LHCb-INT-2022-004
October 17, 2022

Decay finder algorithm for reconstructed particles in Run III at the LHCb experiment

Sacha Barré[1], Abhijit Mathad[2].

[1] *The University of Manchester*
[2] *University of Zurich*

Abstract

The LHCb software is being adapted to tackle the challenges associated with the Run III of LHC. As a result, a vectorised reconstruction event model is planned to be introduced for parallel processing of Run III data. As a consequence, the software used for offline processing of data is also being updated. Shipped as part of the new suit of upgrades is a new decay finder algorithm. This algorithm improves upon the old ones used in the Run I/II periods, particularly in handling of identical particles and improvements with added flexibility in the syntax of the user-specified decay descriptors. The structure of the new decay finder, which includes templated functions, makes it agnostic towards the type of the input particle. It is designed in-line with modern software practices through systematic implementation of unit tests and detailed documentation/comments to ensure good coverage of the features and code maintainability.

1 Introduction

As a consequence of the increase in luminosity and a fully software-based trigger in Run III of the LHC, a factor 30 increase in the rate of data collected by LHCb is for-seen compared to Run II [1]. Vectorized event model and highly efficient selection algorithms, based on ThOr functors, are therefore being developed. Offline analysis tools are also being adapted to tackle the challenges of Run III. Coming as part of the upgrades, is a new decay finder algorithm.

The previous decay finder algorithms, particularly the LoKi-based [2], cannot be trivially modified to accommodate the new Run III event model. Therefore, a new decay finder algorithm is required that is capable of handling both the new and the old reconstruction event models. Development of such a new algorithm would also help in addressing any deficiencies faced by the previous algorithms.

In this note, the purpose of the decay finder is discussed in section 2, the parsing of the user-specified decay descriptors is discussed in section 3, the implementation details of the decay finder interface is discussed in section 4, the future developments are discussed in section 5 and finally the conclusions are presented in section 6.

2 What is a decay finder ?

A decay finder is a C++ tool that takes as input user-specified decay descriptor[1] and a container of reconstructed particles, compares the decay structure of each of the reconstructed particle to the one specified by the decay descriptor and returns the head of the decays when there is a positive match. Users can also mark the decay descriptor with carets symbol ("^") to return one of the daughters instead of the parent particle. As an example, in Run III, the decay finder is used by the `FunTuple` algorithm to select reconstructed particles and store their kinematic or topological information in a ROOT file. In Fig. 1, a code snippet is shown where we store the kinematic information of particles involved in the decay of $B^0 \to K^- K^+$.

3 Parsing of user-specified decay descriptors

The first step towards building a decay finder is to be able to parse the decay descriptors with arbitrary complexity. For this reason, descriptors are parsed recursively using regular expression[2] and with `boost::regex` [3] library[3]. Internally, the parsing of the decay descriptor handles three different scenarios that are recursively called. The details about each of the scenario is discussed in the appendix A.

The parsing of the decay descriptors involves construction of dedicated C++ objects that are arranged in a tree structure representing the decay, as depicted in Fig. 2. Here the particles in the tree are represented by the class `Decay::Node` (discussed in section 3.1), decay is represented by the class `Decay::Tree` (discussed in section 3.3) and the arrow in

[1] A decay descriptor is a string representation of the decay e.g. `"B0 -> K- K+"`. This descriptor can be marked, e.g. `"B0 -> ^K- K+"` to select `"K-"`, or it can be enclosed in `"[]CC"` e.g. `"[B0 -> K- K+]CC"` to include charge-conjugated particles.

[2] A regular expressions is a pattern used to match character combinations in a string.

[3] Dedicated parsers e.g. `boost::spirit::qi` [4] are planned to be explored for a future revision.

```
#define fields
#Key: Branch prefix name in the ROOT file
#Value: Decay descriptor
fields = {"B0": "[B0 -> K+ K-]CC",
"Kp": "[B0 -> ^K+ K-]CC",
"Km": "[B0 -> K+ ^K-]CC"}
#add variables to fields
#Key: Branch name defined previously
#Value: List of functors returning kinematics information of "B0"
   particle
variables = {"B0": Kinematics()}
#define instance of FunTuple
#An algorithm to make ROOT files
fntuple = Funtuple(...,fields, variables,...)
```

Figure 1: An instance of `FunTuple` that utilizes decay finder to select and store relevant information associated to reconstructed particles in the decay of $B^0 \to K^- K^+$.

the descriptor forms an instance of the `Decay::Arrow` class (discussed briefly in section 3.2). The syntax that user are required to follow for the correct parsing of the decay descriptor is discussed in section 3.4. In the same section, differences of syntax with respect to previous `LoKi` based decay finders is also highlighted. A set of tests have also been put in place for the decay descriptor parsing and is discussed in section 3.5.

