**Noname manuscript No.**
(will be inserted by the editor)

# The Belle II Core Software

**N. Braun · T. Hauth · T. Kuhr · C. Pulvermacher · M. Ritter**

Received: date / Accepted: date

**Abstract** Modern particle physics experiments create huge amounts of data. Sophisticated algorithms for simulation, reconstruction, and analysis are required to fully exploit the potential of this data. We describe the core components of the Belle II software that are the foundation for the development of complex algorithms and an efficient processing of data.

## 1 Belle II Analysis Software Framework

### 1.1 Code Structure

The Belle II software is organized in three main parts, the *tools*, the *externals*, and the Belle II Software Analysis Framework *basf2*.

#### *1.1.1 tools*

The tools are a collection of shell and python scripts for the installation and setup of the externals and basf2. The tools themselves are set up by sourcing the script `b2setup`. This script identifies the type of shell and then sources the corresponding sh or csh type setup shell script. The setup shell script adds the tools directory to the PATH and PYTHONPATH environment variables, sets some Belle II specific environment variables, defines functions for the setup or configuration of further software components, and checks whether a newer version of the tools is available. A pre-defined set of directories is searched for files containing site specific configurations. The Belle II specific environment variables have the prefix `BELLE2` and contain information like repository locations and access methods, software installation paths, and software configuration options.

Two shell scripts take care of the installation of a given version of externals or basf2 releases. Usually they just download and unpack tarballs of precompiled binaries for the given operating system. If no binary is available, the source code is checked out and compiled. Each version of the externals and basf2 releases is installed in a separate directory named after the version. In particular for the compilation of the externals we rely on a few basic tools, like `make` or `tar`, and development libraries to be installed on the system. The tools contain a script that checks whether these dependencies are fulfilled and can install the missing ones.

The command `b2setup` sets up the environment for a basf2 release version that is given as argument. It automatically sets up the externals version which has to be used for this release. The version is identified by the content of the `.externals` file in the release directory. An externals version can be set up independently of a basf2 release with the `b2setup-externals` command. The version dependent setup of the externals is managed by the script `externals.py` in the externals directory. Externals and basf2 releases can be compiled in optimized or debug mode. In addition basf2 supports the compilation with the clang or intel compilers. Different subdirectories are used for the libraries and executables compiled with the different options. These options can be selected with the `b2code-option` and `b2code-option-externals` commands. The commands that change the environment of the current shell are implemented as functions for sh type shells and as aliases for csh type shells.

N. Braun · T. Hauth · C. Pulvermacher
Karlsruhe Institute of Technology, Karlsruhe, Germany

T. Kuhr · M. Ritter
Ludwig-Maximilians-Universität, München, Germany

The tools also support the setup of an environment for the development of basf2 code. The `b2code-create` command clones the basf2 git repository and checks out the master branch. The environment is set up by executing the `b2setup` command without arguments in the local release directory. If a developer wants to work only on one top-level directory of the code, called package, and take the rest from a centrally installed release, the `b2code-create` command can be used with the version of this release as an additional argument which is stored in the file `.release`. The sparse checkout feature of git is used to get a working directory without checked out code. Packages can then be checked out individually with the `b2code-package-add` command. The `b2setup` command sets up the environment for the local working directory and the central release. Further tools for the support of the development work are described in Section 1.2.

To make it easier for users to set up an environment for the development of analysis code and to encourage them to store it in a git repository, the tools provide the `b2analysis-create` command. It requires a basf2 release version as argument and creates a working directory attached to a git repository on a central Belle II server. The basf2 release version is stored in a `.analysis` file and used by the `b2setup` command for the setup of the environment. The `b2analysis-get` command provides a convenient way to get a clone of an existing analysis repository.

The tools are supposed to be able to set up different versions of basf2 and externals and thus must be independent of them. By now this is reasonably well achieved. The last incompatible change was when we moved some binary code, like gcc or python, from the tools to the externals. In the beginning we included them in the tools because the same versions were shared among various basf2 and externals version. But their installation and update was a time consuming and error prone process for many users. The current script-only solution is more robust. When we migrated our code repository from subversion to git the tools were updated more frequently, but now they have become stable and updates are done rarely. One of the challenges in the development of the tool was to cope with the different shell types and various user environment settings.

### *1.1.2 externals*

The third-party code on which we rely (besides the operating system) is bundled in the so-called externals. They contain basic tools, like gcc, python3, or bzip2, and HEP specific software, like ROOT [1], GEANT4 [2] or EvtGen [3]. Some packages, like LLVM or Valgrind, are optional and not included in the compilation of the externals by default. The number of external products has grown over time and has reached a count of about 60 plus 90 python packages by 2018.

The instructions and scripts to build the externals are stored in a git repository. We use a Makefile with specific commands for the download, compilation, and installation of each of the external packages. A copy of the downloaded installation files are kept on a Belle II web server to still have them available if the original source disappears. It also provides redundancy for the download. The integrity of the downloaded files is checked with `sha256sum`.

The libraries, executables, and include files of all external packages are each collected in a common directory. For the external software that we might want to include in debugging efforts we build a version with debug information and an optimized version.

The compilation of the externals is a very time consuming task and thus annoying for users. Moreover, users sometimes experienced problems because of specific configurations of their systems. While we tried to minimize these issues, the largest improvement for the users and those supporting them was achieved by providing pre-compiled binary versions. We use docker to compile the externals on various systems, currently Scientific Linux 6, Enterprise Linux 7, Ubuntu 14.04, and the Ubuntu versions from 16.04 to 18.04. A command provided by the tools conveniently downloads and unpacks the selected version of the externals in the right directory.

