Software Development at Belle II

2015 J. Phys.: Conf. Ser. 664 062024

(http://iopscience.iop.org/1742-6596/664/6/062024)

Journal of Physics: Conference Series **664** (2015) 062024 doi:10.1088/1742-6596/664/6/062024

Software Development at Belle II

Thomas Kuhr[1] and Thomas Hauth[2] for the Belle II Software Group

[1]Ludwig-Maximilians University Munich, Faculty of Physics, Excellence Cluster Universe, Boltzmannstr. 2, 85748 Garching, Germany
[2]Karlsruhe Institute of Technology, Institut für Experimentelle Kernphysik, Wolfgang-Gaede-Str. 1, 76131 Karlsruhe, Germany

E-mail: [1]Thomas.Kuhr@lmu.de, [2]Thomas.Hauth@kit.edu

Abstract. Belle II is a next generation B-factory experiment that will collect 50 times more data than its predecessor Belle. This requires not only a major upgrade of the detector hardware, but also of the simulation, reconstruction, and analysis software. The challenges of the software development at Belle II and the tools and procedures to address them are reviewed in this article.

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 in 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. Challenges

While large parts of the Belle software were written by KEK staff members, the Belle II software development is distributed around the world and relies on contributions from students and staff members. The regional distribution, as illustrated in Table 1, is one of the challenges. People who have not met in person and may be separated by several time zones must work together. With developers in Asia, Australia, Europe, and America (including Hawaii), it is impossible to find meeting times that are convenient for everybody. The different cultural backgrounds and language problems make communication even harder.

Also the large variation in programming skills and experiences of developers are a challenge for a project that has to make sure that all parts fit together. In addition personal preferences and coding styles are often a subject of discussions. Several tools and organizational measures

Table 1. Number of active Belle II software developers per country with the total count of their commits during one year.

Country	Active Developers	Commits Last Year
Australia	6	64
Austria	3	384
Canada	2	13
Czech Rep.	3	101
Germany	24	2424
Italy	6	163
Japan	17	759
Korea	4	26
Mexico	1	3
Poland	2	13
Russia	2	16
Slovenia	5	246
Taiwan	1	1
Turkey	1	2
US	7	257

are employed to allow the developers to focus on their main work and, at the same time, keep them conscious of the overall goal of providing an easy-to-use, robust, and powerful software suite to the Belle II collaboration.

3. 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 4.7 [3], clang 3.4 [4], and the Intel compiler version 14 [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 [6], a powerful and frequently used 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. Basically 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 built with a custom Makefile. 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 code of the Belle II software is maintained in a central Subversion [12] (SVN) repository at KEK and can be displayed via a ViewVC [13] code browser. Alternatives to Subversion, like Git [14], were considered, but it was decided to keep SVN for the following reasons. The philosophy of a central repository with one main development branch discourages divergent developments and it allows to enforce policies on commits, like style rules, via hook scripts. Moreover, it is easier to understand and use than Git, in particular for less experienced developers, but still provides sufficient features. More advanced developers, who are familiar with Git, can still benefit from its more powerful features by using git-svn.

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 SVN repository. The 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 was used for Python with the PythonTidy [17] tool, but abandoned recently because of unsatisfactory formatting results. Now we use pep8 for the style check and autopep8 [18] for the formatting, although there are cases where autopep8 is not able to produce a pep8 compliant result. As the code style is a matter of personal taste it is often a topic of passionate discussions.

Doxygen [19] was chosen as a tool for the documentation of the source code. Although there are probably better documentation tools for Python code it allows to use the same tool for C++ and Python code. Documentation that is beyond the scope of Doxygen is maintained in a TWiki [20]. A Redmine issue tracker [21] was set up 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 applied for the software quality control. googletest [22] can be used for unit tests. The framework can automatically run steering files that are located in a test folder and check whether the log output is identical to the expected one. A check for memory management issues is executed with Valgrind [23].

Finally, we have developed a framework for regression testing. It executes, either locally or on a batch system, a set of scripts that can be added to the SVN repository together with the other code. 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 together with ntuples for monitoring numerical values. XML headers in the scripts provide meta information like the dependencies between the scripts. The plots in all ROOT files are collected and displayed on a web page. The results from older revisions of the code are overlayed 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. A (significant) deviation from the reference is indicated by a yellow (red) plot frame. Fig. 1 shows an example of a validation plot display.