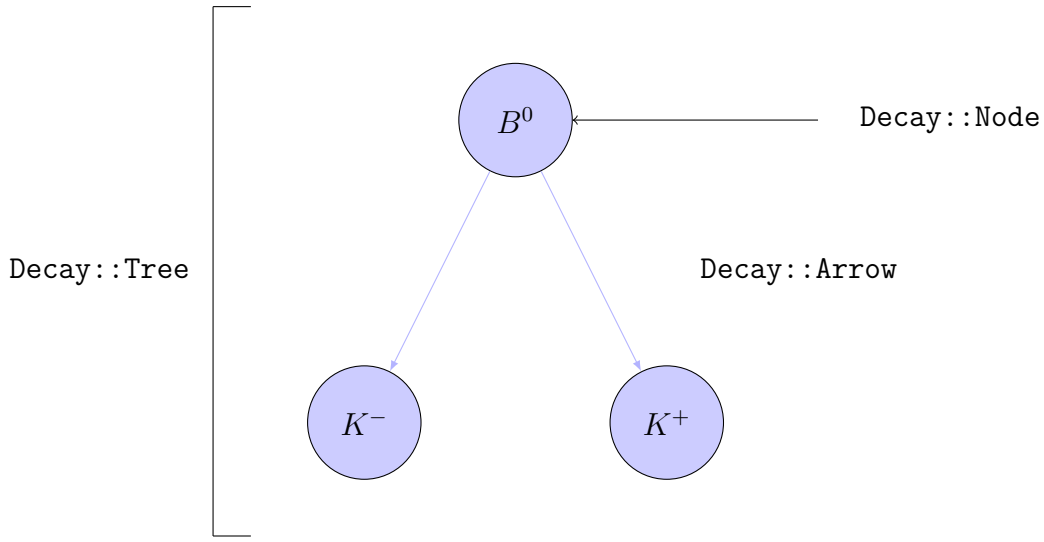


Figure 2: Structure for the parsed decay descriptor `"B0 -> K- K+"`. The decay is represented by the class `Decay::Tree` with particles as instance of class `Decay::Node` and the arrow of the descriptor as the instance of the class `Decay::Arrow`.

3.1 Node class

The most basic element of the tree are particles which are represented by the `Decay::Node` class, with three private member variables as depicted in Fig. 3. The member variables `m_Pid` holds particle identification number [5], the flag `m_isMarked` is set to true if the particle in the descriptor has been marked with the caret symbol e.g. `"^K+"` indicating that the particle needs to be selected by the decay finder after the matching, and the flag `m_hasCC` is set to true if the particle in the descriptor is enclosed in `"[]CC"` indicating that the matching with the reconstructed particle must include both particle and its charge-conjugate.

```
    private:
        /// particle identification number
        int m_Pid;
        /// check if node is marked with caret symbol (^)
        bool m_isMarked;
        /// check if node is enclosed in "[]CC"
        bool m_hasCC;
```

Figure 3: Member variables of the `Decay::Node` class.

3.2 Arrow class

In the decay descriptor, users can specify different arrows which represent different tasks for the decay finder. For reconstructed particles, only two arrow types are valid: single (`"->"`) and long single (`"-->"`) arrows. The single arrow denotes an exact one-to-one matching of the decay between reconstructed particle and the decay descriptor. The long single arrow denotes that the matching with the reconstructed particle must ignore all intermediate resonance states. *The decay finder in the current state only supports matching with single arrow types and a future revision is planned to support the long single arrow type too.* As depicted in Fig. 4, `Arrow` is a `struct` that holds an `enum` of the arrow types.

```
   enum struct Arrow {
       Unknown = 0,
       Single,       /// single arrow              "->"
       LongSingle,   /// long single arrow         "-->"
   };
```

Figure 4: `Arrow` class that holds an `enum` of the arrow types.

