SLAC-TN-11-005

PyDecay/GraphPhys: A Unified Language and Storage System for Particle Decay Process Descriptions

Jesse N. Dunietz

Office of Science, Science Undergraduate Laboratory Internship (SULI)

Massachusetts Institute of Technology

Stanford Linear Accelerator Center

Stanford, CA

August 20, 2010

Prepared in partial fulfillment of the requirements of the Office of Science, Department of Energy's Science Undergraduate Laboratory Internship under the direction of Matthew Bellis with the BaBar team at the SLAC National Accelerator Laboratory.

Participant: _____

Signature

Research Advisor: _____

Signature

Work supported in part by US Department of Energy contract DE-AC02-76SF00515.

TABLE OF CONTENTS

ABSTRACT

PyDecay/GraphPhys: A Unified Language and Storage System for Particle Decay Process Descriptions. JESSE N. DUNIETZ (Massachusetts Institute of Technology, Cambridge, MA 02139) MATTHEW BELLIS (BaBar team at the SLAC National Accelerator Laboratory, Stanford, CA 94025)

To ease the tasks of Monte Carlo (MC) simulation and event reconstruction (i.e. inferring particle-decay events from experimental data) for long-term BaBar data preservation and analysis, the following software components have been designed: a language ("GraphPhys") for specifying decay processes, common to both simulation and data analysis, allowing arbitrary parameters on particles, decays, and entire processes; an automated visualization tool to show graphically what decays have been specified; and a searchable database storage mechanism for decay specifications. Unlike HepML, a proposed XML standard for HEP metadata, the specification language is designed not for data interchange between computer systems, but rather for direct manipulation by human beings as well as computers. The components are interoperable: the information parsed from files in the specification language can easily be rendered as an image by the visualization package, and conversion between decay repesentations was implemented. Several proof-of-concept command-line tools were built based on this framework. Applications include building easier and more efficient interfaces to existing analysis tools for current projects (e.g. BaBar/BESII), providing a framework for analyses in future experimental settings (e.g. LHC/SuperB), and outreach programs that involve giving students access to BaBar data and analysis tools to give them a hands-on feel for scientific analysis.

INTRODUCTION

When physicists want to analyze data from a collider experiment such as BaBar, they generally do so in two steps. First, they must simulate the decay reactions they are looking for so they know what current theories predict, and second, they must analyze the collider data so that they can compare actual results with theoretical predictions. For the first step, they need to specify the decay processes to simulate and the parameters – branching fractions, angular distributions, etc. – under which these processes should be simulated. For the second, they need to specify various parameters describing which events to search for amidst the petabytes of experimental data, which algorithms to use for reconstructing particle decays from detector data, which particle attributes to extract from the detector data, etc.

Currently these are specified in custom textfile formats, with the simulation and data-extraction formats sharing nothing. These configuration file formats are not very user-friendly, and can be so time-consuming to learn that anyone who wants to write such a file generally simply copies an old one and modifies it. It is also often difficult to read out from such a file what decay it is actually analyzing; if that information is directly and clearly present, it is only in comments. Automated visualization of the decay is certainly not possible.

This arrangement is not ideal for a number of reasons. First, there is a decent chance that what an analyst has specified in a configuration file is not actually what he or she intended, and there is no way short of running the analysis or simulation to determine that. Additionally, it is difficult to introduce new collaborators to the project, since the learning curve for using the configuration files is so high. This is particularly problematic given that the SuperB collider, if it becomes a reality, will want to leverage BaBar's tools, but the SLAC researchers currently working with these obscure tools may no longer be available to explain the system at that point. There is also no easy way to search these files for particular decay

processes, and thus no easy way to know that the process one is examining has not been examined already. In fact, some of the configuration files are not even stored in a central, publicly available location. This lack of centrality and searchability can lead to duplication of effort. Finally, some BaBar team members have been advocating releasing some BaBar data to the public for use in classrooms, so that students can acquire a sense of what it is like to work with real physics data, and hopefully increase their excitement about and sense of participation in genuine physics. This of course requires that the tools for analyzing the BaBar data be intuitive and accessible to people outside the laboratory.

To address these problems, it was determined that a new system for specifying and storing decay information was necessary. The system would need to allow simple, intuitive input of decay information, without strange or obscure format features, to eliminate the high learning curve of the current system; easy visualization of the contents of such a configuration file, to allow ensuring the correctness of the input; and a searchable long-term storage system to allow collaborative sharing of and searching for decay descriptions. A system with all of these capabilities and characteristics was implemented in the course of this project.

