PROCEEDINGS OF SCIENCE

WatchMan Project – Computer Aided Software Engineering applied to HEP Analysis Code Building for LHC

Riccardo Maria Bianchi[*] [a,b] [†], **Renaud Brunelière**[b], **Sascha Caron**[b]
[a] *CERN, European Organization for Nuclear Research*
CH-1211 Geneva (Switzerland)
[b] *Physikalisches Institut, Albert Ludwigs Universität*
Freiburg (Germany)
E-mail: rbianchi@cern.ch

A lot of code written for high-level data analysis has many similar parts like environment setup, reading out the data of given input files, data selection, object selections, calculation of basic physical quantities and the output of the analysis results. Moreover nowadays the complexity of software frameworks of HEP experiments forces the user to acquire many technical details before starting writing the code implementing the analysis strategy. Writing such code for each new analysis is error prone and time consuming. Some software frameworks already tried to simplify the task, offering higher-level classes and functions, but the usage of them nevertheless requires from the user a certain knowledge about the framework beneath.

What we wanted to achieve, as much as possible, with this package is the separation of concerns, where the two layers are the physics analysis itself, with its rationale and its algorithms, and the coding part involving the framework of the experiment or the specific data format. In our view the user should mainly take care of the physics part, directly and easily translating an idea into analysis code, leaving the technical details of handling data to the “machinery” beneath.

As a solution to this problem we designed WATCHMAN, a “data analysis construction kit" and a highly automated analysis code generator. WATCHMAN takes as input user-settings from a text-like “steering file”, and it dynamically generates the complete analysis code, ready to be run over data, locally or on the GRID. The package has been implemented in Python and C++, using CASE (Computer Aided Software Engineering) principles. The package can be interfaced to different data formats or experiments via a modular interface mechanism.

WATCHMAN implements a new idea in the HEP field, the usage of CASE to build reliable, easy to maintain and easy to validate data analysis code, mainly aimed at analyzing new data from the LHC collider.

13th International Workshop on Advanced Computing and Analysis Techniques in Physics Research
February 22-27, 2010
Jaipur, India

[*] Speaker.

[†] Many thanks to the organizing committee of the ACAT Workshop and to everyone in my group whose suggestions and criticisms contributed to this work. The work of the author was done while affiliated with *Albert Ludwigs Universität, Freiburg*, and it was supported by the German *BMBF*.

1. Introduction: why using CASE in HEP software?

In these months the Large Hadron Collider (LHC) has started to run again at CERN. The collider has been built to reach a centre-of-mass energy never reached before, and opens a new range of energy to explore for new physics phenomena.

Four main experiments have been built on the collider, on the spots where the particle beams meet each other. Those experiments are revealing, reconstructing and registering data from the collisions. At stable working conditions there will be one collision every 25 nanoseconds, and for each collision many particles are produced at the same time, particles whose nature and behavior reveal the insights of the physics interaction who generated them. Due to the high rate of the collisions, and to the complexity of the detectors, experiments in the LHC epoch will produce petabytes of data per year. To analyze them in a search for new physics, physicists have to write pieces of software which contain algorithms aimed to filter events and objects, in order to keep only the interesting ones which can lead to a great physics discovery. And as we cannot be sure how new physics can display itself, a large number of different data analyses has to be set up to scan all possibilities.

In High Energy Physics (HEP) the typical approach used while building analysis software is creating a new class every time a new analysis is implemented. Then one starts to write code to apply cuts or algorithms to the objects contained in data files. But one has also to add a lot of code related to common operations or related to the specific data format or experiment. The software infrastructure of modern HEP experiments rely on complex frameworks: sets of packages linking sub-detectors output and reconstruction chains, providing interconnections among the various parts of the experiments and providing to the end user the access to data. In the user code the framework layer very often consists of lines of code aimed to configure and access the framework packages. Lines of code not linked to the physics analysis at all, but necessary to initialize the framework, importing necessary modules, loading data, accessing data, accessing containers in data files, looping over collections of particles, booking and filling histograms, handling files. All those operations are not related to the physics we want to explore, but they are necessary in order to run the code to analyze data. A typical analysis code will then contain a great part of setup and common operations code, followed by a small to very large part of actual physics-related code. Then, if we start another analysis, we usually start from a copy of the first one and we modify it; or we just cut-and-paste big portions of code into the new class. And this is the usual approach for every new class we have to implement. After a few iterations we end up with a plethora of classes and files, with huge portions of code in common (Fig.1).

