WatchMan project—A Python CASE framework for High Energy Physics data analysis in the LHC era

Riccardo Maria Bianchi\textsuperscript{a,}\textsuperscript{*,}\textsuperscript{1,} 2, Renaud Brunelière\textsuperscript{b}
\textsuperscript{a} CERN, European Organization for Nuclear Research, CH-1211 Geneva, Switzerland
\textsuperscript{b} Physikalisches Institut, Albert Ludwigs Universität, Freiburg, Germany

\textbf{A R T I C L E    I N F O}

\textbf{Article history:}
Received 26 January 2011
Received in revised form 26 February 2012
Accepted 7 April 2012
Available online 17 April 2012

\textbf{Keywords:}
Large Hadron Collider
LHC
High Energy Physics
Data analysis
Computer aided software engineering
Python
Physics framework

\textbf{A B S T R A C T}

The world’s largest particle collider LHC is taking data at CERN, in Geneva, providing a huge amount of data to be looked at, of the order of several Petabytes per year. Nowadays, Data Analysis in High Energy Physics (HEP) means handling billions of experimental data in custom software frameworks. Physicists have to access and select data interacting with the experiment using dedicated tools; they also have to apply filter functions and analysis algorithms to test hypotheses about the physics underlain. Modern HEP experiments rely on complex software frameworks, hence writing the analysis code is not always an easy task, and the learning curve is usually quite steep. Moreover each hypothesis requires a dedicated analysis, in order to have a better control on it and to be able to validate the results among different groups of researchers. And the writing of so many analyses can be error prone and time consuming.

In order to ease the writing of such data analysis code, we built a software-generator: the idea is that the user inserts the settings of the physics analyses, and the final analysis code is automatically and dynamically generated, ready to run on data. Python helped us to build such a package. Its high-level and dynamic nature, together with its flexibility and prototyping speed are the key features which made our choice.

So we conceived and developed WatchMan, a Python CASE (Computer-Aided Software Engineering) framework to automatically generate reliable, easy to maintain and easy to validate HEP data analysis code.

© 2012 Elsevier B.V. All rights reserved.

1. The contest: analyzing billions of HEP data in complex software frameworks

The Large Hadron Collider (LHC), the world’s largest particle accelerator, is taking data at CERN, in Geneva, exploring the inner mechanism of Nature. LHC has already reached a centre-of-mass energy never reached before, opening a new range of energy where looking for new physics phenomena. Four main experiments have been built on the colliding points of the collider, to reveal and characterize the products of the collisions. The collected information are then filtered and stored to be analyzed. The analysis of those data will lead to new discoveries and to new answers about the mechanisms of Nature.

The amount of data from LHC is of the order of the Petabytes per year. To analyze all those data, physicists have to write pieces of custom analysis software to be able to access data and filter them with selection algorithms and filter functions, in order to keep only the interesting events which can lead to a discovery. But as one cannot be sure how new physics phenomena can show themselves, many models and theories have to be tested, and so the number of analyses which have to be performed on data gets very high.

Moreover modern HEP experiments rely on very complex software frameworks, which physicists have to interact with in order to access data and software tools to handle them, and to gather information about the detector status. So, beside the physics-related analysis code, physicists have to write a large amount of framework-related code, which has nothing to do with the physics one wants to explore; and with certain frameworks or data formats the amount of such extra-code can be much greater than the code directly related to the physics analysis.

Nowadays, within the HEP community, several frameworks on the market try to address the problem of the complexity of the software frameworks of the experiments, helping the user to write the code to analyze HEP data. Those frameworks mainly provide high level classes and functions in order to ease the writing of the code.
itself; that means that a certain knowledge of how the experiment framework works is still requested, and that the user – even using “helper classes” and high level functions – has to write the code itself anyhow. Moreover, many of those frameworks are bound to a specific data format or customized for a specific experiment, and only few are publicly available, i.e. not restricted to collaborators of a certain experiment. We can give here as example SFram [1] or Bender [2].