There exists other types of arrows (see Ref. [6]) that are only valid when matching the decay descriptor with truth particles (i.e. `LHCb::MCParticle`) that do not include reconstructed information. The `Arrow` class makes it possible to extend support to accommodate the arrow types relevant for `LHCb::MCParticle` matching in the future.

3.3 Tree class

The `Tree` class represents a decay and consists of seven member variables. It holds information about the head of the decay through member variable `m_headNode` which is an instance of the `Node` class, `m_Arrow` represents an instance of the arrow class, `m_daughters` represents of vector of daughter trees that sorted by decreasing depth[4], a string representation of the decay in `m_descriptor`, and, similarly to the `Decay::Node` class, information about the marked and CC status of the decay. Additionally, it has an optional member variables, `m_hasMatched`, that holds information about whether or not the decay descriptor has been successfully matched to the reconstructed particle. This flag is especially useful when matching decays that contain identical particles and is described in detail in section 4.2.1.

```
private:
    /// head node
    Node m_headNode;
    /// Arrow type
    Arrow m_Arrow{ Arrow::None };
    /// vector of daughters
    detail::vec_dt_t m_daughters;
    /// decay descriptor
    std::string m_descriptor;
    /// has cc flag in decay e.g. [B0 -> K+ K-]CC
    bool m_hasCC{ false };
    /// is marked flag in decay e.g. ^(B0 -> K+ K-) or ^([B0 -> K+ K-]CC)
    bool m_isMarked{ false };
    /// decay tree has been matched
    std::optional<bool> m_hasMatched{ std::nullopt };
```

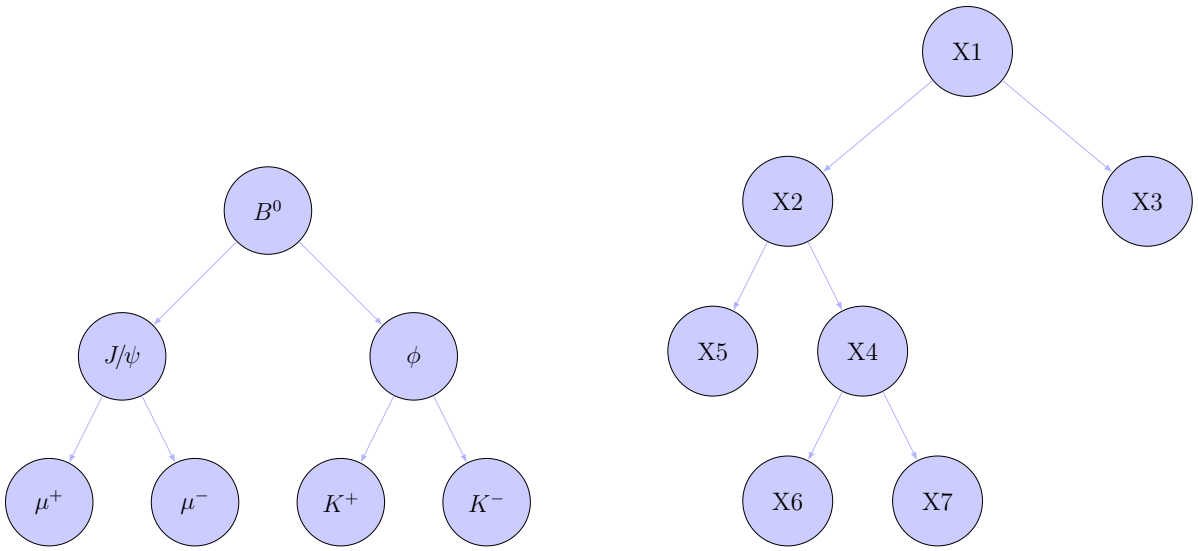
Figure 5: Private member variables of the Tree class. Holds an instance of the `Decay::Node` class as parent, an arrow type, vector of daughters of C++ type `std::vector<Tree>`, the descriptor, flags for the marked, charge conjugate, matched property of the decay.