Within certain software frameworks[1] or with certain data file types the amount of extra-code can be much greater than the code directly related to the physics analysis. And for that reason those files become very difficult to be maintained, updated and validated. For example, let us imagine what would happen if several container names were changed in the data file format; or if a function to access data changed in the experimental framework; things that can happen, especially in experiments in early stages. In that case we would need to open and edit all files we created with the cut-and-paste approach, changing all the concerned commands and values. And the same

[1] like the “Athena" framework in the ATLAS experiment, for which this package was originally developed, with the name ATLASWATCHMAN [7].

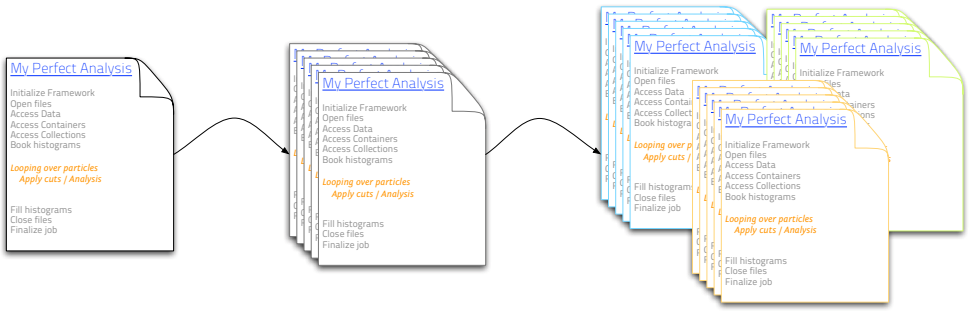

Figure 1: Common approach when writing analysis-aimed source code. One usually starts writing a class, and then this is copied and pasted into new classes when starting implementing new analyses. After few iterations the developer ends up with a plethora of classes to be maintained, with a large part of the code in common.

thing would happen if we decide to change a value of a certain physics cut, or a threshold value of some object selection cuts. Hence this approach is error prone, and it makes the analysis code very difficult to maintain.

That's why we decided to start the development of this package. We realized that most of the physics analysis code produced in our working group contained a huge burden of framework-related common code, not directly related to the analysis itself and difficult to handle. So we thought to develop a CASE (*Computer Aided Software Engineering*) [6] package to ease the development and the maintenance of the physics analysis code. The user has only to define the actual physics analysis strategy, in a user-friendly way; then the package parses the user entries, adds all the common and framework-related code, and it automatically and dynamically generates the complete code, ready to be run on data files, both locally or on GRID [4].

Another source of bugs, while writing repetitive code, is recurring formulas. In physics analysis code, programmers very often use those formulas many times: let's think about cut-based analyses, where one selects particles and events through boolean expressions on thresholds values of specific quantities, like the particle energy or the direction of its trajectory. The concerned formulas are usually quite simple and small, but they are used many times in order to implement the analysis strategy. Let's now think about a change in one of those formulas. If that happens the programmer is obliged to walk through all the classes to edit and update all the formula occurrences. This is even more error prone if the analysis code is based on large and complex formulas. Thus we included in WATCHMAN a library of recurring common formulas, to be used by the user in the analysis. In this way, if a change has to be done in such formulas, it has to be done in one place only, inside the collection of formulas.

The presence of the formula library and the fact that the code is automatically generated, makes the analysis code very easy to validate: once the formulas in the collection are validated, all the automatically generated analysis code is automatically validated as well (except only for custom code added by the user).

Another reason why we decided to start the development of a CASE package is to ease the writing of the analysis code. In fact, once the interface to a certain framework is built, a user can

write the physics code without knowing the framework, which is usually quite complex in modern HEP experiments. In Fig.2 the main idea of WATCHMAN is presented: easily implementing a modern physics analysis on data, just directly translating an idea into analysis code, without wasting time with framework-related and common code; and at the same time avoiding introducing errors.