SYSTEM IMPLEMENTATION

The entire system was implemented in Python. Each of the functionalities listed above was implemented in a separate module. All components of the system assume that the physical processes under study can be characterized as a tree of particles, where the root of the tree is the initial particle and the children of each particle-node are the decay products of that particle. This model is somewhat complicated by the probabilistic nature of decays, as discussed below.

Decay Process Internal Representation

A set of Python classes was created to represent physical decay processes and groups of such processes. In terms of the tree model described above, every node (particle) in the tree has a type, a list of parameters – arbitrary name-value pairs – and a set of associated decays. These decays are essentially lists of product particles, along with associated parameters – again, merely name-value pairs. There is also a class representing a group of unrelated particles, i.e. a forest of such particle-decay trees, which can also contain general parameters pertaining to the whole simulation/reconstruction analysis.

The probabilistic nature of decays complicates the tree representation somewhat. For instance, if a K^0 can decay to either two π^0 particles or a π^+/π^- pair, this effectively means there are two alternative trees for that K^0 particle. This is generally represented as alternative child-node sets for the particle object. For example, $D^+ \to K^*\pi, K^* \to \pi\pi$ internally has only one object representing the K particle, with two alternative subtrees. However, it is sometimes useful, particularly for visualization, to have a single unambiguous tree with no probabilistic alternatives. For this purpose methods have been implemented to "flatten" the alternative trees into a set of full decay path trees. Thus, in the above example, this flattening would produce two trees, as illustrated in Figure 1. A noteworthy feature of this flattening is that, if the database (see below) contains branching fraction information for the decays under examination, the product branching fraction – that is, the probability of the entire flattened decay tree – can be computed in the processes of flattening the tree.

In many cases an analyst would like to specify that a particle should be allowed to decay "generically" – that is, it should decay according to the standard tables of decays and associated probabilities. A decay with no products is used to indicate such a generic decay.

These Python objects are the basis for the other components, which all use this object-based representation of decays. The objects can of course be created directly in Python code, but it is often easier obtain them from a database or "GraphPhys" representation.

GraphPhys Decay Description Language

A decay description language, christened "GraphPhys", was created to allow specification of decays with various parameters to be written down in a simple, easily readable form. GraphPhys is essentially a subset of the Graphviz DOT graph description language. The abstract grammar for GraphPhys can be found in Figure 2.

Terminals are given here as single-quoted strings. Elements in brackets are optional; parentheses indicate grouping. Much as in Dot, whitespace is always ignored, and an ID is one of the following:

- Any string of alphabetic (`[a-zA-Z\200-\377]`) characters, underscores ('_') or digits (`[0-9]`) (unlike in Dot, an ID **may** begin with a digit);
- A numeral (`[\-]?(.[0-9]+ | [0-9]+(.[0-9]*)? )`);
- Any double-quoted string (`"..."`) possibly containing escaped quotes (`\"`)

Unlike Dot, GraphPhys does **not** allow HTML strings as IDs.

In GraphPhys, one particle decaying to another set of particles is represented by the name of the initial particle and an arrow, followed by a curly-brace-enclosed list of decay products. Figure 3 demonstrates this syntax with a sample GraphPhys file that describes the aforementioned D^+ decay with some additional parameters. The `simulator=JETSET` line is an example of setting a default parameter for all decays; the same can be done for particles. The `fraction` parameter, though not demonstrated in this example, is a special parameter used to indicate decay branching fractions: the simulator could explicitly told the probabilities for the $K^* \to \pi^+\pi^-$ and $K^* \to \pi^0\pi^0$, although these probabilities could alternatively be fetched from the database (see below). This is one of the few parameters whose meaning is fixed by the GraphPhys system. The `ChargeConj=True` line demonstrates how global parameters can be configured for the entire set of decays described in the file.

Database Structure and Implementation

In addition to the internal Python-object representation, a database representation was designed and implemented for storing decay specification information in a searchable, scalable database. The hope is that researchers in a group or students in a school will be able to create local databases of stored decay specifications so that they can see what others have looked at. This is in part motivated by a desire to replace the current `.DEC` and `.tcl` files, used for configuring BaBar's MC simulation and reconstruction software, respectively. These files tend to be stored by the thousands alongside each other, sometimes under a user's home directory rather than in some shared location. Their filenames often reflect their contents, but they are still far from easy to search.