2. Python and CASE to automatically generate HEP data analysis software

But our approach is different from that one adopted by other frameworks. What we tried to do is providing to the user a tool to automatically generate the analysis code, instead of helping him to write it. Our idea is an ideal separation between the analysis strategy, related to the physics one wants to explore, and the implementation of such an analysis. In our view the process of analyzing physics data, from the point of view of a physicist, should rely mainly on the design and on hypotheses test: with the ideal possibility of testing many hypotheses in an easy and straightforward way. But nowadays the process is burdened by the need of writing a complex analysis code, with all the technicalities that, in addition, one has to learn in order to achieve that.

The main goal of WATCHMAN, hence, is to try as much as possible to relieve the user of that, letting her or him concentrate over the physics strategy of their analyses. Using the Python programing language and CASE [3] principles we built an analysis software generator: WATCHMAN. As we said above, the main goal of this new tool is to ease the implementation of data analysis strategies. The code and the documentation can be found on the WATCHMAN web site [4].

The package accepts input settings from users, and it automatically and dynamically generates the final analysis code, ready to be run on data. Fig. 1 shows the main idea behind WATCHMAN: the easy and automatic generation of the analysis code from the user inputs.

WATCHMAN is an analysis toolbox, it takes care of the common code necessary to run the analysis, besides helping the design and the implementation of the analysis, providing the most common algorithms used in HEP as building-blocks. WATCHMAN is also a framework: interfaces for different experiments and data formats can be added, and the user can expand it adding custom formulas to be used in the data analysis. Two interfaces have been provided with the framework, so far: one for the publicly available Delphes [5] data format, and one for the ATLAS experiment at LHC, restricted to the collaboration. More details are provided in Section 4.

Another valuable feature of WATCHMAN is the possibility to handle many analyses at the same time: the user can define as many analyses as wanted, and WATCHMAN generates the code for all of them together. Once the generated code is then run on the data, the output contains the results for all the analyses, while remaining quite small and lightweight. This is made possible by a flagging mechanism: all objects inside the output file are flagged according to the analysis they belong. In this way, while sharing objects and disk space, the different analyses are still well separated. More on this in Section 6.

All the WATCHMAN modules are written in Python, while the custom modular interfaces can be written in Python or C++ according to the needs of the user. Python was chosen after having considered other languages more used in HEP, like C++. Python’s dynamic and high-level nature, and its high prototyping speed, made it the ideal programing language to start to implement this new idea.

3. The WATCHMAN Python framework

WATCHMAN main goal, as already stated, is to ease the implementation of data analysis strategy, relieving the user of the writing of the code; and this is achieved generating the complete analysis code from the user settings in an automated and dynamic way. Hence, from the point of view of the final user, the basic usage of the framework is performed merely defining an analysis strategy in a text-like mode through a steering file, using the tools provided by the framework, and running the generated analysis code directly on data.

But WATCHMAN is also a framework, and it has been developed in a modular way. The user can add interfaces to a particular experiment framework or data format, and custom user-defined formulas.

Let’s take a first look at the framework architecture. In Fig. 2 the main layout of WATCHMAN is shown (only the main components and blocks are shown here). The user interacts with the framework through a "Steering File", a text-like file where the analyses can
be defined in a simple way using Python dictionaries. The steering file is the basic user-interface provided by the framework, designed to easily define analysis strategies and algorithms, without the need of knowing the underained programming language and framework. But a more flexible Python interface will be eventually built, to let the user interface the analysis code with external packages. This will be discussed at the end of Section 5. The “CutsLib” is a collection of common formulas and algorithms used in HEP analyses; the usage of those instead of implementing the same formula for every analysis class, makes the analysis code virtually error-free and easier to validate: once the built-in formulas are validated, all the analysis code generated by WatchMan is automatically validated as well. In the “Modular Interface” the details of the specific framework or data format are provided. As written above in Section 1, two interfaces are provided with the package so far, and others can be added by users; more details about the interfaces will be given in Section 4. The “Parser” module is the core of WatchMan. It’s the code generator: it parses the user settings, combining them with the specific experiment interface and with the common algorithms, and it generates the final Python analysis code: the “Generated-AnalysisLib” in the figure. Beside that, it also generates scripts to run the analysis on data – locally or on the GRID network – which contain instructions related to the specific data format in use. In order to handle data and to store the output, in WatchMan we make use of ROOT [6] and PyROOT [7] (more about them in Section 6 and in Appendix A).