One of the most valuable feature of WATCHMAN is the possibility to handle many analyses at the same time: the user can define, in the text-like steering file, as many analyses as wanted, linking each of them separately to certain sets of formulas or custom user code. Then WATCHMAN generates the code for all of them together. Once the analysis is then run over the data, the output will contain the results for all the analyses, while remaining quite small and lightweight. This is made possible by a flagging mechanism, which flags objects inside the output file according to the analysis they belong.

Experiments frameworks (like "Athena" for the ATLAS experiment, to give an example) generally provide low- to medium-level functions to access, handle and analyze data. On the contrary WATCHMAN, working on a higher level, provides tools to access and to manipulate data which are independent of the underlying framework or of the data format which is used. WATCHMAN accepts interfaces to many frameworks or data formats and, once the interface is provided, the user only has to define the data analysis in a text-like form in a "steering file", without taking care of the details of the framework which is used beneath, and without having to write the actual code. The same analysis specification file (the "steering file" cited above) can then be used on different data formats, merely choosing another interface by setting the corresponding options in the "steering file". Different customized actual analysis code is then generated for each different framework or data format interface, starting from the same "steering file" filled by the user.

The problem of the complexity of the software frameworks of modern HEP experiments is something which other software projects have tried to find a solution for, trying to help and to guide the user in the writing of the analysis code. Among them there are frameworks which simplify the life of the physicist providing medium- to high-level functions and classes (among the publicly available ones there is, given as an example, "SFrame" [11]). But, even in this case, the analysis code has to be written by the user – usually in programming languages like C++ or Python – and a certain knowledge of the framework beneath, and/or about the specific data format which is used, is required.

With WATCHMAN we wanted to go beyond that, tring to provide to the user a tool to define the analysis as one would do on a sheet of paper, or on a napkin while sipping a coffee... (see Fig.2). And so, as far as we know, WATCHMAN is the first "HEP analysis code generator" on the market, which not only provides really high-level tools to simplify the writing of the HEP analysis code, but which also actually generates, from the user settings, the complete code, ready to be run on data.

In the following sections more details about the implementation are described, starting with the core software, continuing with the modular interfaces and the user front-end and ending with an example of analysis implementation in WATCHMAN.

2. Core Software

WATCHMAN is a framework with a core part containing common algorithms, which are pre-

From analysis ideas at the coffee table... ...to actual analysis code!

Figure 2: Main idea of WATCHMAN: letting the user easily implement analysis ideas.

sented in this section, and modular interfaces targeted to the specific data format which one wants to analyze; those will be presented in the next section. A parser which parses user inputs, and which handles all the framework components, completes the layout.

The layout of the main components is shown in Fig.3. The core is made up by a parser and a collection of common formulas; a steering file is the only front-end user interface. The parser ("`Parser.py`" in the figure) is the main engine of WATCHMAN: it's a python module, whose duty is to read the user settings from the steering file ("`SteeringFile.py`" in the figure), to interact with the data-format interface in order to acquire specific setup code or default parameters, and to combine the user settings and the user custom formulas with the common formulas in the framework library ("`CutsLib.py`" in the figure). At the end two files are generated by the parser: the actual complete analysis code implementing the user-defined physics analysis, and a script to run this analysis code on data files ("`GeneratedAnalysisLib.py`" and "`Generated Run Script`", respectively, in the figure).

The generated run script can be run both locally or on GRID. For the ATLAS ESD/AOD data format interface provided with the package, a third script is automatically generated by WATCHMAN, together with the run script and the analysis code: a script to launch the user analysis directly on data stored on the GRID in an automated way, via the PANDA GRID Framework [5], over a set of datasets defined by the user. Similar scripts to run the user analysis on GRID, are currently under development for the other interfaces provided by WATCHMAN.

The whole WATCHMAN framework is written in Python, and it makes use of the ROOT framework [10] and its Python bridge PyROOT [9] to read the data files and storing the output. C++ language is also used whenever needed by a particular interface, and Python bindings are automatically built using the binding tools provided with ROOT. The ROOT package is required at the moment to run the generated analysis code, as the output data are saved in a `.root` file. But interfaces to data formats other than ROOT can be implemented.

