Implementing a Domain Specific Language to configure and run LHCb Continuous Integration builds

M Clemencic and B Couturier
CERN, CH-1211 Genève 23, Switzerland
E-mail: marco.clemencic@cern.ch

Abstract. The new LHCb nightly build system described at CHEP 2013 was limited by the use of JSON files for its configuration. JSON had been chosen as a temporary solution to maintain backward compatibility towards the old XML format by means of a translation function.

Modern languages like Python leverage on meta-programming techniques to enable the development of Domain Specific Languages (DSLs).

In this contribution we will present the advantages of such techniques and how they have been used to implement a DSL that can be used to both describe the configuration of the LHCb Nightly Builds and actually operate them.

1. Introduction
The new LHCb nightly build system[1] used JSON files for its configuration. The JSON format had been chosen as a temporary solution to maintain backward compatibility towards the old XML configuration files used in the old system.

The new system has several advantages with respect to the old one, like flexibility and modularity. In particular it can be used by any user or developer to checkout, build and test the whole LHCb software stack, independently of the regular automatic builds.

With experience we realized that the JSON configuration format of the new system, although more compact and simpler than the old one, was suffering the same limitations of the old XML files. For the build to work, the user had to fill the configuration with boilerplate details, and there is no easy way to know what can be done and how. Moreover XML and JSON were designed as exchange formats between applications, so they are not very practical to write and read for a human being.

To overcome the limitations, and in particular the user-friendliness, of the configuration files, we investigated possible implementations of a Domain Specific Language (DSL). Since the core of the Nightly Build System is written in Python[2] we opted for an internal DSL based on Python.

Working on the implementation of the DSL, it became clear very soon that having Python under the hood, the DSL could be used not only for the description of what should be built, but also to actually perform the build actions (checkout, build and test), at the point that the DSL becomes at the same time the configuration format and the engine of the build system.
Listing 1. Example of the look and feel we wanted to have for the configuration. In the LHCb nightly builds the software is organized in interdependent software projects, built together in a consistent set, called slot.

```python
# Import required classes
from LbNightlyTools.Configuration import Slot, Project
from LbNightlyTools import CheckoutMethods

# Basic usage
lhcb_head = Slot('lhcb-head',
    projects=[Project('Gaudi', 'head', disabled=True),
             Project('LHCb', 'head')],
    env={'CMTPROJECTPATH=${LHCBDEV}:${CMTPROJECTPATH}'}
)

# Customize and reuse
class Gaudi(Project):
    def checkout(self, **kwargs):
        CheckoutMethods.git('http://git.cern.ch/pub/gaudi.git',
            self.version)

class LHCb(Project):
    pass

class LHCb_HEAD(LHCb):
    version = 'HEAD'
    overrides = {'GaudiObjDesc':'r183453'}

class CMakeSlot(Slot):
    build_tool = 'cmake'

slot = CMakeSlot('special',
    projects=[Gaudi('v26r1'),
              LHCb_HEAD(),
              Project('Lbcom', 'HEAD')],
    env=['SPECIALSETTING=special'])

# Perform build actions
slot.checkout() # checkout all projects in a slot
slot.Gaudi.build(jobs=4) # build a specific project
```

Being able to write the configuration in Python and use the configuration objects to perform the build actions will open to the possibility of interactive build and test sessions, which were not possible before.

Here we will describe the development process we followed and the techniques used to implement specific features.

2. Definition of the requirements

The development of the DSL started with the definition of the requirements the DSL would have to satisfy. Since the focus was on usability and user-friendliness, we started by writing down examples of the configurations we would like to use in the final implementation, like in Listing 1, where we used the basic concepts of the LHCb nightly build system: software projects (releasable entities with dependencies between them) and slots (consistent stacks, dependency chains, of software projects). Of course, during the development we modified the examples whenever needed, either to improve the user experience or to simplify the implementation, still focusing on high usability.

