

DChain - Combinatorics made easy

Simon Patton¹ and Christopher D. Jones²

¹*BaBar Collaboration, Lawrence Berkeley National Laboratory, Berkeley, CA*

²*CLEO Collaboration, University of Florida, Gainesville, FL*

Abstract: This paper introduces the C++ package called DChain which provides a set of STL like classes that can be used to create the combinatorics normally associated with decay chains in High Energy Physics experiments.

1 Introduction

DChain was developed as a response to the question “What is the impact of object oriented programming on physics analysis?”. While much time and effort had been put into developing object oriented solutions for online systems and event reconstruction, there have been few offerings that have had a direct impact on a physicist’s analysis code. Therefore object oriented principles were applied to design an algorithm for candidate particle creation and this design resulted in the classes that make up the DChain package

The aim of this paper is to introduce the reader to the basic ideas that are the foundation of the package. Equipped with these idea, the features of the package are discussed and the reader is encouraged to “take it home” and try the code from themselves by using the stand-alone `tar` file that is available[1].

2 Objective

Experience with object oriented approached to software has shown it to be a powerful and useful way of developing software. However many physicist are more comfortable using a procedural approach when creating analysis programs. To encourage physicists to adopt an object oriented approach it seemed appropriate to develop a package that could demonstrate the power of object oriented programming in an area with which they were already familiar, i.e. physics analysis.

The result of this was the development of the DChain package which applies the object oriented approach to the problem of reconstructing decay chains. The classes in the package are designed to encapsulate the repetitive bookkeeping that is a feature of the procedural approach. This encapsulation should lead to clearer code making the purpose of a program more obvious, rather than having it hidden by the mechanics of the combinatorics. Moreover with the mechanics no longer cluttering the program, it should be much easier to spot errors in the physics logic.

3 Interface

To understand the interface used by DChain it is helpful to consider how a typical procedural analysis reconstructing decay chains is written. It would normally start out with a group of `DO` loops (if the language is FORTRAN) that iterate over the visible evidence of

the detector, e.g. tracks and showers, and these would be used to create some initial candidate decay products. Then, for each step in the decay chain, there is a set of nested DO loops that creates all the possible combinations of the decay's products. Within these nested loops there must be a series of tests to check that the combination, or partial combination, is consistent with the decay in question. These tests may include: checking the charge combinations; checking that a candidate appears only once in a chain and checking that the kinematics are consistent with the decay.

What should be clear from this is that there are a number of repeatable patterns in such code and these are the initial seeds for an object oriented solution.

The conceptual model of a decay is usually thought of as a particle X decaying into particles A and B and reconstruction of the parent can be thought of as the addition of its decay products. This can be written as follows:

$$A + B = X.$$

However, as discussed above, the execution model physicists typically use is that of combining lists of products to create a list of potential parent particles. This can be written as follows:

$$\text{List}(A) * \text{List}(B) = \text{List}(X).$$

The * symbol is used here to signify the combination of the two lists and this choice will be explained shortly.

It was decided that the DChain package should be based on the execution model rather than the conceptual model as this was the more familiar approach for the physicist and so it would be easier to learn and use. Therefore the core of the DChain interface is based around lists of candidate particles¹.

The contents of the lists in the DChain package are candidate particles. While these particles need some minimal behavior to allow them to be handled by the list, the core of the functionality is specific to the experiment at which the analysis is being executed. Therefore it was decided that the candidates themselves should not be part of the package but should be supplied by the experiment and that the package should provide templated list classes. This approach allows the candidate classes to be tailored to the specific need of the experiment.

Earlier the * symbol was used to represent the combination of two lists. The reason behind this choice can be seen by considering the idea of a multiplication “box”. Figure 1 shows an example of two such boxes. Figure 1(a) demonstrates the case of $4 * 2 = 8$ (where the C++ symbol * is used to indicate multiplication.) and Figure 1(b) shows all possible combinations of π^+ and K^- for the case of combining a list containing four π^+ 's with a list containing two K^- 's. The similarity between these two boxes shows that the * symbol is a good representation for combining lists in a computer program².

Whenever a list is filled, either by using the visible evidence of the detector or by combining candidate particles together, there needs to be some way for a physicist to define which candidates are “good” and which are “bad”, i.e. no use to their analysis. With-

1. One way of extending the current DChain package would be to implement an layer on top of the list interface that is constructed to reflect the conceptual model of decays.

2. The + operator is planned to be available in a future release of DChain and it will allow two lists of candidates to be merged.