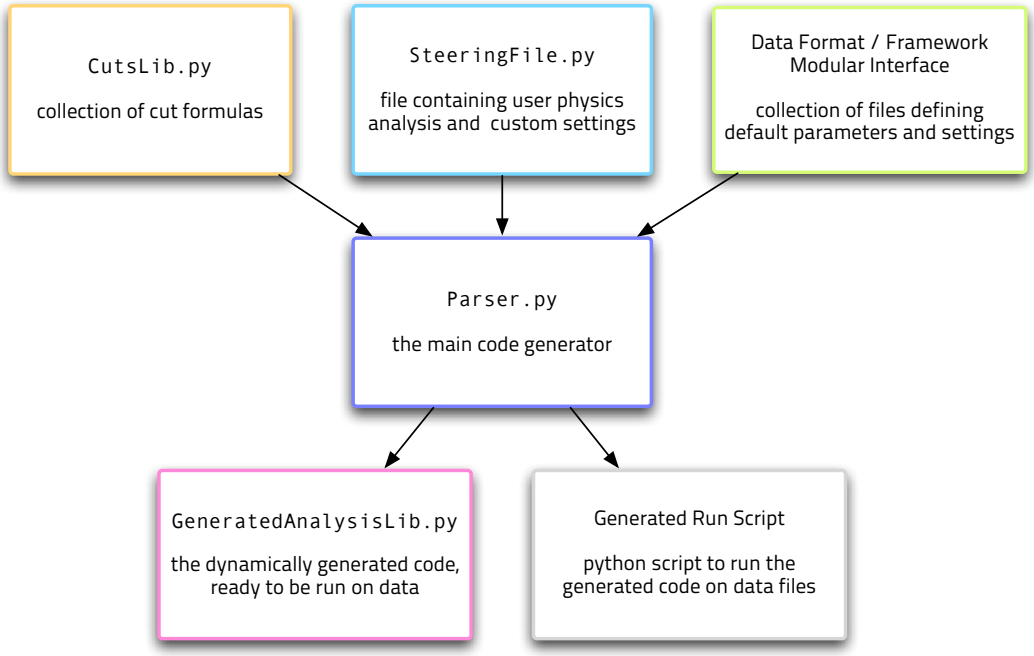

Figure 3: WATCHMAN main components layout. The "parser" is the main engine of the framework, whose duty is to combine user settings with common code, to generate a complete analysis code ready to be run.

3. Modular interfaces

The core packages of WATCHMAN do not contain any code related to a specific experiment or data-format. All the specific code necessary to build complete code for a certain framework, is provided to the parser via a modular interface mechanism (see Fig.4). 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.

Three interfaces are provided with the framework so far: an interface to the publicly available Delphes [3] data files, and two others for two different data formats used in the ATLAS experiment, which is running on the Large Hadron Collider (LHC) at CERN. Other interfaces can be added by the user, in a modular way.

The interface provides the specific instructions related to the 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. More details about the interface components and instructions on how to add a custom interface are presented and explained in the WATCHMAN wiki, currently under preparation [1].

4. User front-end interface: the Steering File

The so called "*steering file*" is the only user interface of WATCHMAN, so far. Within the