WatchMan was born as a tool to handle code generation within the ATLAS experiment – which is running on the Large Hadron Collider (LHC) at CERN – and it was originally presented as “ATLASWatchMan”. Then the software was re-shaped, expanded and partially re-written, adding the modular interface mechanism described in the next section. The ATLAS-related code went, then, in a separate interface plug-in.

4. The modular interfaces

The core packages of WatchMan do not contain any code related to a specific experiment or data-format. All the specific code for a certain framework, is provided to the parser via a modular interface mechanism (see Fig. 3). For each specific interface a set of files, containing for instance setup code or container names, has to be provided. The parser then will blend those information with user settings, to build the analysis code. Two interfaces are provided with the framework for the moment: an interface to the publicly available Delphes [5] data files, and another for two different data formats used in the ATLAS experiment. Other interfaces can be added by the user, in a modular way.

Each interface provides the specific instructions related to a particular data format or experimental framework. For instance the names of the containers storing the physics objects in the data file, or the calls to external packages to set up the environment; or also the implementation of the function that returns the physical properties of the objects, for the specific data format.

The interface can be written in Python or C++, according to the requirements of the data format or the experiment framework one has to interact with. When written in C++, Python bindings are built in an automated way, through dictionaries automatically built with the tools provided by ROOT [6].
Fig. 3. The WatchMan modular interface main components. Data-format or experiment-related settings are specified in the files of the modular interface. The parser blends that information with the user settings to build the analysis code.

```python
channels = {
    'ljlep': {
        'channel': 'ljjle',
        'objSelection': {'electron': {'ptMin': 20. * Units.GeV,
                                      'deltaR_electron': 0.25},
                         },
        'cuts': { 1: {'label': 'leptonPtCutsExclusive',
                      'value': [20 * Units.GeV]},
                  2: {'label': 'jetPtCuts',
                      'value': [100 * Units.GeV, 40 * Units.GeV, 40 * Units.GeV]},
                  3: {'label': 'jetPtVeto',
                      'value': [40 * Units.GeV]},
                  4: {'label': 'missingEtCut',
                      'value': [80 * Units.GeV]},
                  5: {'label': 'meff',
                      'value': [100 * Units.GeV,
                                'formula': 'meff3jetsNetleps',
                                'custom': True],
                  },
             },
    },
}

# ---
collections = {'electron': {'select': True}}
# ---
userBranchesToFill = {'meff4j0lep': {
    'label': 'meff4j0lep',
    'type': 'float',
    'formula': 'meff'}
}
# ---
dumpContainers = {'jet': {}, 'electron': {}},
```

Fig. 4. Example of physics analysis implementation using the steering file.
5. The user interface

The steering file is the only final user interface, for the moment; an additional Python interface will be discussed at the end of this section. The steering file is a text-like file containing the user settings written in the form of Python dictionaries and variables. The look & feel of such a Python file is like a simple text file with labels and fields to be filled, easy to be understood even by users not experts in programming languages. An example of steering file usage is shown in Fig. 4, where a simple example analysis is defined as explained in the following.

Each steering file contains four main sections: a first part where global options are defined (like folders, file names, etc.), a second part where the analyses and the containers of the output file are chosen, a third part where the user can set the data files which the generated code will be run upon, and a last fourth part where the user can insert custom formulas to be used in the analyses. Skipping the first and third parts, which are mainly lists of file system paths and booleans – whose meaning is straightforward for the user – we will describe the second and fourth parts in the following.