After the definition of the user interface, we ensured that all the functionalities of the current system were represented, like

- fine tuning of the checkout
different checkout methods
overrides of parts of the software projects
• override of environment variables at the slot level
• support for different build systems
• flag software projects as disabled in a slot

In this way we guarantee that the new configuration system features a superset of the functionalities of the old JSON and XML files.

We can then consider new functionalities, like tuning the environment on individual projects, or extended project patching.

3. Implementing the DSL

In addition to the usual Object Oriented Programming (OOP) techniques, Python allows tuning of classes and objects behaviours via several hooks, each of them fit for a specific family of needs, so to implement the functionalities we need, we first need to identify the hook that can be used.

special methods make it possible to use custom objects as numbers, iterables, mappings, etc. descriptors are the Python extension to getters and setters methods; they allow to use simple assignment and retrieval of data members to trigger complex behaviours

metaclasses hook onto the different phases of the classes lifetime to modify them and introduce functionalities not explicitly coded in the class definition

decorators are syntactic sugar that allow easy flagging of free functions and member functions; they are typically used to flag methods or to introduce actions executed before and after the wrapped function

In the rest of the section we describe how the different hooks can be used to implement specific features and how they can be used in conjunction to improve user experience.

3.1. Virtual data members

Projects in a Slot instance are kept in a list data member. While this is practical for looping over all the projects (e.g. for the checkout) accessing a specific project by its name requires a dedicated getter method:

```python
class Slot(object):
    # ...
    def __getattr__(self, name):
        return self.project(name)
    # ...
```

This can be achieved implementing the `__getattr__` special method like this

```python
class Slot(object):
    # ...
    def __getattr__(self, name):
        return self.project(name)
    # ...
```

The `__getattr__` method is invoked by Python when trying to access a data member (or method) that is not defined in the object, so that writing `slot.Gaudi` is equivalent to write `slot.__getattr__('Gaudi')`. 

3
3.2. Mapping (dictionary) interface
The technique described above works only for keys that are valid Python identifiers, so if you need to access an element that contains, for example, a - in the name, you need another way.

We can leverage on another special method, \_\_getitem\_\_, used in Python to implement the square brackets operator. In this case the code \texttt{slot['Gaudi']} is equivalent to \texttt{slot.__getitem__('Gaudi')}, so what we need is

\begin{verbatim}
class Slot(object):
    # ...
    def __getitem__(self, key):
        return self.project(key)
    # ...
\end{verbatim}

It must be noted that, as tradition in Python, there is no constraint on the type of the argument \texttt{key}, so \_\_getitem\_ can be implemented to handle any type you may need (strings and integers are just the simplest cases).

3.3. Special handling of set and get
In our use case we want to be able to change the checkout method used for a project. Using the OOP paradigm, we can define a base class that uses a default implementation and specializations that override that method, then you have to instantiate objects of different classes depending on the method you want to use. Writing configuration files for the nightly build system is closer to \textit{interactivity} than to \textit{development}, so standard OOP techniques may be too verbose and we would like to write

\begin{verbatim}
from CheckoutMethods import git
lhcb = Project('LHCb')
lhcb.checkout = git
lhcb.checkout()
\end{verbatim}

where \texttt{git} is a function that accepts a \texttt{Project} instance as first argument.

Although it may seem that the above code would work fine, it actually fails with an error about the number of arguments passed to \texttt{git} (due to the differences between functions, unbound methods and bound methods in Python).

To correctly achieve the expected behaviour, we need to use a \textit{descriptor}, i.e. a special object with a \_\_set\_\_ and a \_\_get\_\_ method that will be automatically invoked by Python when trying to set or get the connected data member. As show in Listing 2, the \_\_set\_\_ method will keep a reference to the function in a private data member of the instance, and the \_\_get\_\_ method will return a \textit{bound method}, i.e. a function that passes the instance as the first argument of the wrapped function.