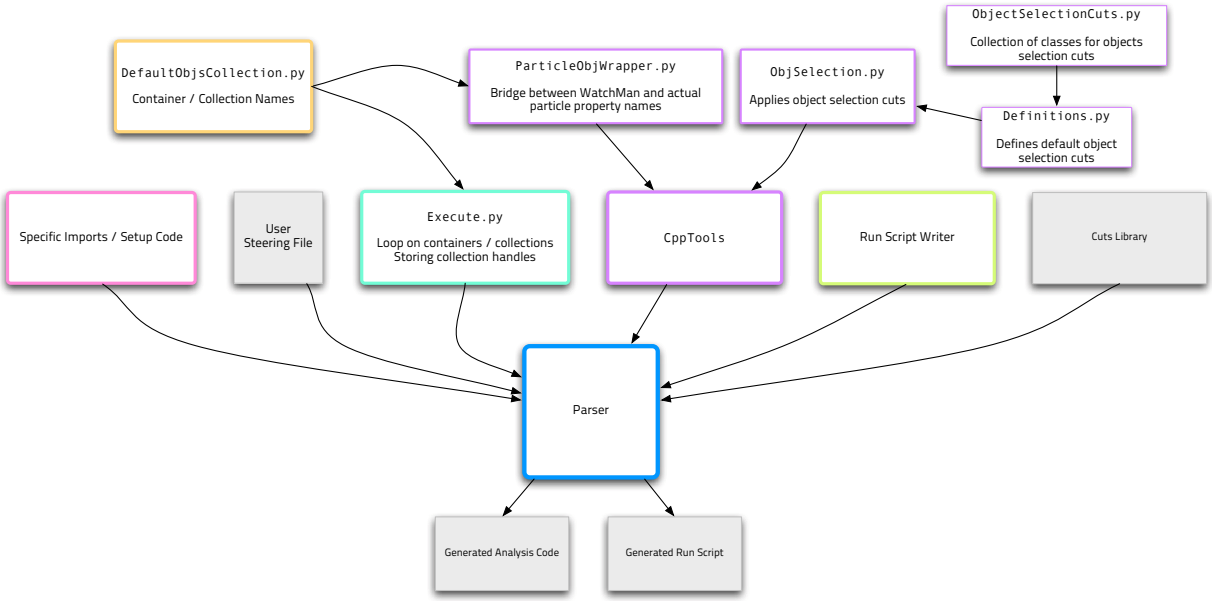

Figure 4: The WatchMan modular interface main components. Data-format or experiment-related settings are specified in the files belonging to the modular interface. The parser blends that information with the user settings to build the analysis code.

steering file the user can set global options for the generated code and define as many analyses as wanted. Through the global options the user can choose to store different metadata in the output file and to change the behaviour of the generated code. The analyses are defined in a text-like way through Python dictionaries: the user inserts the steps for each analysis as it would be done on a sheet of paper, providing the name of the formula used in each step, and the values taken as thresholds. Fig.2 shows an example of the translation of the analysis idea scribbled down on a sheet of paper at the coffee table (on the left side of the figure), into actual analysis code through the steering file (right side). In the figure a simple analysis called "`3j0lep_new`" is being implemented, setting object selection and event selection cuts; for each cut a threshold value is given; for many cuts defined in the formula library (as for the "`jetPtCuts`" in the figure) the length of the list containing the threshold values sets the number of particles which the formula takes into account. If a custom formula is used, that has to be added in the steering file. Then it will be parsed by the parser and integrated in the generated code. Custom formula can be used both for cuts or to fill containers in the output file containing particular user-defined values. In the case study presented in Section 5 more details will be shown on how to implement a physics analysis in WATCHMAN.

5. Case study example: how to implement a SUSY-related analysis

We present here a short example of implementation of a physics analysis with the WATCHMAN analysis code generator. We will use the Delphes [3] interface provided with the framework, and we will implement a cut-based analysis aimed to discover evidences of Supersymmetry (SUSY) [8] at LHC energy, with several physics channels. We skip here the set-up phase; instructions about it

can be found on the Wiki page of the project [1]. Instead let's look at how to implement a physics analysis in the steering file. Let's say we want to analyze a certain decay of a SUSY particle. We start by defining a channel selecting certain particles with certain properties: for example we want to select 3 hadronic jets plus one lepton plus missing transverse energy, and we want to apply certain cuts to the physical properties of those objects (like the transverse momentum P_T), or of the whole event (like the sphericity[2] or the effective mass M_{eff}[3]). In the steering file the user has just to define those cuts in a text-like way, as shown in Listing 1. As the reader can see Python dictionaries are used to contain the definitions for each channel and for each cut; but besides some extra parenthesis, the user settings appear mainly as plain text. The definition of a channel is composed by a label defining the channel ("`3j0lepMediumCuts`" in the listing), by a set of cuts used for selecting objects inside data before using them for the physics selection over the event ("`objSelection`") and a set of sorted cuts for the event selection ("`cuts`"), which are actually those more related to the physics which we want to discover. In the example channel all the cuts but one use built-in formulas provided by WATCHMAN; only the last cut, ("`5:'meff'`") uses a custom user-defined formula: the flag "`custom`" is set to "`True`" and the formula is provided by the user in the same steering file, as shown in Listing 2. In that listing one can see how the containers store the physics objects from data: all the containers whose names are defined in the modular interface (cfr. Section 3) are taken into account by the "Parser" and the code to make them available to the user code is generated. Thus in the end the user can define formulas where she/he can loop easily over all the physics objects containers, looping over "`candidates`". Other not-particle-like objects are accessible via another container called "`collections`" . Moreover the same formula can be used both for cuts, as in the example of Listing 1, or to fill containers in the output file.