Because the absolute path of an externals installation is abritrary, we invested significant effort into making the externals location independent. First studies to move from the custom Makefile to spack [4] were done with the aim to profit from community solutions, but relocateability of the build products remains an issue.

### *1.1.3 basf2*

The Belle II specific code is organized in directories, called packages. As of 2018 there are about 40 packages, including for example the core framework, a package for each detector component, the track reconstruction code, and the analysis tools. Each package has one or two librarians who are responsible for the code in their directory.

The code is written in C++ and header and source files are kept in `include` and `src` subdirectories, respectively. By default one shared library is created per package and installed in a `lib` directory that is included in the library path. A special treatment of the code is

achieved by placing it in one of the following subdirectories

- `modules`: The code is compiled in a shared library and installed in a `module` directory so that it can be dynamically loaded by basf2.
- `tools`: C++ code is compiled in an executable and installed in a `bin` directory that is included in the path. Executable scripts are symlinked to this directory.
- `dataobjects`: This contains the classes that can be stored in output files. The code is linked in a shared library with `_dataobjects` suffix.
- `scripts`: Python scripts are installed in a directroy that is included in the python path.
- `data`: All files are symlinked to a common `data` folder.
- `tests`: Unit and script tests (see Section 1.2).
- `validation`: Scripts and reference data for validation plots (see Section 1.2).
- `examples`: Example scripts that illustrate features of the package.

Users of basf2 usually work with centrally installed versions of basf2. They are provided on CVMFS [5], but users can also install pre-compiled binaries on their local systems with the `b2install-release` command. If no pre-compiled version is available for their system, the command will compile the requested version from source.

## 1.2 Development Infrastructure and Procedures

The basf2 code is maintained in a git repository and we use bitbucket [6] to manage pull requests. The ability to review and discuss code changes in pull requests before they are merged to the main development branch is a significant improvement in the development process compared to the previous workflow based on subversion. It helps the authors to improve the quality of their code and allows the reviewers to get a broader view of the software. We also profit from the integration with the jira [7] ticketing system to better track and plan the development work.

Developers obtain a local copy of the code with the `b2code-create` command provided by the tools. They can choose to take most of the code from a central release and check out only selected packages. The build system based on SCons takes care of the dependencies between files in the central release and local working directory.

Although cmake was also considered as build system, Belle II decided to use SCons [8] for this purpose. It has the advantage that the build process is a one-step procedure and the build configuration is written in python, a language anyhow used for the basf2 steering files. With some tuning, we could reduce the time SCons needs to determine the dependencies before starting the build. The build system is set up in a way that developers and users usually do not have to care about it. They only have to place their code in the right directories as described in the previous section. One exception is the definition of linked libraries which can be done with a three lines `SConscript` file.

We have implemented an access control for commits to the master branch using a hook script on the bitbucket server. Librarians, identified by their user names in a `.librarians` file in the package dircetory, can directly commit code in their package. They can give this right also to others by adding their user names to a `.authors` file. Since we migrated from subversion to git we allow all Belle II members to commit code to any package in feature or bugfix branches. The merging of these branches to the master has to be approved by the librarians of the affected packages.

To achieve some conformity of the code, we have established coding conventions, but we have to largely rely on developers and reviewers to follow them. We enforce a certain style of the code which underlines that it belongs to the collaboration and not the individual developer. The AStyle tool [9] is used for C++ code and pep8 [10] and autopep8 [11] for python code. As some developers feel strongly about the code formatting it is important to keep the hurdles to follow the rules and thus frustration low. Therefore we provide the `b2code-style-check` tool to print style violations and the `b2code-style-fix` tool to automatically fix them. The style conformity is checked by the stash server hook. It also rejects files larger than 1MB to prevent an uncontrolled growth of the repository size which is more of an issue with git than with subversion. To provide feedback to developers as early as possible and avoid annoying rejections when commits are pushed to the central repository we have implemented the checks of access rights, style, and file size also in a hook for commits to the local git repository.

To facilitate test driven development, unit tests can be implemented in each package using gtest [12]. All of them are executed with the `b2test-units` command. Furthermore test steering files in all packages can be run with the `b2test-scripts` command. It compares the output to a reference file and complains if it differs or the execution fails. The unit and steering file tests are executed by the bamboo [13] build service whenever changes are pushed to the central repository. Branches can only be merged to the master if all tests succeed.

The tests are also executed by a buildbot [14] continuous integration system which compiles the code with the gcc, clang, and intel compilers and informs the authors of commits about new errors or warnings. In addition, the buildbot runs cppcheck, a geometry overlap check, doxygen and sphinx documentation generation, and a valgrind memory check each night. The results are displayed on a web page and the librarians are informed by email about issues in their package. A detailed history of issues is stored in a mysql database and displayed on a web page, too. The page also shows the evolution of the execution time, output size, and memory usage of a typical job.

A quality control at a higher level is provided by the validation framework. It executes scripts in a folder named `validation` of packages to generate simulated data files and produce plots from them. The validation then spawns a web server that shows the plots and compares them with versions from previous validation runs and a reference. A shifter, whose task is to monitor the software quality, checks the nightly produced validation plots for regressions.

As a regular motivation for the libarians to review the changes in their package we do monthly builds. For a monthly build we require all librarians to agree on a common commit on the master branch. They signal their agreement using the `b2code-package-tag` command to create a git tag for the package at the selected commit. It asks for a summary of changes that are then included in the announcement of the monthly build. The procedure of checking the agreement, building the code, and sending the announcement is fully automated with the buildbot.