For an efficient and effective software quality control it has to be highly automated. We use the Buildbot tool [24] for this task. It is very flexible and it fits well into our environment because it uses Python for its configuration. Our setup consists of one server and four slaves with different operating systems: SL5, SL6, Ubuntu 14.04, and a 32bit version of Ubuntu 14.04. Furthermore, there are additional slaves at two sites, KEK and KIT. Several actions are triggered either by commits or at given times:

- On commits an incremental build is started and failures of compilation or tests are reported by email to the committer. This catches, for example, cases where the developer forgot to

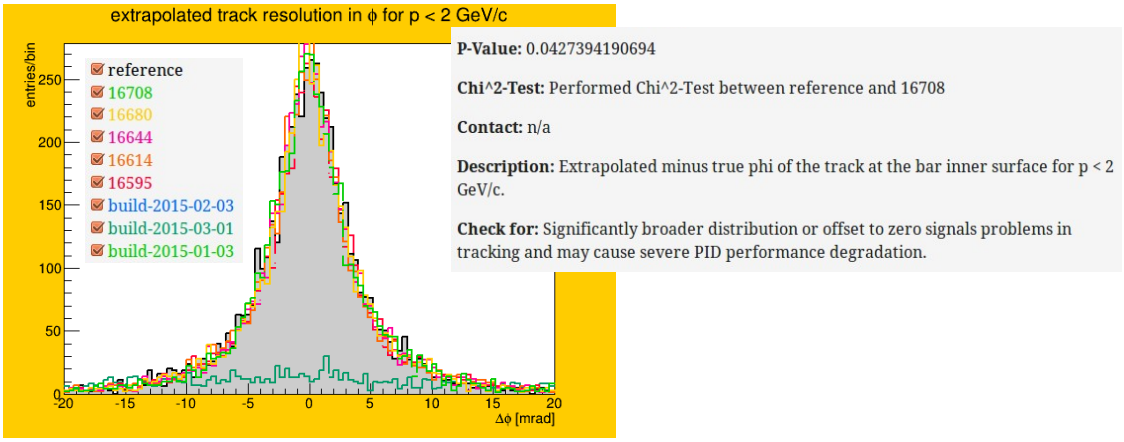

Figure 1. An example of a validation plot (see text for more details).

include a new file in the commit.

- Commits also trigger full builds from scratch on four different 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.
- Each night a full build with different compilers is executed. Furthermore, cppcheck [25] 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. Fig. 2 shows an example. It contains links to log files and locations of problematic code.
- We also use the Buildbot to update the installation of setup scripts, externals, and releases at sites.
- Whenever a new externals version is tagged, it is automatically compiled on the four slaves with different operating systems and tarballs of the binaries are made available for download.
- Monthly builds are automated such that the latest tagged versions of the parts of the code are collected and built. In case of a failure this procedure is repeated the next day. If the build is successful the code is committed to SVN and the new version announced on a mailing list together with a report of changes that are provided by the responsible persons on a TWiki page.

The full list of so-called builders as shown on the Buildbot web interface can be seen in Fig. 3.

4. 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.