The second part of the steering file is where the user defines the analyses. Fig. 4 shows an example of the usage of that second part of the steering file. The container of such definitions is the “channels” Python dictionary (at line 1 in the listing), whose main entries are the different analyses defined by the user, identified by a string label as key (the “3J01lep” in the example, at line 3). For each of them, three main sections can be configured, as string entry or sub-dictionaries: respectively, “channel”, “objSelection” and “cuts”. The first one, “channel” (at line 5), is a string saying which objects (particles) are going to be analyzed in that analysis, and it’s used then for labels, plot names and some internal checks. The second key of the dictionary is the “objSelection” key (at line 7 in the listing). Here the user can set the criteria used to select the objects inside data; in this simple example the user sets threshold values on physics quantities (more in detail, we request here a certain energy for electrons, and a certain angle for their trajectories); many other options and requirements can be added. Objects passing such first-stage selection will be flagged as described in Section 6 and shown in Fig. 8.

Analysis definition continues then in the “cuts” sub-dictionary (starting at line 11 in Fig. 4), which sets the second stage of the selection.⁵ Here the user can enter as many selection criteria as wanted: each entry has an integer number as key, defining the order of application of each criterion. Here integers are used as hash-able keys, since the filters on the physical quantities have to be applied in a specific order. Practically the user sets a sorted sequence of actions, whose integer key sets the position in the sequence. The same order will then be used when generating the code, so that the actions will be taken on data in the right sequence. As you can see in the listing (for example at lines 14–16), each criterion has a key “label”, whose value is a string (e.g. “jetPtCuts”); that string is the name of the formula, or the algorithm, which will be applied. That name is associated to a formula stored in the built-in library (the “cutsLib” module shown in Fig. 2), and it’s used simply by calling its name in the steering file. Arguments to the formula are passed by the user in the “value” entry (for example at lines 15–16 we pass the energy thresholds used in the formula “jetPtCuts”).

After having run WATCHMAN on this steering file, we get the complete Python analysis code automatically generated, and ready to be run on data. In Fig. 5 a very simple example of the generated code is shown; more complex selection criteria trigger the generation of more complex and longer formulas. In Fig. 5, on the right side, we show only the snippet directly generated from the settings shown on the left side; but more code is actually generated, for example the code needed to interact with the experiment framework and to access data.

The example steering file here, in Fig. 4, is the direct implementation of the “napkin scribble” shown in Fig. 1: that is the goal of WATCHMAN: from an idea about a great physics analysis, to the complete analysis code, in few minutes!

It’s worth to notice that a whole complete, yet very simple, analysis, like that one taken as example, would have required a certain – large – amount of lines of code to be written, without using WATCHMAN; and it would have also required also a certain – usually long – developing and debugging time.

While using WATCHMAN the user can define the strategy of the analysis, in the steering file, almost like he or she would do on a napkin, perhaps sipping a coffee, as already shown in Fig. 1. The user – who is a physicist – ideally should not need to know how to program the analysis; he should just have to plug in the strategy in a simple way and obtain the results, without taking care of the

⁵ You can find a real complete example of a steering file at http://svnweb.cern.ch/trac/WatchMan/browser/trunk/run/WatchMan_AnalysisSteeringFileBenchmarkChannels_Delphes.py.

⁶ In HEP those two selection phases are something very different and well separated. The first stage selection is usually called “object selection” and its main goal is to “clean” the data, involving selection criteria related to the detector performances; while the second stage is called “event selection” and it contains selection requests based on the physics processes of the underlain theoretical models.
software details underneath. And WatchMan is a first attempt to get that.

In the steering file the user can also decide what to read from the data files and what to store in the output file: which object collections or variables. In the same Fig. 4 at line 33 the user decides to read only the "electron" particle collection from the input data sets. At lines 35–36 the user decides also to store the outcome of the custom formula "meff" (discussed here below) as a variable called "meff4j", and at line 38 it sets that also the collections of electrons and jet-particles have to be stored in the final output file where the results of the analyses defined in the steering file are saved.