An extensive manual validation including the generation of larger samples is done before releasing major official versions of basf2. Based on these major versions, minor or patch releases that require less or no validation effort can be made. In addition light basf2 releases containing only the packages required to analyze mini DST (mDST, see Section 1.5) data can be made by the analysis tools group convener. This allows for a faster release cycle of analysis tools than the full software stack. Releases are triggered by pushing a tag to the central repository. The build process on multiple systems and the installation on CVMFS is then automated.

In general, a guiding principle for the development infrastructure and procedures is to keep the thresholds to use and contribute to the software as low as possible and at the same time strengthen the mindset of a common, collaborative project and raise awareness of code quality issues. This includes early feedback, e.g. about style rule violations on commits to the local git repository, and the idea to not bother people with things that can be done by a computer, like the correction of style rule violations. In this way, users and developers can focus on their actual work and use their time more efficiently.

### 1.3 Modules, Parameters, and Paths

The basf2 framework executes a series of dynamically loaded modules. The selection of modules, their configuration, and their sequence are defined via a python interface, see Section 2.1.1.

A module is written in C++ or python and derived from a `Module` base class that defines the following interface methods:

- `initialize()`: Called before the processing of events to initialize the module.
- `beginRun()`: Called each time before a sequence of events of a new run is processed.
- `event()`: Called for each processed event.
- `endRun()`: Called each time after a sequence of events of the same run is processed.
- `terminate()`: Called after the processing of all events.

Several flags can be set in the constructor of a module, for example to indicate that it is capable of running in parallel processing mode. The constructor should also set a module description and define module paramters that can be displayed on the terminal with the command `basf2 -m`.

A module parameter is a property that can be set via the python interface and then used in the module execution. It can be of basic type or a list thereof. Each parameter has a name, a description, and an optional default value.

The sequence in which the modules are executed is stored in a `Path` class. Multiple paths can be connected using conditions on an integer result value set in a module `event()` method. The processing of events is initiated by calling the `process()` method with a path as argument. The framework checks that there is exactly one module that sets the event numbers. It also collects information about the number of module calls and their execution time. This information can be printed after the event processing or saved in a root file.

Log messages are managed by the framework and can be passed to different destinations, like the terminal or a text file, via connector classes. Methods for five levels of log messages are provided:

- `FATAL`: For situations were the program execution cannot be continued.
- `ERROR`: For things that went wrong and must be fixed. If an error happens during initialization the event processing is not started.

- `WARNING`: For potential problems that should not be ignored and only accepted if understood.
- `INFO`: For informational messages that are relevant to the user.
- `DEBUG`: For everything else, in particular debug information that is useful for developers. An additional integer debug level is used for debug messages so that the amount of debug messages can be controlled.

The log and debug level can be set globally, per package, or per module.

## 1.4 Data Store and I/O

### *1.4.1 Data Store*

Modules exchange data via a globally accessible interface to objects or arrays of objects, the so-called Data Store. Objects or arrays of objects (stored in a TClonesArray) are identified by name which by default corresponds to the class name. The convention is to append an "s" to the class name for the name of arrays. Users can however choose a different name to allow different objects of the same type simultaneously. Objects in the Data Store can have either permanent or event level durability. In the latter case the framework takes care of clearing them before a new event is processed.

Different arrays of objects in the Data Store can have weighted many-to-many relationships between their elements. For example, a higher level object will usually have relations to all lower level objects which were used to create it. Each relation carries a floating point weight. The relationship information is stored in a separate object so no direct pointers are placed in the objects themselves. This allows to strip parts of the event data without affecting data integrity: If one side of a relationship is removed the whole relation will be dropped. The relations are implemented by placing a special `RelationArray` in the DataStore which knows the names of the arrays it relates as well as the indices and weights of the related entries. The default name for this object is the name of the two arrays connected by "To".

The interface to objects in the Data Store is implemented in the templated classes `StoreObjPtr` for single objects and `StoreArray` for arrays of objects, both derived from the common `StoreAccessorBase` class. They are constructed with the name identifying the objects or without any argument in which case the default name is used. The access to the objects is type safe and transparent to the event-by-event changes of the Data Store content. To make the access efficient, the `StoreAccesorBase` translates the name to a pointer to an `DataStoreEntry` object in the Data Store on first access. The `DataStoreEntry` object is valid for the whole duration of the job and contains a pointer to the currently valid object which is automatically updated by the Data Store. An access to an object in the Data Store thus requires only on first access an expensive string search and then just a double dereferencing of a pointer on subsequent accesses.

The usage of relations is simplified by deriving the objects in a Data Store array from `RelationsObject`. It provides methods to directly ask an object for its relations to, from, or with (ignoring the direction) other objects. Non-persistent data members of `RelationsObject` and helper classes are used to make the relations lookup fast by avoiding regeneration of information that was already obtained earlier.

We also provide an interface to filter, update or rebuild relations when some elements are removed from the DataStore. It is also possible to copy whole or partial arrays in the DataStore where optionally a new relation to the original array can be created or the existing relations on the original array are copied.

### *1.4.2 I/O*

We use ROOT for persistency. This implies that all objects in the Data Store must have a valid ROOT dictionary, The `RootOutputModule` stores the content of the Data Store of permanent and event durability in two separate `TTree`s with a branch for each Data Store entry. A selection of branches, the file name, and some tree configurations can be specified using module parameters. The corresponding module for reading root files is the `RootInputModule`.