With the descriptor in place, we can simplify further users’ life, for example allowing something like

\begin{verbatim}
lhcb = Project('LHCb')
lhcb.checkout = 'git'
lhcb.checkout()
\end{verbatim}

by modifying \texttt{CheckoutDescriptor.__get__} like this

\begin{verbatim}
class CheckoutDescriptor(object):
    def __get__(self, obj, value):
        if isinstance(value, basestring):
            # use the function with the given name in the standard module
            value = getattr(CheckoutMethods, value)
        # store the function in a data member
        self.implementation = value
        # ...
\end{verbatim}
Listing 2. Implementation of a descriptor for the Project.checkout method.

```python
import CheckoutMethods

class CheckoutDescriptor(object):
    def __init__(self, method=CheckoutMethods.default):
        # initialize the descriptor with the requested function
        self.__set__(None, method)
    def __set__(self, obj, value):
        # store the function in a data member
        self.implementation = value
    def __get__(self, obj, objtype=None):
        import types
        # return a method bound to obj
        return types.MethodType(self.implementation, obj, objtype)

class Project(object):
    # ...
    checkout = CheckoutDescriptor()
    # ...
```

3.4. Solving the conflict between descriptors and inheritance

The descriptor used for the checkout method in the Project base class will simplify users’ life, with a small draw back: when specializing Project overriding the checkout method, the assignment hook will be overridden as well.

One way to overcome the limitation is to use CheckoutDescriptor as a decorator:

```python
class MySpecialProject(Project):
    # ...
    @CheckoutDescriptor
    def checkout(self):
        # delegate to a standard method
        CheckoutMethods.git(self)
        # and do something more
        open('version_info.txt', 'w').write(self.version)
        # ...
```

The decorator notation (prefix a function name with `@`) is equivalent (but more elegant) to writing

```python
class MySpecialProject(Project):
    # ...
    def checkout(self):
        # delegate to a standard method
        CheckoutMethods.git(self)
        # and do something more
        open('version_info.txt', 'w').write(self.version)
        # ...
```

Using decorators still relies on some explicit user action that is mandatory. We can achieve the same result with the help of a *metaclass*. In Listing 3 we show how we can automatically decorate a method with a given name.

The metaclass `__new__` hook is triggered after the class definition is parsed and before the object representing the class is instantiated, both for the base class and for all the derived ones. Users can now write

```python
class GitProject(Project):
    checkout = 'git'

class CustomCheckoutProject(Project):
    def checkout(self):
        # delegate to a standard method
```
Listing 3. Using a metaclass for projects to decorate ‘checkout’ methods. The declaration of `checkout = 'default'` in the project class is wrapped by the metaclass in a CheckoutDescriptor instance, which will map the string to a function as described in 3.3.

```python
# ...

class ProjectMetaclass(type):
    def __new__(cls, name, bases, dct):
        if 'checkout' in dct:
            # if we have a user-defined 'checkout' class member,
            # we wrap it with CheckoutDescriptor
            dct['checkout'] = CheckoutDescriptor(dct['checkout'])
            # continue with the standard class creation
            return type.__new__(cls, name, bases, dct)

class Project(object):
    __metaclass__ = ProjectMetaclass
    # ...
    checkout = 'default'
    # ...

CheckoutMethods.git(self)
# and do something more
open('version_info.txt', 'w').write(self.version)

p = CustomCheckoutProject('Test')
# override the special checkout method
p.checkout = 'svn'
```

4. Conclusions
We used the techniques described in this paper to implement a Python-based internal Domain Specific Language used to configure and operate the LHCb Nightly Builds System. The obtained DSL is so neat and straightforward, that it can be used effectively in interactive build sessions. The examples shown are meant to give an idea of the potential of Python as the engine for an internal DSL, and may not correspond to the actual implementation used in LHCb.

References
[1] Clemencic M and Couturier B 2014 J.Phys.Conf.Ser. 513 052007
[2] Python Programming Language – Official Website URL http://python.org/