6. The output file containing final results: the WATCHMAN Ntuple.

A characteristic of WATCHMAN is that the output file contains the results for all the analyses defined in the steering file. That means that common objects belonging to different analyses (let's think of common particles like muons, jets or electrons) are stored only once for all the analyses, and they are merely flagged according to the specific analysis they belong. Each physical quantity is then computed with the right subset of particles which satisfy the requirements defined by the user for a specific analysis.

WATCHMAN accepts also a list of steering files, and it builds a unique analysis code to be run on data. Thus people within a working group can develop their own analysis separately, which then can be automatically combined with the others and run together. In this case the output file will contain the results from all the analyses defined in all the steering files.

An important work was done to find an effective way to have independent analyses, while assuring a lightweight output file. To reach this goal an attentive flagging mechanism of particles and events has been set up, which make possible to share objects among analyses while avoiding the storage of redundant information. A more detailed explanation follows.

[2] *Sphericity* is a measure of how spherical an event is, i.e. of how spherical is the spatial distribution of objects belonging to the event (like jets, leptons, MET).

[3] The *effective mass* M_{eff} is defined as the sum of the transverse momentum p_T of the selected particles (like jets and leptons) and the missing transverse energy MET

```
channels = {

    '3j0lepMediumCuts':{
        'channel': 'ljjjv',

        'objSelection':  {'muon':{'ptMin': 20.*Units.GeV},
                                  'electron':{'ptMin': 20.*Units.GeV},
                                  'photon':{'applyOverlapRemoval': False},
                                  'tau':{'applyOverlapRemoval': False},
                                  },

        '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': 'meff3JetsMetLeps',
                        'custom': True },
                },
       },
}
```

Listing 1: Example of physics analysis implementation in the steering file.

Figure 5 shows how the "`ObjSelection`" branch in the output file is filled for particle-like objects. In this example we consider the jet collection. Let us assume that we have three object selection definitions: "`Default`", "`TauSelec`" and "`MyChan`". "`Default`" is used when the user does not define an object selection for an analysis; the other two are custom definitions, specified by the user in the steering file, and in this example they are used for two other different analyses. Hence three flags for the three object selection definitions are created; and those flags are stored in the output file, in the "`objSelectionMap`" vector contained in the "`InfoTree`" ROOT `TTree` object. The selection flags for particles (here for the jet collection) are filled according to the position in the "`objSelectionMap`": if the physical properties of a jet satisfy the selection cuts defined for a specific object selection, a "**1**" is stored inside the "`jetObjSelection`" at the corresponding position; otherwise a "**0**" is stored.

In a similar way the events are flagged according to the analyses they belong. The "`channel`" branch is a vector of strings, filled with the analysis name, each time an event has the characteristics to satisfy the event selection of a particular analysis. In this way the scan over the events belonging to a specific analysis is very easy: it's only matter of specifying a flag. The events are stored only once, even if they belong to several analyses.

In Figure 6 a distribution of the "`channels`" branch is shown, where all the analyses specified in the example steering file, shipped with WATCHMAN, are visible. The plot is related to the same SUSY analysis example presented in section 5, but here we consider all the analyses defined in the steering file: 10 SUSY-like analyses, with different settings, i.e. different object and event selection cuts. Each event can satisfy the requirements of one or more analyses, thus in the output file we