The `RootOutputModule` takes care of writing an object named `FileMetaData` to the permanent durability tree of each output file. It contains a logical file name, the number of events, information about the covered experiment/run/event range in the output file, the steering file content, and further information about the file creation. The file meta data also contains a vector of logical file names of the input files, called parents.

This information is used for the index file feature. A `RootInputModule` can be asked to not only load a file, but also its ancestors up to a level given as parameter. A file catalog in XML format, created by the `RootOutputModule`, is used to translate logical to physical file names for the ancestor files. The unique event identifier is then used to load the correct event. With the index file feature one can produce a file containing only `EventMetaData` objects (see next section) of selected events and then use this as input file to access

the selected events in its parents. However, the usual structure of trees in root files is not optimal for sparse event reading. Further use cases are to add objects to an existing file without copying its full content or to access lower level information of individual events for display or debug purposes.

The high-level trigger uses a custom output format with a sequence of serialized root objects to limit the loss of events in case of crashes. The files in this format are not stored permanently, but converted to standard root files before they are written to mass storage.

### 1.5 Event Data Model

The Data Store implementation makes no assumption about the event data model. It can be flexibly chosen to match the specific requirements. In basf2, the full event data model is not explicitly defined, but a result of the creation of objects in the Data Store by the executed modules. The only mandatory component is the `EventMetaData` object. It uniquely identifies an event by event, run, and data taking period number, called experiment number. In addition, it contains a unique production identifier to distinguish simulated events with the same event, run, and experiment number. Further data members are the time when the event was recorded or created, an error flag indicating problems in data taking, an optional weight for simulated events, and the logical file name of the parent file that is needed by the index file feature.

The format of the raw data is defined by the detector readout. Unpacker modules for each detector component can convert the raw data to digits, the output format of the simulation. During the event simulation, the energy deposits are stored as detector specific `SimHits`. The use of a common base class for `SimHits` allows for a common framework to add energy depositions from simulated background to that of simulated signal processes, called background mixing.

The output of the reconstruction consists mainly of detector specific objects. The `RecoTrack` class is used to manage the pattern recognition and track fitting across multiple detectors. It allows to add hits to a track candidate and is interfaced to GenFit [15,16] for the determination of track parameters.

The data format for analyses, called mini data summary table (mDST), is explicitly defined in the steering file function `add_mdst_output`. It consists of the following classes

- `Track`: The object representing a reconstructed track of a charged particle contains references to track fit results for multiple mass hypotheses and a quality indicator that can be used to suppress fake tracks.
- `V0`: Candidates of $K_S^0$ and $\Lambda$ decays and of converted photons are provided by references to pairs of positively and negatively charged tracks and track fit results.
- `TrackFitResult`: The results of track fits for a given particle hypothesis are five helix parameters, their convariance matrix, a fit p-value, and the pattern of layers with hits in the vertex detector and drift chamber.
- `PIDLikelihood`: Each track has a related object that stores the likelihoods for being an electron, muon, pion, kaon, proton or deuteron for each detector providing particle identification information.
- `ECLCluster`: For reconstructed clusters in the electromagnetic calorimeter the energy and position measurements and their correlations are stored together with shower shape variables. If an extrapolated track is matched to a cluster this is represented by a relation.
- `KLMCluster`: Reconstructed clusters in the KLM detector that are not matched to tracks are considered candidates for $K_L^0$ and provided a position measurement and momentum estimate with uncertainties.
- `KlId`: Candidates for $K_L^0$ particles are represented by relations to KLM and ECL clusters where the relation weight stores the particle identification information.
- `TRGSummary`: The information about level 1 trigger decisions before and after prescaling are stored in bit patterns.
- `SoftwareTriggerResult`: The decision of the high-level trigger is provided via a map of trigger names to trigger results.
- `MCParticle`: For simulated events, the information about simulated particles is provided in addition. The momentum, production and decay vertex, relations to mother and daughter particles, and information about hit detector components is stored for each particle. Relations are created when simulated particles are reconstructed as tracks or clusters.

The average size of an mDST event is a critical parameter for the disk storage requirements and for the analysis turn-around time. Therefore, the mDST content is strictly limited to information that is required by general physics analyses. No raw data information is allowed. For detailed detector or reconstruction algorithm performance studies as well as for calibration tasks a dedicated format, called cDST for calibration data summary table, is provided.

## 2 Central Services

### 2.1 Python Interface and Jupyter Notebooks

#### *2.1.1 Python Interface*

To apply the functionality described in Section 1 to a data processing task – at the most basic level arranging appropriate modules into a path and starting the event processing – basf2 provides a Python interface. Most commonly, users perform tasks using Python scripts (called "steering files" in this context), but interactive use is also supported. The following listing shows a minimal example for the former, while Section 2.1.2 discusses applications for the latter.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Import the python interface to the Belle II ↩
    framework
import basf2 as b2

# Create a path
main = b2.create_path()

# Create a module of type EventInfoSetter,
# set the parameter evtNumList to produce events↩
    with numbers
# 0 to 99, and add the module to the main path
main.add_module('EventInfoSetter', evtNumList↩
    =[100])