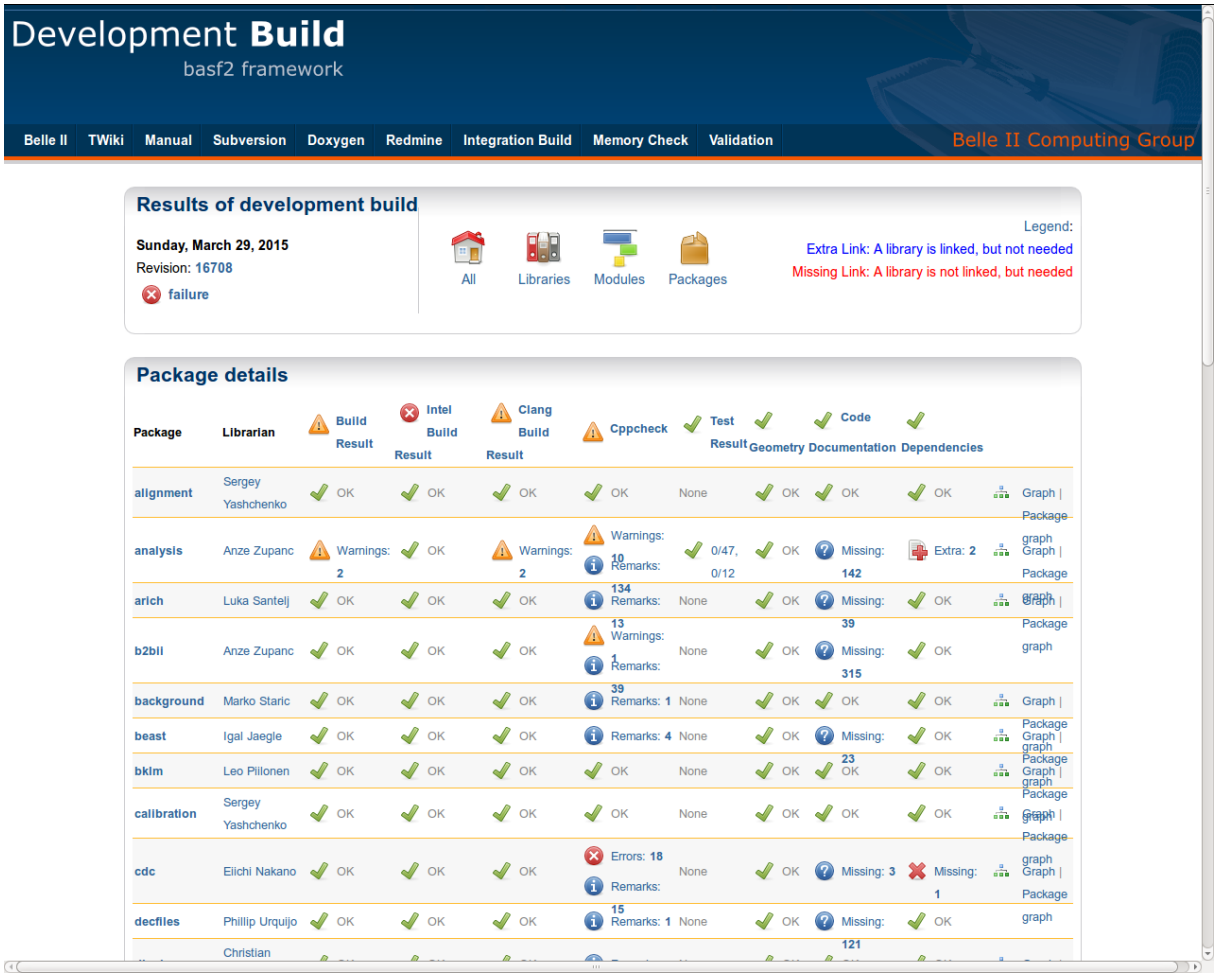

Figure 2. A screenshot of the nightly build results web page.

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.

The source code 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 can control who has write access to the package and has the right to make a tag of the package.

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 the central computing system at KEK 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

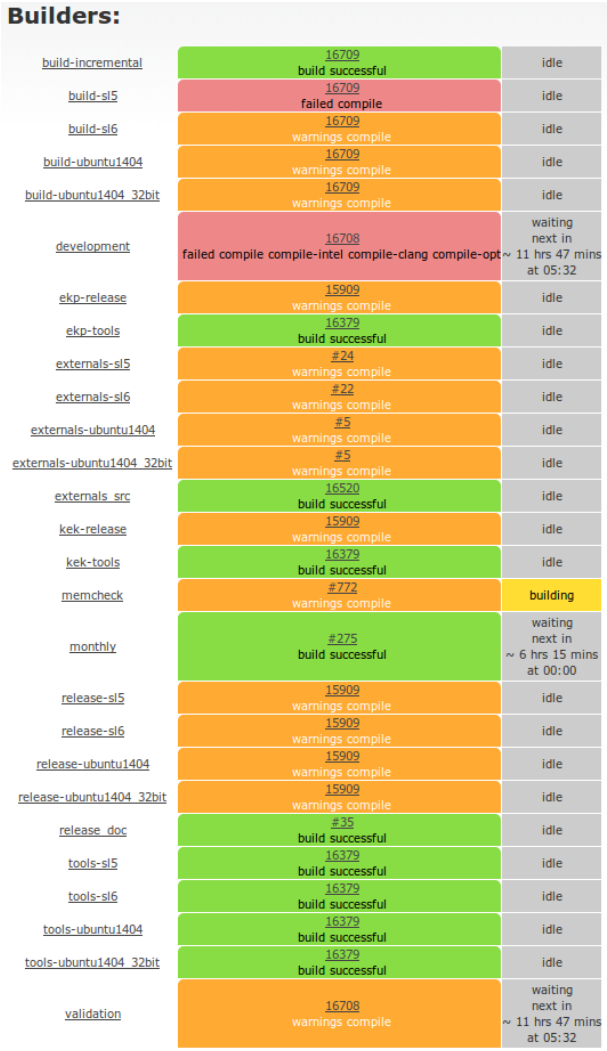

Figure 3. The list of automated builds for the Belle II software.

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.

5. Summary

Large software projects are a challenge and the distribution of the Belle II software developers all around the world does not make it easier. 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.

6. References

[1] Abe T *et al.*, arXiv:1011.0352 [physics.ins-det]
[2] Abashian A *et al.*, *Nucl. Instrum. Meth.* A **479** 117
[3] https://www.gnu.org/software/gcc/
[4] http://clang.llvm.org/
[5] https://software.intel.com/intel-compilers/
[6] https://www.python.org/
[7] http://www.scons.org/
[8] http://www.cmake.org/
[9] R. Brun and F. Rademakers, Nucl. Instrum. Meth. A **389** (1997) 81.
[10] S. Agostinelli *et al.* [GEANT4 Collaboration], Nucl. Instrum. Meth. A **506** (2003) 250.
[11] D. J. Lange, Nucl. Instrum. Meth. A **462** (2001) 152.
[12] https://subversion.apache.org/
[13] http://viewvc.org/
[14] http://git-scm.com/
[15] http://astyle.sourceforge.net/
[16] https://pypi.python.org/pypi/pep8
[17] https://pypi.python.org/pypi/PythonTidy/
[18] https://pypi.python.org/pypi/autopep8
[19] http://www.stack.nl/ dimitri/doxygen/
[20] http://twiki.org/
[21] http://www.redmine.org/
[22] https://code.google.com/p/googletest/
[23] http://valgrind.org/
[24] http://buildbot.net/
[25] http://cppcheck.sourceforge.net/