As result of the above class structure, the `Tree` class can be used to represent decays with N "outer" decays as shown in Fig. 6a, as well as N "inner" decays as shown in Fig. 6b.

3.4 Syntax and differences compared to previous decay finders

When using the decay finder with various operators (e.g. caret symbol (^) or "`[]CC`"), a specific grammar has to be respected. The syntax for node and decay parsing are discussed in the following subsections. In these subsections, we also highlight the syntax differences with respect to LoKi-based decay finder algorithms used in Run I and Run II.

[4]Depth of a tree represents the number of daughters, granddaughters, etc associated to the head of the tree. In other words, depth represents the number of arrows present in a given tree.



(a) Outer Decay (b) Inner Decay

Figure 6: Schematic diagram for the trees `"B0 -> (Jpsi -> mu+ mu-) (Phi -> K+ K-)"` (6a) and `"X1 -> (X2 -> X4 -> X6 X7) X5) X3"` (6b) illustrating examples of outer and inner decays respectively.

3.4.1 Syntax for `Node` parsing

To mark individual nodes e.g. a K^+ meson node, the input string should read `"^K+"` and to select its charge conjugate the syntax is `"[K+]CC"`. To do both, the syntax should be `"^[K+]CC"` where the caret symbol is in front of the square bracket. Parsing nodes with the opposite syntax, i.e. `"[^K+]CC"`, is not supported and will flag an error.

Syntax differences compared to previous finders: The node parsing syntax is different to the previous LoKi-based decay finder used in Run I and Run II. In the LoKi based finder, the the charge conjugated nodes were selected using lower-case `"[...]cc"` directive (e.g. `"B0 -> [K+]cc K-`")[5], where as the decays were selected using upper-case `"[...]CC"` directive (e.g. `"[B0 -> K+ K-]CC"`). This can get confusing, particularly when requesting charge-conjugated decays in both nodes and trees (e.g. `"[B0 -> [K+]cc K-]CC"`). To avoid any confusions, the current decay finder only supports upper case directives, `"[...]CC"`, for both nodes and trees.

3.4.2 Syntax for `Tree` parsing

The marking and requesting of charge conjugated decays, e.g. for $B^0 \to K^+K^-$, should be written as `"^(B0 -> K+ pi-)"` and `"[B0 -> K+ pi-]CC"` respectively. When doing both, the correct syntax should be `"^([B0 -> K+ pi-]CC)"`, where, similarly to the grammar for nodes, the caret comes before the `"[...]CC"` directive with additional normal brackets enclosing the decay: `"^(...)"`.

For complex decays, like $D^0 \to K^0_S(\to \pi^+\pi^-)\pi^+\pi^-$, sub-decays can be marked both in the node, i.e. `"D0 -> (^KS0 -> pi+ pi-) pi+ pi-"` and in the decay, i.e.

[5]Although decays such as `"B0 -> K- K-`" do not conserve charge, such decay candidates are still built and the samples containing such candidates are "same-sign" samples. These samples are used as proxies to study "combinatorial" background originating from random combination of tracks in the event.

"D0 -> ^(KS0 -> pi+ pi-) pi+ pi-", where the two markings have equivalent meaning when matching against particles. This offers more flexibility when specifying decay descriptors.

Syntax differences compared to previous finders: To select a sub-decay the previous tool did not allow for marking the head node of sub-decays (i.e. "D0 -> (^KS0 -> pi+ pi-) pi+ pi-" was not allowed), whereas the current decay finder, as mentioned above allows for marking both the decay and the head-node for added flexibility.

Further more, it is now also possible to mark more than one particle in a decay descriptor, e.g. specify "D0 -> (^KS0 -> ^pi+ pi-) ^pi+ pi-", to select them all at once.

Finally, difference exists in the ordering of the normal and square brackets. In the current decay finder, normal brackets should always be outside the square ones, i.e. "^([B0 -> K+ pi-)]CC", whereas the previous finder supported the opposite ordering, i.e. "[^(B0 -> K+ pi-)]CC". This is due to differences in the parsing routine. In the next update of the current decay finder, we plan to also support the opposite ordering to allow for a more flexible parsing of decay descriptor.