# Start the processing of the modules in the ↩
    main path
b2.process(main)
```

Key benefits in using Python as a configuration language, compared for example with the custom language used at Belle, are the general prevalence and the easy to understand syntax so that new users can rather quickly use the framework successfully. Furthermore python provides the power of a modern script language including access to various libraries. This is exploited, for example, to build a meta-framework in python for performing typical analysis tasks in a user-friendly way. The docstring feature of python is used to generate documentation web pages with sphinx [17].

To make the framework features available in python we use boost.python [18]. While steering files can be executed by passing them directly to the python interpreter, we provide the `basf2` executable which adds framework specific command line arguments. Among those are options to print versioning information, list available modules and their description, and specify input or output file names.

Besides the implementation of modules in C++, the framework allows to execute modules written in python. This makes it even easier for users to write their own module code as it can be part of the steering file. It can also facilitate rapid prototyping. With a few exceptions for tasks that are not performance critical the modules provided by the framework are written in C++ to profit from advantages of compiled code.

Access to the Data Store is provided in the python code by classes resembling the `StoreObjPtr` and `StoreArray` interfaces. In an equivalent way, interface classes provide access conditions data. The python integrations of these classes are implemented with PyROOT [19].

A feature that facilitates development and debugging is the possibility to interrupt the event processing and present an interactive python prompt. In the interactive session based on IPython [20] the user can inspect or even modify the processed data.

#### *2.1.2 Jupyter Notebooks*

Today's analyses for high-energy physics (HEP) experiments are mostly based on the execution of small scripts written in Python or ROOT macros which call complex compiled algorithms in the background for processing a large amount of data. Jupyter notebooks allow to develop Python-based applications, which bundle code, documentation and results, e.g. plots and provide an enriched working environment like a browser-based frontend to an interactive Python session, which can be hosted centrally on a high-performance computing cluster. Additional features include syntax highlighting and tab-completion as well as integration with data-science tools like ROOT, matplotlib [21] or pandas [22].

The implemented jupyter integration into basf2 helps to simplify the process of creating and processing module paths within jupyter notebooks – which is a natural next step after the already extensive integration of python into basf2. The package for the interplay between Jupyter and basf2 is encapsulated into a more general hep-ipython-tools project [23], which can also be used with the framework code of other experiments.

The processing of one or more paths is decoupled into an abstract *calculation* object, which plays well with the interactivity of the notebooks, because multiple instances of this calculation can be started and monitored, while continuing the work in the notebook. Abstracting the basf2 calculation together with additional interactive widgets and convenience functions for an easier interplay between juypter and basf2 not only improves the user experience but also accentuates the narrative and interactive character of the notebooks.

The decoupling of the calculations is achieved using the multiprocessing library and depends heavily on the ability to steer basf2 completely from the python process. Queues and pipelines are used from within the basf2 modules to give process and runtime-dependent

information back to the notebook kernel. The interactive widgets are created using HTML and javascript and display information on the modules in a path, the content of the data store or the process status and statistics.

### 2.2 Parallel Processing

Since several years, the frequency of CPUs and with it the performance of a single core, is not increasing significantly any more. Instead the processing power of a CPU has gained by an increased the number of cores. To efficiently use modern CPU architectures it is thus essential to be able to run applications on many cores.

A trivial approach would be to run multiple applications, each using one core. The problem is that many other resources are shared. In particular the size of and the access to the memory can be a bottleneck. The amount of memory per core on typical sites used by HEP experiments has remained in the range of 2 to 3 GB for many years.

A more efficient shared usage of memory can be achieved by multi-threaded applications. The downside is that this imposes much higher demands and limitations on the code as it has to be thread safe. While the development of thread safe code can be assisted by libraries it requires a change in the style how code is written. Only very few Belle II members already have the skills to write thread safe code. To embark in a multi-threaded framework would require to educate of the order of hundred developers. Several of them only contribute with a small fraction of their time and it would be an enormous effort for them and the core team to keep them involved.