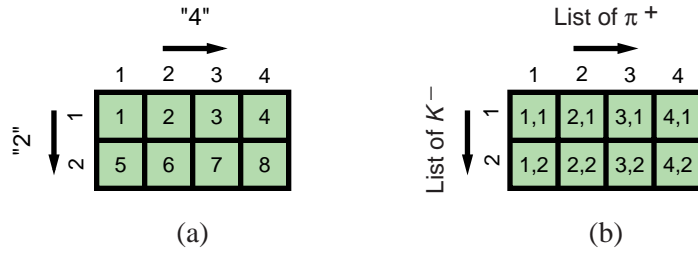


Figure 1: The multiplication boxes for (a) $4 * 2 = 8$, and (b) a list for four π^+ combined with two K^-

out this ability to select candidates a lot of the power of the DChain package is lost as the physicist would still have to iterate over the contents of any list.

Candidate selection is implemented by allowing each list to be passed a selection object when it is created. Each potential candidate is passed to this object and if the object returns `true` then the candidate is included in the list, whereas if it returns `false` the candidate is discarded. The selection objects can take one of two forms. It can be a simple global function with the following signature:

```
SelectionResult function( Candidate& )
```

where `function` is the name of the function and `Candidate` is the candidate class provided by the experiment. Alternatively it can be a subclass of the abstract class `SelectionFunction`, which is a functional object and is dealt with in more detail later.

Figure 2 shows an example of creating a list of candidate D^0 's that decayed into

```
//
ChargedPionList pions( defineChargedPion );
pions = trackTable ;
//
ChargedKaonList kaons( defineChargedKaon );
kaons = trackTable ;
//
DecayList dZeros( definedDZero );
dZeros = pions.plus() * kaons.minus() ;
```

Figure 2: A example DChain snippet that shows the creation of a list of candidate D^0 's by combining a list of pions with a list of kaons

pions and kaons. The first two pairs of statements are examples of creating the necessary candidate decay products. The `defineChargedPion` and `defineChargedKaon` elements are selection objects for selecting appropriate candidates. The `trackTable` element is an instance of the list of tracks in the event, which is the visible evidence from the detector that is used in this example to build charged candidates.

The first statement in the final pair defines a `DecayList` which is created with the `definedDZero` selection object. The decay channel itself is defined in the second statement of this pair. The `plus` function of the `pions` object selects the π^+ candidates in the list,

which can contain both positive and negative candidates, while the `minus` function of the `kaons` object selects the K^- . Each π^+ are now combined with each K^- to create all the possible D^0 candidates, which are then passed to the `definedDZero` selection function to be either placed in the `dZeros` object or thrown away. The combination of π^- and K^+ candidates will be discussed in the next section.

4 Features

Now that the interface of the DChain package has been explained, the in-built feature of the package can be introduced.

Charged Conjugation: At the end of the previous section the code example used to create a list of D^0 candidates was discussed. However, most analysis are not just interested in a particular decay but also its charged conjugate, therefore the lists in DChain are designed to create both conjugations of a decay chain. This means that the `dZeros` object in the example in Figure 2 not only contains D^0 's from π^+ and K^- , but also \bar{D}^0 's from combinations of the π^- and K^+ candidates in the `pions` and `kaons` objects. This saves the physicist having to write code for both decay chains.

Self-Conjugation: There are occasions where there is not a conjugate decay chain because the decaying particle is self-conjugate. DChain is designed to recognize two such occasions and on these occasions the conjugate list is not produced as it would only duplicate candidates. One occasion is when all the decay products are self conjugate, e.g. $\pi^0 \rightarrow \gamma\gamma$, the other is when each of the decay products can be matched to another decay product that comes for the same list but with the opposite conjugation. The following line of code is an example of this second occasion.

```
K0L = pion.plus() * pion.minus() ;
```

One point of caution should be noted here. If there are two lists that contain the same type of candidates, e.g. two pion lists, a positive candidate from one of these lists will not be matched with a negative candidate from the other list when determining self-conjugation. The reason for this behavior is that both lists may not contain an identical set of candidates, because they use different selection objects, and this will effect the two conjugate decay chains differently so both must be constructed.

Avoiding Double Counting (Type I) and Efficient Looping: There are a number of ways candidates can be double counted, e.g. creating a list twice as covered in the previous section on self-conjugation. Another way is for a list, with the same conjugation, to appear more than once in a decay chain. The following line of code shows an example of this is for a D^+ decay chain.

```
d3plus = pion.plus() * pion.plus() * kaon.minus() ;
```

Figure 3(a) uses a multiplication box to illustrate the double counting in this decay where there are four π^+ in the list of pions. The combinations in the upper left are duplicates of the combinations in the lower right and therefore should not be created. Also the leading diagonal of the box uses the same candidate twice and similarly these combinations should be ignored. The lists in DChain can detect when a list is being used twice and on these occasions the second pass (and any subsequent passes) is begun at the correct entry so that the

