of the detector and the computing system. As a consequence, also the software for the acquisition, simulation, reconstruction, and analysis of data has to be upgraded substantially. Early in the project it was realized that an evolution of the Belle software would not meet all our goals. Therefore, most software components are new implementations, which take the experience from Belle and other experiments and advances in technology into account.

2. Tools

Several state-of-the-art tools are used to facilitate the collaborative software development at Belle II. Modern tools and programming practices are also important to attract and educate new students.

We use modern compilers, currently GCC 6.2 [3], clang 3.9 [4], and the Intel compiler version 17.0 [5]. They allow us to benefit from C++11 features in our code. The parallel usage of different compilers also helps to maintain the portability of the code. The steering of the framework, like the selection of modules and the setting of their parameters, is done in Python 3 [6], a powerful and popular scripting language. This interface provides the possibility to create higher-level meta-frameworks on steering file level, e.g. for physics analyses.

The code is built with SCons [7] which provides a few nice features; it uses Python and therefore does not introduce yet another language in our system, it builds the code in one step and not via an intermediate Makefile like CMake [8] does, and it can work with a central location of the source code plus a partial, updated version in a local directory so that developers only have to check out the part of the code they are working on. Based on SCons, we have developed a build system that is very easy to use because many functionalities are automated. In most cases the user only has to put the code in the right folder structure and specify the libraries that should be linked.

We rely on several external products, like ROOT [9], GEANT4 [10], and EvtGen [11] which are downloaded and built using a custom GNU Make script. A consistent, tagged set of these external products is called an externals version and the basis for our Belle II software. The externals can be compiled from source, but often the developers install a pre-compiled version that we provide for a selected set of Linux distributions. Scripts for the installation and setup of external and Belle II software are in place.

All the code of the Belle II software is now maintained in a central Git [12] repository at DESY and is managed using Atlassian Stash [13]. The repository was recently migrated from Subversion [14].

Developing software in a large project is very different from writing code just for personal use, like analysis programs. To keep people aware that they are part of a community and to produce maintainable code, common guidelines and rules are established. While coding conventions usually cannot be strictly enforced, rules for the formatting style are checked on commits to the Git repository, both locally and on the server side. Commits which do not adhere to these formatting rules are rejected. The Artistic Style (astyle) tool [15] is used for C++ code and pep8 [16] for Python code. One reason for choosing astyle, besides being an open source tool, was that it is not just a style checker, but formats the code. We have included astyle in a small script that developers can use to format their code. Therefore it is very easy to make the code compliant with the style rules. A similar approach is used for Python with the pep8 tool for the style check and autopep8 [17] for the formatting, although there are cases where autopep8 is not able to produce a PEP8-compliant result.

Doxygen [18] was chosen as the main tool for the documentation of the source code as it allows to use the same tool for C++ and Python code. In addition the core part of the Python interface is supplemented with a separate user documentation generated with Sphinx [19]. Documentation that is beyond the scopes of these tools is maintained in the team collaboration software Confluence [13]. A Jira issue tracker [13] is used to coordinate development and to make sure that discovered problems are not forgotten. Developers are reminded once a week about issues assigned to them that are due or have no due date set.

Several tools are used to verify the functionality of the software in different steps. First Google Test [20] is used for low level unit tests. In a next step a custom tool is employed to automatically run steering files that are located in certain test folders and check whether the log output is identical to the expected one. Both tools output a JUnit compatible test summary in

XML format which can be easily parsed by automated tools. A check for memory management issues is executed with Valgrind [21]. In addition we use code analyzers like Cppcheck [22] and Clang Static Analyzer [23] to find problems in the code which the compiler doesn't see.

Finally, we have developed a framework for regression testing and physics validation. It executes, either locally or on a batch system, a set of scripts in the Git repository. There are production scripts that create simulated data files and plot scripts that read the data files and create output ROOT files with validation plots. One- or two-dimensional plots are supported as well as ntuples for monitoring numerical values. XML headers in the scripts provide meta information like the dependencies between the scripts. The plots from all the ROOT files are collected and displayed on a web page. The results from older revisions of the code are overlaid so that regressions or improvements can be detected. Additionally, a reference plot, contact information, a description, and instructions for checking the quality can be provided. Small deviations from the reference are indicated by a yellow plot frame while significant deviations are marked in red.

The output of all these tools is parsed and the number of errors, execution time, event record size and output file size are saved in a database which can then be inspected on a dedicated website. This allows to monitor the time evolution and identify the origin of regressions more easily.

For an efficient and effective software quality control it has to be highly automated. We use Buildbot [24] and Bamboo [13] for this task. Buildbot is very flexible and fits well into our environment because it uses Python for its configuration. Bamboo is part of the Atlassian suite and integrates very well with the issue tracker and the Git server. Our Buildbot setup consists of one server and four slaves with different operating systems: SL6, SL7, Ubuntu 14.04, and Ubuntu 16.04. For Bamboo we only use a single, but powerful Ubuntu 16.04 agent. Several actions are triggered either by commits or at given times:

- On commits an incremental build is started on Bamboo and failures of compilation or tests are reported by email to the committer. This catches, for example, cases where the developer forgot to include a new file in the commit. This is done on all branches and only branches which compile successfully and pass all tests can be merged into the main development line.
- Commits also trigger full builds from scratch on our four different Buildbot systems with three different compilers where all commits in a 10-minute interval are grouped together. A Python class on the Buildbot server keeps track of compilation error and warning messages and informs the developer if any new ones show up. The errors and warnings are also reported on a web page with links to relevant locations in the source code. Results are also directly visible in the Stash interface.
- Each night a full build with different compilers is executed on Buildbot. Furthermore, cppcheck is applied to the code, tests are executed, the geometry is checked for overlaps, documentation is generated with Doxygen and messages about missing documentation are recorded, and SCons debug output is analyzed to determine code dependencies and report missing or unnecessary library links. If any issues are detected, an email is sent to the person responsible for the corresponding part of the code base. The result of all checks is displayed on a web page.
- The validation suite and checks memory management issues are also executed automatically on a daily basis.
- We also use the Buildbot to update the installation of setup scripts, externals, and releases at sites. New monthly builds, releases or external versions are automatically installed on CernVM-FS [25].