To nevertheless exploit multi core machines we have implemented a parallel processing feature where processes are started by fork. As the processes are running independently developers do not have to care about thread safety. Still we can significantly reduce the memory consumption of typical jobs because of the copy-on-write technology used by modern operating systems. A large portion of the memory is used for the detector geometry. Because it is created before the forking and does not change during the job execution the multiple processes share the same geometry representation in memory. Figure 1 illustrates the speedup of execution time with increasing number of parallel processes on a 16-core system. For both measured reconstruction scenarios, one with smaller $e^+e^-$ and one with larger $B\bar{B}$ collision events, the speedup is either equal or very close to the theoretical maximum speed-up. The minor loss in efficiency when going to more cores can be attributed to shared resources, like level-3 caches, used by all processing cores.

Each module can indicate to the framework whether it can run in parallel processing mode or not. Not parallel processing capable are in particular the input and output modules that read or write root files. As the input and output modules are usually at the beginning or end of a path, the framework analyzes the path and splits it into three sections. The first and last section are executed in a single process each. Only the middle section is executed in multiple processes. The beginning of the middle section is defined by the first module that can run in parallel processing mode. The next module that is not parallel processing capable defines the beginning of the third section.

To transfer the event data between the processes dedicated transmitter and receiver modules are added at the end or beginning of the sections. A transmitter module serializes the event data and writes it to a ring buffer in shared memory. A receiver module reads the event data and deserializes it so that it becomes available in the Data Store of the process.

This parallel processing scheme works well if the computational effort of the modules in the middle section dominates over the input, output, and (de)serialization load. For high throughput jobs with little computational demands the serialization and deserialization can impose a sizable penalty so that the multiple cores of a CPU are not optimally exploited. For typical Belle II reconstruction jobs and data sizes, we have measured that the input and output processes don't become a bottleneck at 20 concurrent processes which is well within the envelope of parallelism we currently foresee to employ during the online reconstruction or grid simulation and reconstruction.

### 2.3 Random Numbers

Belle II will need to generate very large samples of Monte Carlo to be able to achieve the targeted precision. We have to ensure that this production is not hindered by issues with the pseudorandom number generator (PRNG). A PRNG is a deterministic algorithm to generate numbers whose properties approximate the properties of random numbers while being completely deterministic. It has an internal state which determines both the next random number and the next internal state uniquely. If the internal state is known at some point, all subsequent random numbers can be reproduced.

For Belle II we chose xorshift1024* [24], a newer generation PRNG based on the Xorshift algortihm pro-

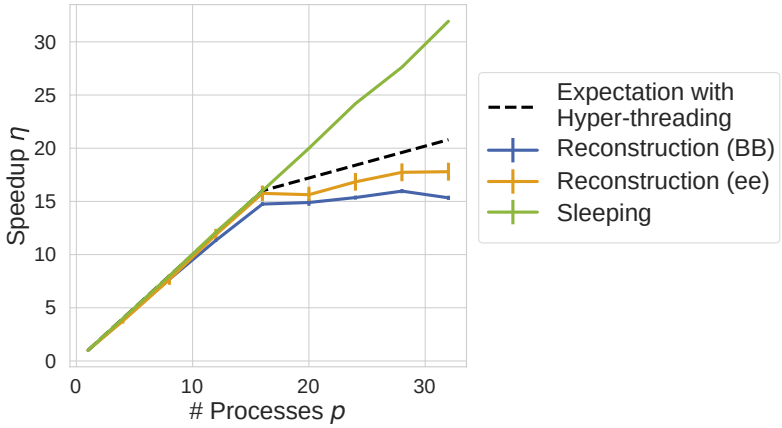


Fig. 1: Speed-up of parallel processing measured on a 16-core system for smaller $e^+e^-$ and larger $B\bar{B}$ collision events. As reference, the expected perfect scaling is plotted as the dotted line, assuming a 20% gain in the hyper-threading domain. Furthermore, the measured speedup when using sleep instructions instead of running the reconstruction algorithms is plotted in green.

posed by Marsaglia [25]. It generates 64bit random numbers with a very simple implementation, it features high speed, and passes all well-known statistical tests with an internal state of only 128 bytes (1024 bits). This PRNG is used consistently throughout the framework for all purposes from event generation to simulation down to analyis.

To make sure events are independent from each other we set the state of the random generator at the beginning of each event using an common, event-independent seed string together with information uniquely identifying the event. To minimize the chance for collisions between different events we calculate a 1024 bit SHAKE256 [26] hash from this information which we use as the generator state. This also allows us to use a common seed information of arbitrary length so any string can be used as random seed.

The small generator state also allows us to pass the random generator for each event along with the event data in parallel processing mode to achieve reproducibility independently of the number of worker processes.

### 2.4 Conditions Data

In addition to event data and constant values we have a number of settings or calibrations which can change over time but not on a per event level. These are called "conditions" and are stored in a central Conditions Database (CDB) [27]

Conditions are divided into separate "payloads". Each payload is one atom of conditions data and has one or more "intervals of validity" (IoV), the run interval in which the payload is valid. One complete set of payloads and their IoVs is then called a "global tag". There can be different global tags, for example for different versions of the calibrations. When a new global tag is created it is open and assignments of IoVs to payloads can be added or removed. Once a global tag is published it becomes immutable.

The CDB is implemented as a representational state transfer (REST) service. Communication is performed by standard HTTP using XML and JSON data. The CDB is by design agnostic to the contents of the payloads and only identifies them by name and revision number. The integrity of all payloads is verified using a checksum of the full content. Clients can query the CDB to obtain all payloads valid for a given run in a given global tag.

The choice of a standardized REST API makes the client implementation independent of the actual database implementation details and allows for a simple and flexible implementation of clients in different programming languages.