The Object-Relational Mapper (ORM) component of the Django web application framework [1] was used to define the structure of the database (though this structure can of course be dumped as generic SQL) and to manipulate it programmatically. This abstraction layer makes migration between database engines trivial and makes querying the database far easier.

The database was initially designed only to store the information contained by GraphPhys files, but it quickly became clear that it would be useful to store generic known particle/decay information as well. Broadly speaking, then, the database design can be split into two sets of tables:

1. **Particle/Decay Types:** The tables for storing the names and properties of known particles and decays. These tables exist to encapsulate some of the information contained in the publications of the Particle Data Group (PDG), the premier publisher of particle and decay data, in a computer-readable, searchable format that is also well-integrated with the other half of the database. The tables in this group are:

    - `pydecaydb_particlebasetype`: Contains the masses, charges, PDG ID numbers,

etc. of various types of "base" particle types, i.e. type information excluding information about charge conjugation.

- `pydecaydb_particletype`: Contains actual particle types, referencing base types and including charge-conjugated names. This table and the previous onecan be automatically populated from the PDG's CSV file [2] of particle data.
- `pydecaydb_decaymode`: Identifies standard known decay modes. The table itself stores only the initial particle type foreign key and the branching fraction of the decay mode.
- `pydecaydb_productsetmembership`: Associates particle types with decay modes. Each particle type/product set association also records a count of how many particles of that type are present in that product set.

2. **Particle/Decay Instances:** These tables store information pertaining to specific particles or decays that users have input using the PyDecay system – perhaps through GraphPhys, perhaps by direct Python object creation. Every time a user specifies that there is, for instance, a π^+ particle somewhere in his or her decay tree, this can be thought of as declaring a new π^+ instance, and it will be recorded as such if the decay tree is stored in the database. The tables in this set are:

   - `pydecaydb_particleinstance`: Stores the foreign keys for the type, parent particle (if any), and instance group (see below) of each particle instance. Because charged types are stored sign-independently, the entries in this table also indicate whether the particle is positively or negatively charged.
   - `pydecaydb_decayinstance`: Stores the foreign key for the initial particle of each decay instance. Products can be inferred by looking for particles whose parent is the initial particle. The decay mode can be inferred from initial instance type, product types, and intermediate angular momentum (if specified).

- `pydecaydb_instancegroup`: Provides group ID's for particles to identify themselves with. Each group ID indicates a group of unrelated particles – for example, a set of unrelated root particles from a single GraphPhys file.

Each of these tables also has an associated table for storing instance parameters as name-value string pairs.

Various functions for searching for and manipulating entries in the database in an object-oriented manner have been implemented in Python using the Django system.

While these tables constitute the primary database system in the PyDecay framework, a need was also recognized for allowing use of the framework without the heavyweight requirements of Django and a relational database system. To that end, an abstract interface to the particle type/decay mode segment of the database was created to allow alternative implementations for storing/fetching such data. Two alternative implementations are provided besides the Django implementation: a null implementation that always returns no results when retrieving particle/decay data, and a system that uses Python dictionaries to specify particle/decay information. A library-global settings file is used to specify which database implementation to use.

Visualization/Conversion Tools

A visualization/conversion package was created for converting between various representation formats. The following representations of a decay tree are interchangeable, i.e. any one of them can be converted in to any other one (via PyDecay objects, in some cases):

1. **GraphPhys** – a string of text conforming to the GraphPhys language specification.
2. **PyDecay objects** – Python objects from the PyDecay library discussed above.
3. **Database instances** – rows from the second class of database tables described above, represented in code by Django Python objects.

4. **Database types** – rows from the first class of database tables described above; converting to this representation is primarily useful for inserting new decays into the decay modes table.

In addition, any of the above representations can be converted via PyDecay objects into any of the following alternative representations, though the reverse conversion is not possible:

1. **GraphViz diagram** – a visualization of decay trees illustrating decays as arrows between boxes that contain particle names and parameters and decay parameters. The visualization is generated by the GraphViz diagram-layout system [3] via the Pydot Python package [4].

2. **PyFeyn diagram** – a visualization of decay trees as "cartoon" decay diagrams, rendered using the PyFeyn Feynman diagram-drawing package [5]. Unlike the GraphViz visualizer, this visualizer does the layout internally, since PyFeyn does not do it automatically.

3. **tcl file** – a `.tcl` file in the same format as those used to configure BtaTupleMaker, the BaBar reconstruction software. This converter is a proof-of-concept to show that the PyDecay libraries can be used to create tools as powerful as the current ones but which can be used much more easily.