3.5 Tests for parsing of decay descriptors

Parsing of decay descriptors is tested using the python package `pytest` [7], where Fig. 7 shows the list of developed tests and are available in the `Rec` package at Ref. [8].

```
1  test_parsing_exceptions PASSED                    [ 13%]
2  test_parsing_node PASSED                          [ 26%]
3  test_parsing_CC PASSED                            [ 39%]
4  test_parsing_hat PASSED                           [ 52%]
5  test_parsing_CC_and_hat PASSED                    [ 65%]
6  test_parsing_whitespaces PASSED                   [ 78%]
7  test_parsing_complex_decays PASSED                [ 91%]
8  test_parsing_arrows PASSED                        [100%]
```

Figure 7: Terminal output when all parsing tests successfully pass

These tests verify correct exception handling due to wrong particle name or incorrect grammar (line 1), parsing nodes (line2), decay descriptors with CC operators, carets or both (line 3-5), descriptors with unnecessary whitespace (line 6), complex descriptors where daughters decay themselves (line 7) and descriptors with different arrows (line 8). As an example, one of the tests that verify correct exception handling due to wrong particle name or incorrect grammar (line 1) is discussed in the following subsection.

3.5.1 Testing Parsing Exceptions

Fig. 8 displays an example of a test developed to verify correct exception handling when parsing descriptors. This makes sure that the `Tree` or `Node` class correctly raises errors for incorrect particle name, un-balanced brackets (of any type) and incorrect grammar when applying operators on a decay.

```
1    def test_parsing_exceptions(...):
2    map_wrong_to_correct_descriptors = {
3        #exception raised for wrong particle name
4        "Bs" : "B_s0",
5        #exception raised for wrong name in decay descriptor
6        "Bs -> mu+ mu-" : "B_s0 -> mu+ mu-",
7        #exception raised when brackets (square or parenthesis) not balanced
8        # following raises exception since no balanced square brackets
9        "[[B_s0]CC -> mu+ mu-" : "[[B_s0]CC -> mu+ mu-]CC",
10       #exception raised if hat inside CC
11       "[^B_s0]CC" : "^[B_s0]CC",
12       #exception raised if parenthesis inside CC
13       "[(B_s0 -> mu+ mu-)]CC" : "([B_s0 -> mu+ mu-]CC)",
14       #exception raised if square brackets specified without CC
15       "[B_s0 -> mu+ mu-]" : "B_s0 -> mu+ mu-",
16       #exception raised if hat in front of square brackets without CC
17       "^[B_s0 -> mu+ mu-]" : "B_s0 -> mu+ mu-", # no hat => hat on head node
18       #exception raised if hat in front of square brackets with CC
19       "^[B_s0 -> mu+ mu-]CC" : "^([B_s0 -> mu+ mu-]CC)"
20   }
```

Figure 8: Code snippet from the `test_parsing_exceptions` definition which tests correct exception handling.

4 Decay finder class

Once the instances of `Node` and `Tree` classes have been constructed by parsing the user-specified descriptor of arbitrary complexity, we then continued on to develop the `DecayFinder` interface. A `Gaudi` interface named `IDecayFinder` with virtual function `findDecay` was developed and integrated into the LHCb software, see the code snippet in appendix B. The new class `DecayFinder` inheriting from `IDecayFinder` and `GaudiTool` was also developed. This new class holds the implementation of the member function `findDecay`, which is discussed the following subsection.

4.1 Implementation for `findDecay` method

The method `findDecay` internally calls a private member function `findDecayImpl` which is a templated function. This templated function is responsible for comparing trees with particles of any time and for collecting the marked particles (via the caret symbol (^). As shown in Fig. 9, this method has three main tasks: building possibilities (line 7) that is discussed in subsection 4.1.1, comparing trees with particles (line 12) that is discussed in subsection 4.1.2 and collecting pointers to the reconstructed particle that have been marked in the descriptor (line 14) that is discussed in subsection 4.1.3.

```
1  StatusCode DecayFinder::findDecayImpl( const std::string& descriptor,
2          const IDecayFinder::vec_ptr<PARTICLE>&  input,
3          IDecayFinder::vec_ptr<PARTICLE>&        output ) const {
4      // build the user-defined tree
5      Tree tree( descriptor );
6      // get the possibilities from a complex decay descriptor
7      std::vector<Tree> possibilities = tree.getPossibilities();
8      // loop over reconstructed particles
9      for ( const PARTICLE* particle : input ) {
10         // loop over possibilities
11         for ( const auto& possibility : possibilities ) {
12             if ( possibility == particle ) {
13                 // the match has been made, now get the marked particles
14                 possibility.collectMarkedParticles( particle, output );
15                 break; // move to the next particle
16             } else continue; // no match continue to the next possibility
17         }
18     }
19     return StatusCode::SUCCESS;
20 }
```

Figure 9: Code details of the `findDecayImpl` function. It is tasked to compare particle with the user-specified tree and to fill the `output` vector with the marked particles.