In addition to communication with the CDB we have implemented a local database backend which will read global tag information from a text file and use the payloads from a local folder. This allows to use the framework also without connection to the internet or if the CDB is not reachable, provided the necessary payloads have been obtained previously. Such a local database is created automatically in the working directory for all payloads that are downloaded from the server during a basf2 execution.

Multiple metadata and payload sources can be combined. By default global tags are obtained from the

central server and payloads from a dump to a local database on CVMFS. If a payload is not (yet) in the local database it is downloaded from the server. If the central database is not available the global tag is taken from the database on CVMFS.

#### *2.4.1 Access of Conditions Objects*

The software framework assumes that payload contents are serialized ROOT [1] objects but also direct access to the files is possible. User access to conditions objects is provided using two interface classes, one for single objects called `DBObjPtr` and one class for arrays of objects called `DBArray`. These classes reference payload objects, so-called `DBEntry` objects in a global store, the `DBStore`. Multiple instances of the interface class all point to the same object. It is identified by a name which is by default given by the class name. Access to the conditions objects is available in C++ and in Python where the class interface has been designed to be as close as possible to the already existing interface for event level data. Users familiar with the event level storage should have no problems accessing conditions data.

The interface classes always point to the correct payload objects for the current run, updates are transparent to the user. If the user needs to be aware when the object changes, they can either manually check for changes or register a callback function to be notified on change. Figure 2 visualizes the relations between all the entities.

The CDB only handles payloads at run granularity but the framework can transparently handle conditions changing inside of a run: If the payload is a ROOT object inheriting from the base class `IntraRunDependency` the framework will transparently update the conditions data on event granularity. Different specializations of `IntraRunDependency` can be implemented, for example changing the conditions depending on event number or time stamp.

#### *2.4.2 Creation of Conditions Data*

To facilitate easy creation of new conditions data, for example during calibration, we provide two additional classes `DBImportObj` and `DBImportArray` which also have a similar interface as `DBObjPtr` and `DBArray` but are meant to create new payloads. Users can just instantiate one of the creation classes, add objects to them and commit them to the configured database with a user supplied IoV. This includes support for intra run dependency. The possibility for a local file based database allows for easy preparation and validation of payloads as is needed during the calibration of the detector.

#### *2.4.3 Management of CDB Content*

To simplify the inspection and management of the CDB contents we provide a standalone command line client written purely in Python 3 using the excellent requests package [28]. It allows to list, create and modify global tags as well as inspect their contents. It can also be used to download a global tag for usage with the local database backend as well as uploading a previously prepared and tested local database configuration to a global tag.

### 2.5 Geometry and Magnetic Field

In Belle II we use the same detailed geometry description for simulation and reconstruction purposes which is implemented using the Geant4 [2] geometry primitives. A central service is responsible for setting up the complete geometry and each sub detector can register a "creator" which is responsible to set up the detector specific volumes as a separate component of the geometry.

All parameters for the geometry description of all sub detectors are provided by payloads in the conditions database. In the original implementation, which is still available, all parameters were obtained from one XML tree where the separate descriptions of all components is joined using XInclude [29] directives to allow the librarians full control over their parameters. When building the geometry from XML a service called "Gearbox" reads the whole tree using libxml2 [30]. It provides easy access to the XML document tree for the developers including unit conversion of values which have a "unit" attribute. New materials and their properties can also be defined directly in the XML files.

When building the geometry, the `Materials` provider will first search for all material definitions using XPath [31] expressions and create the corresponding Geant4 material objects. The Geometry service will instaniate all creators for a list of specified components and call them to build the geometry. Each creator gets passed a pointer to the `G4LogicalVolume` where all the volumes need to be placed and, in case the gearbox is used, a reference to the XML tree where the parameters for this component are to be found.

For the concretization of the XML description into conditions objects a special operation mode was implemented which calls all Creators with only the reference to the their XML parameters and asks them to create a payload from the given parameters. This allows to

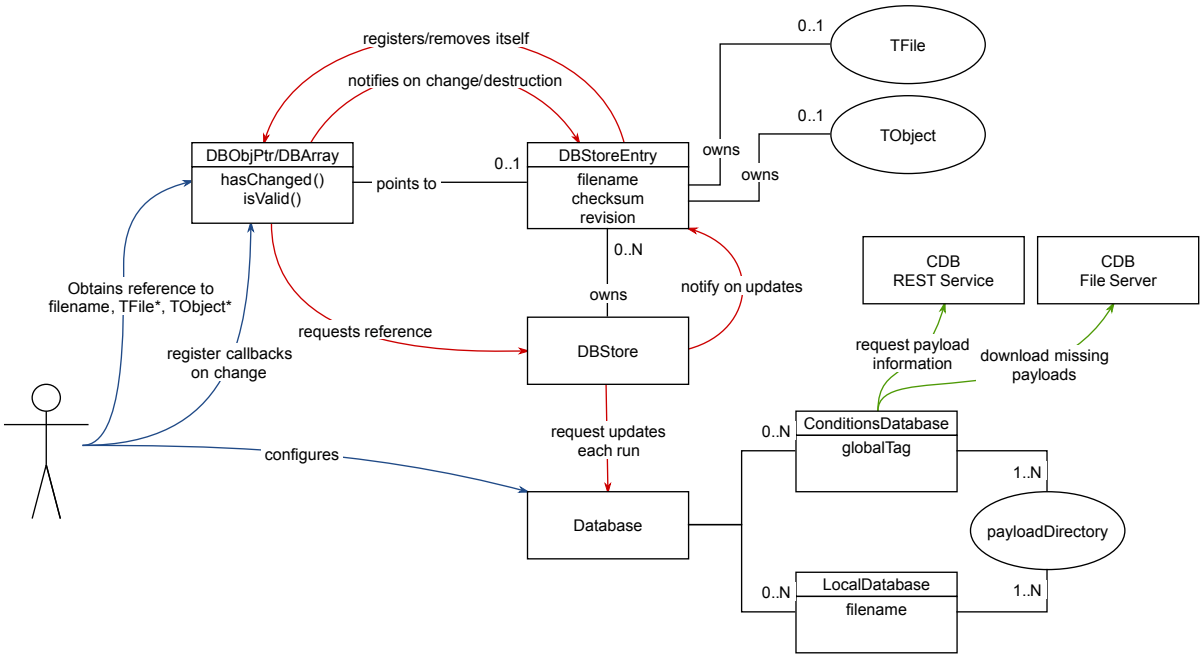


Fig. 2: Relations between all entities for the Conditions Database Client. The user usually only interacts with the `DBObjPtr` and `DBArray` objects and maybe configures the database sources (shown in blue). Everything else is handled transparently, including the communication with the CDB (shown in green).

edit the XML files to adapt the geometry description as necessary and test the changes locally. Once testing is complete a fixed set of database payloads can be created to be uploaded to the database.