As we said above the main and most common formulas and algorithms are built-in in WatchMan, and they simply can be used in the analysis definition. But the user can also expand the built-in library adding custom formulas. Those can be specified in the last section of the steering file, in the dictionary "userFormula": each formula fits in a dictionary entry, whose key is a string with the formula name, as shown in Fig. 6. Let’s back step for a moment. In Fig. 4, the last field in the "cuts" section is an entry called "meff" (at lines 24–27 of the listing), which is flagged as "custom". The user flags it as "custom" if it’s not part of the built-in library but a custom formula he or she wants to add. To do so the user have to add the formula definition in the last part of the steering file. Custom formulas have to be written in Python, as shown in the listing 6. Once the formula is defined there, and provided that the "custom" flag is set to "True", the custom formula can be used like those contained in the built-in library: the "meff" formula in the example is used in the analysis definition, and values are passed to it in the same way as to built-in formulas, through the key "value". A same custom formula can be used in the definition of one or more analyses, or to fill a container in the output file.

Here Python helps a lot again. Its dynamic and interpreted nature let us to insert a formula here without having to strongly typed variables and without having to compile and linking it, like the user had to do when using other languages, like C++. So the user can enter a custom formula in the steering file and run the generation of the code directly, without passing through a compilation phase which would make the whole process more complex and slower.

Concerning security, so far there are no routines to checks and sanitize the user input formula. In fact, even if the tool is not intended to be exposed to other people different than the user, and even if it’s run only on private workstations and not exposed on the web, a check on the formula inserted by the user should be performed anyway, in order to verify that the code is not accidentally harmful for data or for the underlain system. Input formula checks will be implemented in future releases.

The steering file is the only user interface provided up until now. It was designed to let the user describe analysis strategies and algorithms in the simplest way but, of course, it lacks many of the features that a full Python interface could add to the framework. Through the importing of a "watchman" Python module into a script, the user could access all WatchMan functionalities while still being able to connect the user’s code with external packages and libraries. A full Python interface, for example, could let the user steer its own analysis through a custom GUI, or use data visualization and analysis tools like PyROOT [7] or Matplotlib [8]. Other HEP packages offer such a Python module as main interface. Among the data analysis toolboxes providing a full Python interface, the closest one to WatchMan is – as far as we know – BENDER [2]. BENDER exposes a Python module: the user can access all its functionalities importing the "bendermodule" module in the Python code. A similar full Python interface will be implemented in one of the next WatchMan releases.

6. The WatchMan output file and the objects flagging mechanism

Once the data analysis code generated by WatchMan is run over the data, the output of all the analyses is collected in a unique file, to save storage space and to let the user easily combine together or compare the different analyses. But, as we already said in the introduction, we also have to ensure that each analysis can be kept well separated from the others, especially for validation purpose. And here Python helped us again: its dynamic-typed nature let us to build a flagging mechanism which flags all the objects in data, at run-time, without having to know the type of the object a priori.

---

[1] BENDER is a Python-based analysis environment designed to help the physicists to write analysis code to analyze data from the LHCb experiment. WatchMan and BENDER are based on the same design principles and provide similar functionalities, still having been conceived and developed separately and independently.

Fig. 7. Definition of custom user-defined object-selection requirements in the steering file. In this way it’s possible to globally define the requirements, and then using them in different analyses, as in the example “tau_channel” above (lines 17–19). The object-selection sets get stored then in a map in the output file, to be used by the flagging mechanism, as explained in the text.

Each object in data (e.g. particles, events, physics quantities, etc.) can “belong” to one or more analyses, according to the requirements and selection criteria which have been applied in the algorithms. All objects are stored only once in the output file, but they are flagged according to the analysis or the analyses they belong. In Fig. 8 an example of this flagging mechanism for the object selection is shown. In this example we take a collection of “jet-particles”, and we consider the energy associated with each of them. Each particle is stored only once in the output file, but when they pass the different energy selection requirements of the different analyses, they are flagged accordingly, and the flags are saved with the object itself. More in details, in Fig. 8 the middle row labeled as “jet4mom.pt()” is the container storing the value of the energy for each jet-particle in one event: “154” is the value of the energy of the first jet-particle, “128” is the value of the second one, and so on. The upper row labeled as “InfoTree::objSelectionMap”, instead, is the map storing all the selection criteria the user set in the steering file, grouped under the name given to a specific set of requirements. Here in Fig. 8 we see the map storing the object selections defined as example in Fig. 7. In that example the user globally defined in the steering file two sets of object-selection criteria, under two names:

```c++
objectSelectionAndOverlap = {
    'TauSelect': {'jet': {'ptMin': 40.*Units.GeV, 'etaMax': 2.5},
                  'tau': {'applyOverlapRemoval': True},
    'MyChan': {'jet': {'ptMin': 60.*Units.GeV, 'etaMax': 2.5},
               'objSelection': 'TauSelect'},
}

# ---
channels = {
    'tau_channel': {
        'channel': 'ltX',
        'objSelection': 'TauSelect'
    }
}
```