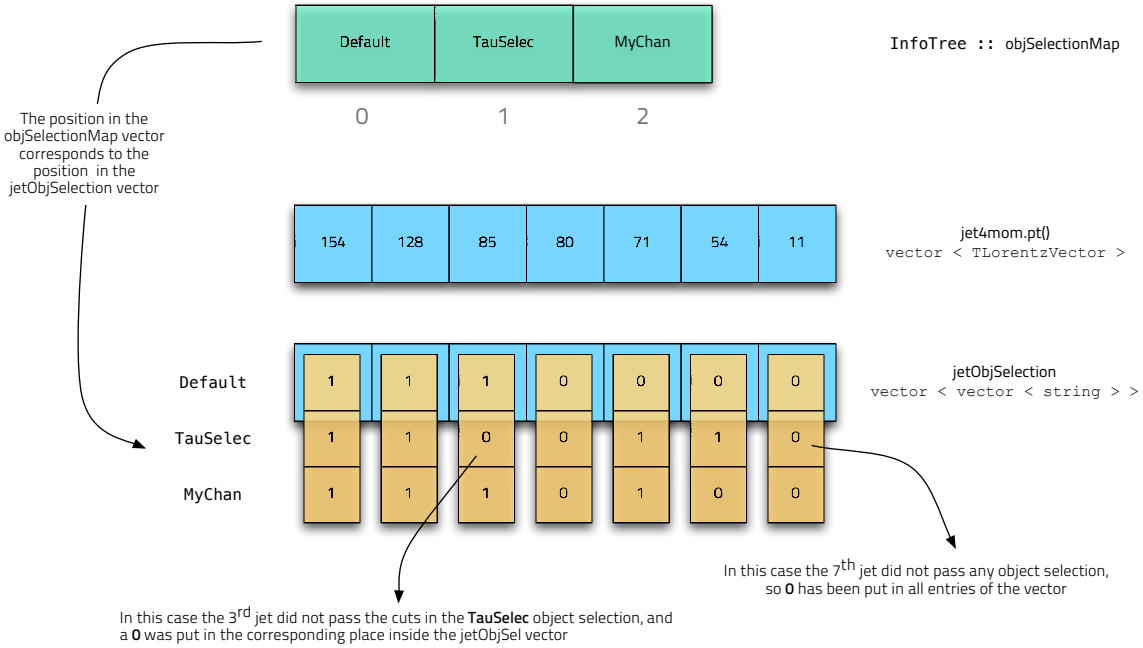

Figure 5: How the `ObjSelection` branch is filled for particle-like objects.