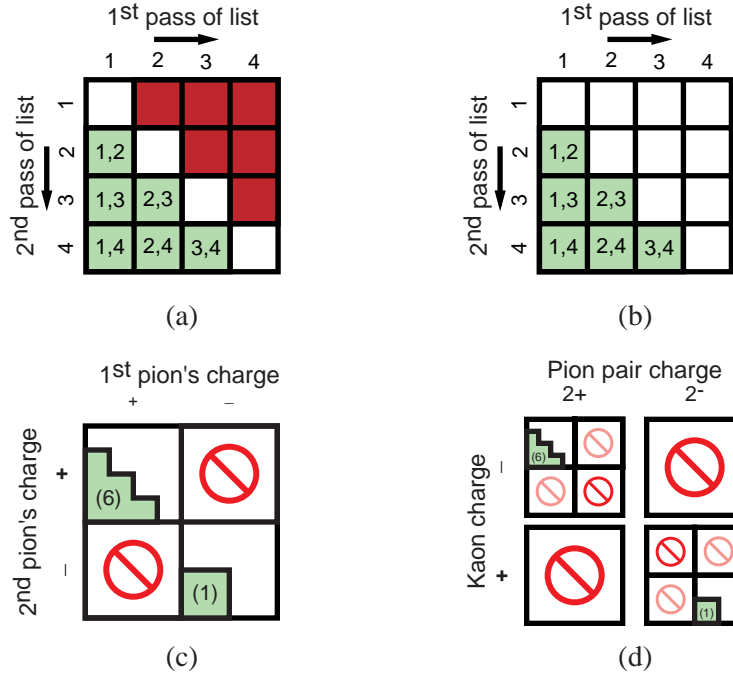


Figure 3: Illustrations of how avoidance of multiple counting (type I) and efficient looping are handled by DChain in the decay $D^+ \rightarrow \pi^+ \pi^+ K^-$ where there are four π^+ and two π^- in the list of pions.

repeated combinations are never created. Figure 3(b) shows this as only the shaded elements are created.

Figure 3 also illustrates how DChain makes sure that loops are efficiently executed. Figure 3(c) shows that because the lists know their elements conjugation the “wrong” conjugation pairs are automatically avoided. Figure 3(d) shows that for each $K^+(K^-)$ in the kaon list only 6(1) candidate D^0 's are created instead of the possible 36.

Avoiding Double Counting (Type II): Another possible way to double count is to use the same candidate, or same visible evidence twice when creating a new candidate. An example of this would be the decay chain $D^{*+} \rightarrow D^0 \pi^+$ where $D^0 \rightarrow \pi^+ K^-$. Clearly the same pion that was used to create a candidate D^0 should not be used to with that candidate when attempting to create a candidate D^{*+} , neither should the pion be used if it is made from the same visible evidence that was used to create the kaon in the D^0 . To avoid this problem the candidate class needs to supply an `overlaps` function the returns `true` if the candidate and that supplied as the argument are the same or have products or visible evidence in common.

5 Extracting Results

Once the lists of candidate decays have been created there needs to be a mechanism to extract information from these candidates. The lists in DChain have iterators that have been modeled after those used in the STL and these can be used, in conjugation with `for` loops, to extract information. However in most cases the physicist wants to extract infor-

mation for every element in the list, therefore an `iterate` function is provided which will pass each entry to the analysis object which is passed as the argument to the `iterate` function. The analysis object can take one of two forms. It can be a simple global function with the following signature:

```
void function( const Candidate& )
```

where `function` is the name of the function and `Candidate` is the candidate class provided by the experiment. Alternatively it can be a subclass of the abstract class `AnalysisFunction`, which is a functional object and is dealt with in the next section.

As well as iterating through the whole list it is possible to iterate through just one conjugation of the list with the `partial_iterate` function, which will iterate the primary conjugation of the list³

6 Functional Objects

Both selection and analysis objects can be either global functions or functional objects. A functional object is a class that implements the `operator()` member function. In the case of selection objects the function's signature is:

```
SelectionResult operator()( Candidate& )
```

while for analysis objects the function's signature is

```
void operator()( const Candidate& )
```

The benefit of this type of object compared to a global function is that as a class it can be assigned its own context, and not just use the global context. This means more than one instance of the class can exist and each can be tailored to its particular purpose. A simple example of this would be an analysis object that selects the hardest candidate. Two different instances of this class can be used on different lists and because they each have their own context they will not interfere with each other as a global function might.

This type of objects can be very powerful and are likely to be a key feature in any sophisticated analysis.

7 Summary

This paper has introduced the concepts behind and features of the DChain package which applies an object oriented approach to the problem of reconstructing decay chains. The resulting package demonstrates to physicists the power of object oriented programming in an area with which they were already familiar and hopefully encourages them to adopt this type of approach in their future analysis programs.

More details about DChain can be found elsewhere[2]

References

[1] <http://www.slac.stanford.edu/~patton/dchain/dchain-example.tar.gz>

[2] <http://www.slac.stanford.edu/~patton/dchain/>

3.The bar function returns a list whose contents is the same as the original list, except with the primary conjugation changed.