The conversion framework was designed to be sufficiently generic that conversion to new formats can easily be integrated. Conversion is performed by "converter" objects which all support the same conversion API. This would even allow building a system in which the precise type to be converted to was specified in a config file via the qualified name of the converter class to use.

DISCUSSION AND CONCLUSIONS

PyDecay is a set of libraries for use in the creation of analysis tools. While it does not itself perform the analysis, the functionality it provides should make the creation of such tools almost trivial, thus accomplishing the major goals of the project.

As a proof of concept, a minimal Monte Carlo decay simulator was modified to use GraphPhys files as its input format. A kinematics checker was also implemented for checking that a decay specified in the GraphPhys format is kinematically possible. This latter functionality is particularly useful given that currently this type of check is typically done painstakingly by hand. The checker works with very minimal GraphPhys files, since the masses of the particle types can be loaded from the database.

The niche filled by PyDecay is, as far as we know, unique. There have been attempts to create a common XML dialect for HEP decay information, called HepML, which appears to be a similar concept to GraphPhys. This specification, however, appears to be for data *interchange* between software packages [6], e.g. for passing configuration parameters around between two Monte Carlo simulators in the same toolchain or in different projects. As an interchange format, HepML's requirements are fairly different from GraphPhys's. GraphPhys is meant to be an input format: GraphPhys files are human-editable textfiles which easily yield their meaning to another human on inspection, as well as being computer-readable. As anyone who as written XML by hand knows, XML files are neither easily writable nor easily readable by humans. In fact, the PyDecay converter framework could easily be used to write a converter between PyDecay objects and HepML strings.

As far as the database component, we know of no other HEP project that has created a similar searchable database of researchers' past decay specifications, nor even one that allows searching for known general particle/decay data. There are existing databases of such general data, most notably the PDG; unfortunately, the PDG does not make this database

available for public access. The Durham HepData project [7] allows searching for decay process information by inputting a text string describing the decay, but the information returned cannot be manipulated programmatically; the system simply returns a list of links to publications containing relevant information. PyDecay's database module is much more flexible, allowing queries such as "What are all the decay modes with any initial state and a final state of exactly 3 pions?"

The PyDecay suite has been demonstrated to several groups of potential users, generally getting a positive response. All the potential users who examined it have indicated that they would likely find it useful, and some indicated an interest even in just the minimal proof-of-concept tools that have been implemented so far.

Some work still remains to be done, however. Most notably, it is not currently possible to represent a certain type of decay specification that is sometimes useful to analysts. The problem occurs when analysts would like to specify a decay such as "X decays to Y, Z, and some other unspecified particles which are not of interest." Currently there is no way in any of the decay tree representations to capture the idea of "other unspecified particles." The feedback we have received indicates that this is something that analysts do often want to specify, making this functionality gap a potentially serious deficiency. On a similar note, the PDG lists several "inclusive modes" – modes including unspecified particles as described above. Thus, such underspecified decays may be present not just in an analysis but even in PDG standard data. These cannot currently be represented in the database, though some preliminary work has been done on figuring out how they could be. A few minor issues have also been identified with the GraphPhys syntax.

Despite these shortcomings, we feel that the functionality provided by PyDecay is sufficiently compelling that the system will be adopted in both experimental and outreach settings. We have made the tools and code publicly available, thus enabling adoption and improvement in any setting in which the libraries might prove useful.

ACKNOWLEDGMENTS

I would like to thank Matt Bellis, my mentor, for all his passionate support, guidance, and teaching. He gave the project direction and enabled it to go as far as it did. I would also like to thank Steve Rock, the SLAC SULI Program Director, and the other SLAC SULI administrators who made the program run smoothly. Additional thanks to the SLAC National Accelerator Laboratory, the Office of Science, and the Department of Energy for running the SULI program. Special thanks to all those BaBarians who listened to our ideas, tested out the software, and gave us feedback.