4.1.1 Building possibilities

In line 5, the descriptor is parsed and an instance of the `Tree` class is created. Possibilities are then built (line 7) from the created object. These refer to the possible decay permutations of trees labelled by CC operators. For example, a tree represented by the descriptor `"[B0 -> [K+]CC pi-]CC"` has four possible variations: `"B0 -> K+ pi-"`, `"B0 -> K- pi-"`, `"B~0 -> K- pi+"` and `"B~0 -> K+ pi+"`. The public method `getPossibilities()` from the `Tree` class is used to create and return the possible permutations of the decay descriptors in a vector.

4.1.2 Comparing particles with trees

Once the possibilities are built, the implementation function loops over the particle in the input vector, loops over the possibility (of variable type `Tree`) in the possibility vector and compares a particle with a possibility using the "equal to" operator. The operator internally calls the method `hasMatchWithParticle` which acts as an implementation function of the operator. The function first compares the head of the particle and of the tree by matching their particle identification number. It then checks for daughters in the tree: if the tree has no daughters and the head nodes have previously matched then the function returns true. When the tree has daughters the function verifies that the size of the daughter vector of the two objects match. When equal in size, daughters are compared by calling the `hasMatchWithParticle` function recursively.

4.1.3 Collecting marked particles

When the match between a particle and a tree is positive, the method `collectMarkedParticles` loops over both objects simultaneously and evaluates, at each level, the `m_isMarked` property of the daughters. If the flag is positive, the corresponding reconstructed particle is stored in the output vector. When none of the daughters have been marked the head of the decay is stored instead.

4.2 Testing the decay finder

The finding of particles of the `DecayFinder` class is also tested using `pytest` package. These verify that possibilities are correctly built form the user descriptor (line 1), that the decay finder works for two body decays (line 2), for decays with identical particles (line 3-5) and for decays of different depth (line 6-8). As an example, the test related to the finding of the identical particles is discussed in the following subsection.

```
1  test_decay_possibilities PASSED                                    [ 13%]
2  test_decayfinder_twobody PASSED                                    [ 26%]
3  test_decayfinder_identical_threebody PASSED                        [ 39%]
4  test_decayfinder_identical_fourbody PASSED                         [ 52%]
5  test_decayfinder_identical_headdecay_and_daugterdecay PASSED       [ 65%]
6  test_decayfinder_depth_1 PASSED                                    [ 78%]
7  test_decayfinder_depth_2 PASSED                                    [ 91%]
8  test_decayfinder_depth_3 PASSED                                    [100%]
```

Figure 10: Terminal output when all decay finder tests pass successfully

4.2.1 Testing the decay finder: dealing with identical particles

One must take care when matching decays with identical particles in the final state, such as $D^0 \to \pi^-\pi^+\pi^-\pi^+$, where two identical π^- and two identical π^+ particles are present in the final state. If a user requests the first identical π^-, then we need to guarantee that at every call to the decay finder, only the first identical π^- is returned from the container i.e. we need to negate the randomness of returning one of the two identical particles.