- Monthly builds are automated such that the latest tagged versions of all components are checked for compatibility and built. In case of a failure this procedure is repeated the next day. If the build is successful the code is tagged in Git and the new version announced on a mailing list together with a report of changes that are provided by the responsible persons in their tags.

3. Organization

A bunch of tools is not sufficient for a successful collaborative software development. In particular for the collaborative aspect an organization that on the one hand provides sufficient guidance to prevent a divergent development and on the other hand leaves enough freedom to not suppress the creativity of the developers is essential.

The Belle II software group has sub-groups for different tasks: the conditions database, the signal event generators, the detector simulation, the generation and simulation of background, the track reconstruction, and the alignment and calibration. Many further tasks are covered by individuals or small groups so that there is no need for a formal group definition. Developers are often also members of a detector or physics group which facilitates the communication of the software group with the whole Belle II collaboration. In addition we have an official contact person from each sub detector group to make sure that the software group is aware of all the problems faced by the other groups and can provide common solutions if possible.

The source code currently consists of about one million lines of code and is structured in packages. A package, for example, contains the code related to a detector, the core framework, or the event generators interfaces. Each of the about 30 packages has a librarian who is responsible for the code inside it. The librarian has the right to make a tag of the package. Librarians can decide who is allowed to commit changes to their package directly to the master branch. All other users can propose changes to all package in separate feature branches which can then comfortably reviewed and merged using the Stash pull-request feature.

The tags are used to produce monthly builds or releases. As the name indicates, monthly builds are schedule-driven. They provide a certain pace for the developers. They are also installed on CernVM-FS and provide well-defined code versions for users and developers. Besides a successful compilation, no quality requirements are checked for monthly builds. In contrast, releases are feature-driven and a more thorough quality assessment is performed. The set of requested features and checks is defined in a consensus between users and developers.

Virtual and in-person meetings are a key component for the exchange of information and the coordination of activities. Progress and problems in the daily work are discussed informally in a weekly developers meeting. The sub-groups often have dedicated (bi-weekly) video meetings. The tracking group also organizes face-to-face meetings two or three times a year. A good opportunity to discuss software topics in person are the one-week software and computing workshops which are held at KEK in autumn and at one of the other collaborating institute in spring. The software sessions at the collaboration meetings are often joint with other groups to foster the information flow in both directions.

Finally we have established a regular software quality shift. Each week one collaboration member is tasked to help improve the software quality. The responsibilities include checking for failing builds, following up on orphaned issues and outdated branches and check documentation and examples. This shift both serves as a help for the existing developers and also as a means to introduce new developers to the software.

4. Summary

Large software projects are a challenge and distribution of the Belle II software developers all around the world does not improve the situation. We have established a set of tools that lets the developers focus on their main work and provides them with detailed feedback on the code quality

in a highly automated way. An organizational structure for collaboration and coordination is in place. Since the start of the project in 2008 about 100 different people have contributed, from undergraduate students to professors. The enthusiasm and devotion of developers will be the key for the success of the Belle II software project.

References

[1] Abe T *et al.* 2010 Belle II Technical Design Report *Preprint* arXiv:1011.0352
[2] Abashian A *et al.* 2000 The Belle Detector *Nucl. Instrum. Meth. A* **479** 117
[3] https://www.gnu.org/software/gcc
[4] https://clang.llvm.org
[5] https://software.intel.com/intel-compilers
[6] https://www.python.org
[7] http://www.scons.org
[8] https://www.cmake.org
[9] Brun R and Rademakers F 1997 ROOT: An object oriented data analysis framework, *Nucl. Instrum. Meth. A* **389** 81.
[10] Agostinelli F *et al.* [GEANT4 Collaboration] 2003 GEANT4: A Simulation toolkit, *Nucl. Instrum. Meth. A* **506** 250.
[11] Lange D J 2001 The EvtGen particle decay simulation package *Nucl. Instrum. Meth. A* **462** 152.
[12] https://git-scm.com
[13] https://www.atlassian.com
[14] https://subversion.apache.org
[15] http://astyle.sourceforge.net
[16] https://pypi.python.org/pypi/pep8
[17] https://pypi.python.org/pypi/autopep8
[18] https://www.stack.nl/~dimitri/doxygen
[19] https://www.sphinx-doc.org
[20] https://github.com/google/googletest
[21] http://valgrind.org
[22] http://cppcheck.sourceforge.net
[23] https://clang-analyzer.llvm.org
[24] https://buildbot.net
[25] P Buncic, C Aguado Sanchez, J Blomer, L Franco, A Harutyunian, P Mato and Y Yao 2010 CernVM a virtual software appliance for LHC applications *J. Phys.: Conf. Ser.* **219** 042003