REFERENCES

[1] (2010, May) Django: The Web framework for perfectionists with deadlines. [Online]. Available: http://www.djangoproject.com/

[2] (2008) Masses, Widths, Quantum Numbers (IGJPC) and MC ID Numbers from 2008 Edition of RPP. [Online]. Available: http://pdg.lbl.gov/2010/mcdata/mass_width_2008.csv

[3] (2008, Apr.) Graphviz – Graph Visualization Software. [Online]. Available: http://graphviz.org/

[4] (2007, Oct.) Pydot. [Online]. Available: http://dkbza.org/pydot.html

[5] (2007, Sept.) PyFeyn. HepForge. [Online]. Available: http://projects.hepforge.org/pyfeyn/

[6] (2008, Mar.) HepML. HepForge. [Online]. Available: http://projects.hepforge.org/hepml/

[7] Durham Reaction Database. The Durham HepData Project. [Online]. Available: http://hepdata.cedar.ac.uk/reaction

[8] (2008, Aug.) The DOT Language. [Online]. Available: http://www.graphviz.org/doc/info/lang.html

FIGURES

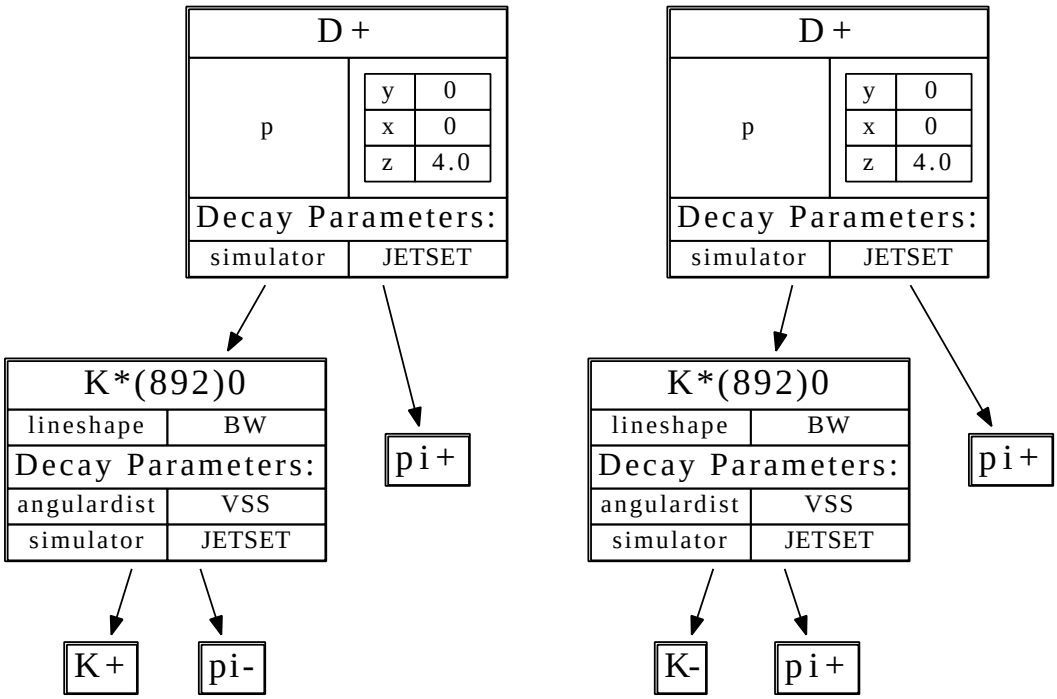

Figure 1: An example of how one decay tree with alternatives can be split into several distinct trees, (produced using the PyDecay Dot visualization package).

```
stmt_list      : [ stmt ';' [ stmt_list ] ]
stmt           : node_stmt
                     |       edge_stmt
                     |       default_stmt
                     |       param_stmt
default_stmt   : ('particle' | 'decay' ) param_list # Like 'node' and 'edge' in Dot
param_list     : '[' [ param_sequence ] ']'
param_sequence : ID [ '=' param_val ] [ ',' ] [ param_sequence ]
param_val      : ID | param_list | float_number
edge_stmt      : ID edgeRHS [ param_list ]
edgeRHS        : edgeop node_set
node_stmt      : ID [ param_list ]
node_set       : '{' id_list '}'
id_list        : ID [id_list]
param_stmt     : ID '=' param_val
```

Figure 2: The GraphPhys abstraction grammar. This is a simplified and slightly modified version of the standard DOT grammar[8].

```
decay [simulator=JETSET];
ChargeConj=True;

"D+" [p=[x=0, y=0, z=4.0]];
PiPlus1 [type="pi+"];
PiPlus2 [type="pi+"];
"K*(892)0" [lineshape=BW];

"D+" -> {"K*(892)0" PiPlus1};
"K*(892)0" -> {"K+" "pi-"} [angulardist=VSS];
"K*(892)0" -> {"K-" PiPlus2} [angulardist=VSS];
```

Figure 3: A sample GraphPhys file describing two possible decay paths for a D^+, with some additional parameters for simulation.