To achieve this, the `hasMatchWithParticle` method, which is used in the decay finding process, has to keep track of which daughter has already been matched. It is by modifying the `m_hasMatched` property of the daughters that the function guarantees a one-to-one match between the reconstructed daughters and the ones in the decay tree. This feature is tested in several tests as shown in Fig. 11, where the tests ensure that the two identical particles returned by the decay finder using two decay descriptors have different internal properties.

```
def test_decayfinder_identical_threebody(tool_DecayFinder_Particles):
  #test finding of identical particles present in four-body decay
  descriptor_1 = "[B- -> K+ K- ^K-]CC"
  descriptor_2 = "[B- -> K+ ^K- K-]CC"
...

def test_decayfinder_identical_fourbody(tool_DecayFinder_Particles):
  #test finding of identical particles present in four-body decay
  descriptor_1 = "[D~0 -> ^pi- pi+ pi- pi+]CC"
  descriptor_2 = "[D~0 -> pi- pi+ ^pi- pi+]CC"
...

def test_decayfinder_identical_headdecay_and_daugterdecay(
    tool_DecayFinder_Particles):
  #test finding of identical particles present at two levels: head decay and
     #daughter decay
  descriptor_1 = "[B+ -> ^phi(1020) (phi(1020) -> gamma ^pi0 pi0) K+]CC"
  descriptor_2 = "[B+ -> phi(1020) ^(phi(1020) -> pi0 gamma ^pi0) K+]CC"
...
```

Figure 11: Code snippet from the `pytest` for the `DecayFinder` class which test for correct match of identical particle in three-body and four-body decays and for identical particles of different depth.

These tests, having passed successfully, guarantee that for descriptors like `"[D 0 -> ^pi- pi+ pi- pi+]CC"`, the finder will always collect the first negative pion in the particle's daughters and collect the second one when using the decay finder with the descriptor `"[D 0 -> pi- pi+ ^pi- pi+]CC"`. In the last definition of Fig. 11, the two ϕ mesons are also distinguishable by the decay finder and the two particles can be safely marked.

The matching of identical particles was not supported in the LoKi-based decay finder. Correct matching was not guaranteed and the use of the `CHILD` functor was favored instead.

5 Future developments

The current decay finder consists of templated methods such that it is agnostic towards the type of the input particle. Currently it is tested with the old reconstruction event model used at LHCb during the Run I and Run II periods. A new reconstruction event model is planned to be introduced for Run 3 data taking next year (2023). The new event model is fast evolving and many aspects of this model are still under active development. The next step for the decay finder is to accommodate this new model and develop relevant tests.

To do this, one needs to convert the functions that retrieve particle identification number and children of the reconstructed particle into argument-dependent lookup (ADL) functions. Then we need to overload the the `findDecay` function to take as input, particles

represented by the new event model. Both of these changes would ensure that the current decay finder would be fully compatible with both the new and the old event models.

We also need to align as much as possible with the old decay finder and implement the feature mentioned at the end of section 3.4.2.

In the longer term, with the extension and support for various `Arrow` structures, one can extend the current decay finder to include simulated particles. Since the simulation model for Run III is not expected to change, the LoKi-based `MCDecayFinder` algorithm will still be employed in the finding of the simulated particles.

6 Summary and conclusions

A new decay finder algorithm has been developed to be used with the reconstructed particles in Run III at the LHCb experiment. This algorithm improves upon the old LoKi-based decay finder algorithm used in the Run I and Run II periods. Particularly, handling of the identical particles has been successfully addressed and there have been improvements with added flexibility in the specification of the decay descriptor.

This new algorithm takes as input the user-specified decay descriptor and parses it to build various `C++` classes: `Node`, `Tree` and `Arrow`. The syntax that needs to be respected for the correct parsing of the decay descriptor and the differences with respect to the LoKi-based decay finder have been highlighted. The detailed implementation of the `DecayFinder` interface has been discussed in this note. Various tests have also been put in place for both parsing of the decay descriptor and finding of the reconstructed particles.

Currently, the new decay finder is tested with the old reconstruction event model used during the Run I and Run II periods. A new reconstruction event model, which is planned to be introduced for Run 3 next year (2023), is under active development. The structure of the new decay finder, which includes templated methods, makes it possible to be agnostic towards the type of the input particle. Therefore, once the developments with the new event model have been stabilised, the natural next step would be to develop tests for the new decay finder with the new event model.

Acknowledgements

We thank the coordinators of Data Processing and Analysis (DPA) project, particularly Dr Patrick Koppenburg and Dr Eduardo Rodrigues for their input and support throughout the project. We also thank Dr Gerard Raven for invaluable discussions during the course of the project. We express our gratitude to the members of Real Time Analysis (RTA) working group, for reviewing and for providing us a platform to present and discuss the work conducted. Finally, we are thankful to the CERN summer studentship program, University of Zurich and the LHCb collaboration for providing us the opportunity to carry out the project.