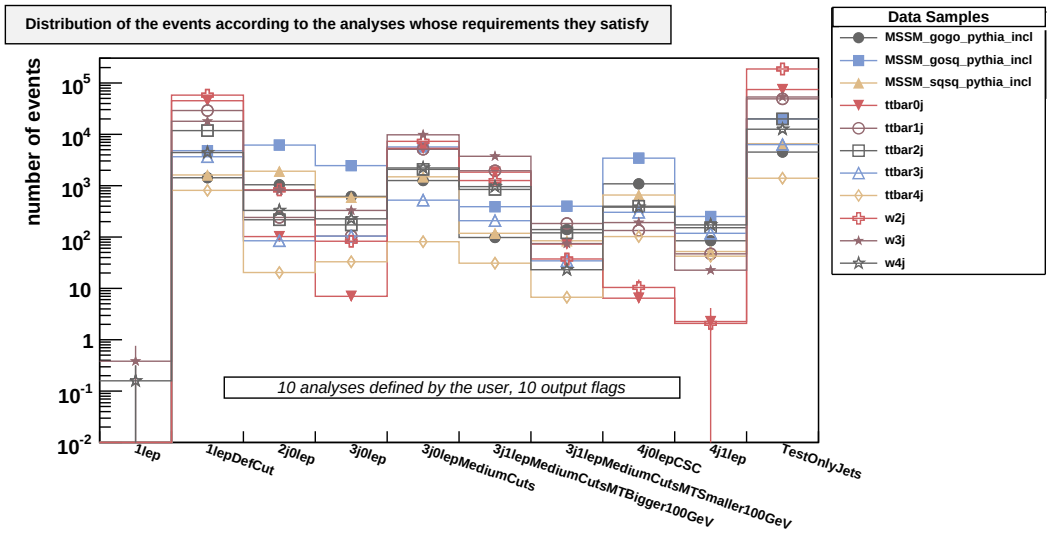

Figure 6: Example of a plot of the "`channels`" container from the output file, after having run the code generated by WATCHMAN on different data samples. In this example the user defined 10 SUSY-like analyses, with different object selection and event selection cuts; and events passing the selection cuts defined in those 10 analyses have been flagged according to them.

```
##--- User-Defined Formula
userFormula = {
# Meff formula: Highest Pt 4 Jets + MET + All Leptons
'meff': {
'position':3,
'formula':
"""
meff = 0.
if len(candidates['jet']) < 3: return meff
for i,jet in enumerate(candidates['jet']):
    if i >= 3: break
    meff += getVal(candidates['jet'][i], 'Pt')
    pass
for i,el in enumerate(candidates['electron']):
    meff += getVal(candidates['electron'][i], 'Pt')
    pass
for i,mu in enumerate(candidates['muon']):
    meff += getVal(candidates['muon'][i], 'Pt')
    pass
meff += MET_corrected(candidates)
return meff
"""},
}
```

Listing 2: Example of custom user-defined formula to be used for cuts or to fill container in the output file.

will find – according to the example which we are considering – the events flagged according to the 10 analyses.

As already said, the user can define custom formulas to compute interesting quantities. This custom information can be stored in the output file, automatically computed with the right subset of selected particles, and flagged as any other object.

7. Conclusions

WATCHMAN presents and implements a new idea in the HEP field, the usage of Computer Aided Software Engineering to build reliable, easy to maintain and easy to validate data analysis code.

WATCHMAN is an analysis code construction kit, with which it's possible to handle many analyses at the same time and to generate the actual complete code, ready to be run. The framework also takes care of the specific data format setup, relieving the user of the need of learning the details of it. And it can be expanded with modular interfaces to work with new formats.

WATCHMAN is a new open-source Python project under continuous development, with an already first stable release; it has a small community of active users and 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 [13] and [12]).

Acknowledgements

We would 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 Störig for their extensive tests and for their suggestions. And many thanks also to others who contributed with suggestions, comments, hints or criticisms to the development of this package.

References

[1] "WATCHMAN – *An highly automated Analysis Code Generator*", `https://twiki.cern.ch/twiki/bin/view/Main/WatchMan`.

[2] R.M. Bianchi, R. Brunelière, "WATCHMAN *Project – An automated analysis code generator for High Energy Physics data analysis in the LHC era. Applying* CASE *to* HEP *analysis*", (*In preparation. It will be linked from the* WATCHMAN *website*).

[3] "*Delphes – A framework for fast simulation of a generic collider experiment*", `http://projects.hepforge.org/delphes/`.

[4] "*Worldwide* LHC *Computing Grid (*WLCG*)*", `http://lcg.web.cern.ch/LCG/` and `http://public.web.cern.ch/public/en/Spotlight/SpotlightGrid-en.html`.

[5] "*The* PANDA *Production and Distributed Analysis System*", `https://twiki.cern.ch/twiki/bin/view/Atlas/Panda`.

[6] "*CASE – Computer-aided software engineering*", `http://en.wikipedia.org/wiki/Computer-aided_software_engineering`

[7] R.M. Bianchi, R. Brunelière, S. Caron, "ATLASWATCHMAN – *An automated Analysis Code Generator, a D3PD Maker and a Jobs Bookkeeper*", `https://twiki.cern.ch/twiki/bin/view/Main/ATLASWatchMan`.

[8] S.P. Martin, "*A supersymmetry primer''*", pp. 88–94, `http://arxiv.org/pdf/hep-ph/9709356`.

[9] W. Lavrijsen, *"PyROOT – A Python-ROOT Bridge"*, `http://root.cern.ch/drupal/content/how-use-use-python-pyroot-interpreter`

[10] *"ROOT – An Object-Oriented Data Analysis Framework"* `http://root.cern.ch/drupal/`

[11] *"SFrame – A ROOT data analysis framework"* `http://sourceforge.net/projects/sframe/`

[12] *"ATLAS Plots on E_{CM} Dependence of Physics Reach"* `https://twiki.cern.ch/twiki/bin/view/AtlasPublic/AtlasResultsEcmDependence`. WatchMan was used to analyze data in order to produce the plots of the Supersymmetry discovery reach.

[13] *"Prospects for Supersymmetry and Universal Extra Dimensions discovery based on inclusive searches at a 10 TeV centre-of-mass energy with the ATLAS detector"* `http://cdsweb.cern.ch/record/1191916`