![Fig. 8. How the ObjSelection branch is filled for particle-like objects.](image)
from ROOT import gRandom, TCanvas
from ROOT import TH1F

hpx = TH1F('hpx', 'px', 100, -4, 4)

for i in xrange(3000):
    px = gRandom.Gaus()
    hpx.Fill(px)

hpx.Draw()
c1.Update()

definitions

7. Conclusions

WATCHMAN implements a new idea in the HEP field, the usage of Python to develop a Computer-Aided-Software-Engineering framework to automatically build reliable, easy to maintain and easy to validate data particle physics analysis code.

WATCHMAN is an analysis code generation kit, written in Python, which handle many analyses at the same time, generating the final complete analysis code, ready to be run on data. The framework also takes care of the specific data format setup and of the underlain technicalities, relieving the final user of the need of learning the details of them. And it can be expanded with modular interfaces to work with new formats or experiments. In a future release a new Python interface will be added, which will let the user to integrate WATCHMAN functionalities in any Python analysis code.

WATCHMAN is an open-source Python project under continuous development, with a first stable release; it has a small community of active users and in the past months it has already been used with success to analyze data for some scientific notes and contributions at LHC (among the public ones, see for example [9,10]).

Acknowledgments

First of all many thanks to the organizing committee of the “EuroSciPy 2010” meeting for the very interesting and perfectly organized conference, and for having given me the opportunity to show our work. We would also like to thank our colleagues in the Freiburg group who have contributed to the success of the package. In particular Florian Ahles, Asen Christov, Debra Lumb, Jan Erik Sundermann for their contributions to the ATLAS ESD/AOD data format interface; and Janet Dietrich, Michael Rammensee, Zuzana Rurikova and Kathrin Störg for their extensive tests and for their suggestions. And to Pere Mato from CERN for having let us know other interesting HEP tools which we did not know. And many
Appendix A. PyROOT

In WatchMan we made use of PyROOT [7], the Python bindings to the ROOT Analysis Framework [6], in order to handle data and to store the output. PyROOT lets the user access from Python to all classes of ROOT, which is a complete C++ framework providing tools for data analysis, statistics, 1D, 2D and 3D plotting, fitting, event display, Multivariate Analysis, detector geometry and more. ROOT is an open source project started at CERN, and expanded over last years with the contributions from many HEP institutes and universities around the world, and it’s the de facto standard analysis tool in HEP data analysis. PyROOT also allows the “glue-ing” of ROOT libraries with any non-ROOT library or applications that provide python bindings. An example of basic usage of PyROOT is shown in Fig. A.9.

Appendix B. Considerations about speed performances

When we recently tested our framework on the newest real data sets, we noticed that the speed of the generated Python code is not enough for the day-by-day usage with the large amount of data from the LHC experiments. Hence, still keeping the usage of Python for the whole framework, we thought to use embedded C++ code for the more computing-demanding loops which appear in the final automatically generated analysis code. As a first test we implemented loops using Weave [11] and Instant [12]. This let us to speed up the code, yet keeping the simplicity and the cleanness of Python. Other parts have been re-written in C++ and python bindings have been built using the LCG Reflex [13] reflection tools.

Following suggestions from the scientific Python community, the combination of Numpy arrays with Cython [14] code will also be tested soon, aiming at a larger speed-up factor on the generated code.

References