### *2.5.1 Testing the Geometry Description*

Developing a working material description is quite cumbersome as usually complex construction drawings need to be converted into code placing separate volumes with their correct transformation. To assist the sub detector developers with this task we developed a set of tools in addition to the visualization tools already provided by Geant4.

First we run an automated overlap check which uses methods provided by Geant4 to check if any volume in the given tree of volumes has intersections with any of it's siblings or its parent. This is done by randomly creating points on the surface of the volume under question and check if this point is either outside the parent or inside any of the sibling volumes. This check is performed on a nightly basis and repeated with higher statistic prior to major releases or if large changes to the geometry are to be expected.

Second we provide a module to scan the material budget encountered when passing through the detector. This module will track non-interacting, neutral particles through the detector and record the amount of material encountered along the way. It can be configured to scan the material in spherical coordinates, in a two dimensional grid or as a function of the depth along rays in a certain direction. The output is a ROOT file containing histograms of the encountered material. These histograms can be created either by material or by detector component. Especially material distribution by component is a very useful tool to track changes to the material description, allowing us to show the difference in material for each update to the code or parameters for the material description.

### *2.5.2 Magnetic Field Description*

The magnetic field description for Belle II can also be loaded from XML file description as part of the geometry description. However the magnetic field does not actually create any volumes so once a database payload is created the magnetic field can be used independently of the geometry setup. This allows analysis jobs to just obtain the field values without a need to setup the full geometry.

The magnetic field creator supports a list of field components defined for different parts of the detector.

If more than one component is valid for the same region either the sum of all field values is taken or only one component is returned if it is declared as exclusive. We have implementations for constant magnetic field, 2D radial symmetric field map and full 3D field maps and some special implementations to recreate the accelerator conditions close to the beams. For normal simulation and analysis jobs we have a segmented 3D fieldmap with a fine grid in the inner detector region and a total of three coarse outer grids for the two endcaps and the outer barrel region.

## 3 Conclusions

Ten years of development work with emphasis on software quality have paid off. The Belle II collaboration has a reliable software framework that is easy to use and that makes it easy to develop algorithms for it. It fulfills the requirements for data taking, simulation, reconstruction, and analysis. The success is illustrated by the fact that first physics results could be presented to the public only two week after collision data taking had started in Spring 2018.

While the core Belle II software is mature it will have to be adjusted to the evolution of technology and requirements. It is therefore crucial that expertise is preserved like for any other component of the experiment.

## Acknowledgements

We thank the KEK and DESY computing groups for valuable support. We acknowledge support from BMBF and EXC153.

## References

1. R. Brun, F. Rademakers, Nucl. Instrum. Meth. **A389**(1), 81 (1997). DOI 10.1016/S0168-9002(97)00048-X
2. S. Agostinelli, et al., Nucl. Instrum. Meth. **A506**, 250 (2003). DOI 10.1016/S0168-9002(03)01368-8
3. D. Lange, Nucl. Instrum. Meth. **A462**, 152 (2001). DOI 10.1016/S0168-9002(01)00089-4
4. Spack. URL `https://spack.io/`
5. Cernvm file system. URL `https://cernvm.cern.ch/portal/filesystem`
6. Bitbucket. URL `https://bitbucket.org`
7. Jira. URL `https://www.atlassian.com/software/jira`
8. SCons – a software construction tool. URL `http://www.scons.org/`
9. Astyle. URL `http://astyle.sourceforge.net/`
10. pep8 - python style guide checker. URL `http://pep8.readthedocs.io`
11. autopep8. URL `https://github.com/hhatto/autopep8`
12. Google C++ testing framework. URL `https://code.google.com/p/googletest/`
13. Bamboo. URL `https://www.atlassian.com/software/bamboo`
14. Buildbot. URL `https://buildbot.net/`
15. C. Höppner, S. Neubert, B. Ketzer, S. Paul, Nucl. Instrum. Meth. **A620**, 518 (2010). DOI 10.1016/j.nima.2010.03.136
16. Rauch, Johannes and Schlüter, Tobias, J. Phys. Conf. Ser. **608**(1), 012042 (2015). DOI 10.1088/1742-6596/608/1/012042
17. Sphinx python documentation generator. URL `http://www.sphinx-doc.org`
18. Boost.python. URL `https://www.boost.org/doc/libs/1_64_0/libs/python/doc/html/index.html`
19. Pyroot. URL `https://root.cern.ch/pyroot`
20. F. Perez, B.E. Granger, Comput. Sci. Eng. **9**(3), 21 (2007). DOI 10.1109/MCSE.2007.53
21. J.D. Hunter, Comput. Sci. Eng. **9**(3), 90 (2007). DOI 10.1109/MCSE.2007.55
22. Python data analysis library. URL `https://pandas.pydata.org/`
23. Hep ipython tools. URL `http://hep-ipython-tools.github.io/`
24. S. Vigna, ArXiv e-prints (2014)
25. G. Marsaglia, Journal of Statistical Software **8**(14), 1 (2003). URL `http://www.jstatsoft.org/v08/i14`
26. Q.H. Dang, Federal Inf. Process. Stds. (2015). DOI 10.6028/NIST.FIPS.202
27. L. Wood, T. Elsethagen, M. Schram, E. Stephan, Journal of Physics: Conference Series **898**(4), 042060 (2017). URL `http://stacks.iop.org/1742-6596/898/i=4/a=042060`
28. Requests: Http for humans. URL `http://docs.python-requests.org/`
29. Xml inclusions. URL `https://www.w3.org/TR/xinclude/`
30. The xml c parser and toolkit of gnome. URL `http://xmlsoft.org/`
31. Xml path language. URL `https://www.w3.org/TR/xpath/`