Appendices

A Parsing Algorithm

The parsing of descriptor is done by the private member function `parseDescriptor` *in the* `Tree` *class. This sta*
descriptors without an arrow i.e. single nodes, simple descriptors (one arrow) and complex ones (more than o

In the first scenario, descriptors with no arrows, the string is parsed by the `Node` class where the private member method `pruneNodeName` strips the descriptor from any operators and stores the particle name using `boost::regex_search`. Here, regex is used to find structures like `"[...]CC"` or `"^..."` and to find the pruned particle name. The function then returns a `struct` holding the particle name and two booleans for the marked and CC flags to fill the properties of the node. For example, for a input `"[B0]CC"`, the structure `"[...]CC"` will be found by the regex expression. The node is then striped from the operator and the structure will return the node name as `"B0"` and a `false` and `true` flag for the marked and CC property respectively. The particle name is then used to get the particle's identity number (PID).

In the second scenario, when there is one arrow in the descriptor, the string is first stripped from any marked and charge conjugation operators in the decay using `boost::regex_search`. The parent string and the daughter string are then separated by the `getParentDaughterString` member function and the head node of the decay is directly built from the parent string through the same process described in scenario one. To construct the vector of daughters, the daughter string is parsed by the method `getDaughtersFromString`, which uses the regex expression in Fig. 12 to find the delimiter between daughters. Once found, the daughter are separated accordingly and passed to the `Tree` constructor. This calls `parseDescriptor` where they are parsed according to the first scenario before being collected into the private member variable `m_daughters`.

In the third scenario, there is more than one arrow in the descriptor meaning that some of the daughter themselves decay. The parent and the daughter string are separated using the same method described in scenario two and the head node is directly built. However, the daughter string is parsed using a different method called `findStringWithinOuterMostBrackets`. This function first finds sub-decays, using the regular expression in Fig. 13, and then single daughters (those that do not have decay products) using the regex expression of scenario two. Daughters are then passed to the `Tree` constructor which calls `parseDescriptor`. The daughters passed are either complex decays, simple or single and are parsed according to one of the three scenarios discussed. This process is repeated until the bottom of the tree is reached and ensures that descriptors of arbitrary length and complexity can be parsed.

```
boost::regex expression(
    "(?<![\\[\\s\\^])" // "(?<!...)" means negative lookbehind
                       // Lookbehind assertions specifies how to match the
                       // next character in the regex i.e "\\s+", and can
                       // be negative (above) or positive ("(?<=...)")
                       // Inside is a character sets i.e. "[...]" which
                       // matches only one of the characters in the square
                       // bracket
                       // Since it is negative, it won't match a whitespace
                       // preceded by a bracket: "\\[", by a space: "\\s"
                       // or by a hat: "\\^"
    "\\s+"            // finds at least one or more whitespaces (greedy)
    "(?![\\s\\]]|CC)" // "(?!...)" means negative lookahead
                       // Lookahead assertions specifies how to match the
                       // previous character in the regex i.e. "\\s+", and
                       // can be negative (above) or positive ("(?=...)")
                       // Inside is a character set followed by an OR
                       // condition: "|". Since it is negative, it won't
                       // match a whitespace followed by a space: "\\s",
                       // by a bracket: "\\]" or by a CC: "CC"
);
```

Figure 12: Regex expression that matches and selects whitespaces conditionally in the function `getDaughtersFromString`. Used to delimit daughters in the daughter-string (scenario two).

B Gaudi Interface for DecayFinder

```
struct GAUDI_API IDecayFinder : extend_interfaces<IAlgTool> {
    /// interface machinery
    DeclareInterfaceID( IDecayFinder, 4, 0 );
    /// find decays for Particles
    virtual StatusCode findDecay(
        const std::string&          decay_descriptor,
        const finder::vec_ptrv1&    input_parts,
        finder::vec_ptrv1&          out_parts ) const = 0;
    ...
};
```

Figure 14: `Gaudi` interface for the `DecayFinder` class.

References

[1] C. M. LHCb Collaboration, *Computing Model of the Upgrade LHCb experiment* , CERN, Geneva, 2018. doi: 10.17181/CERN.Q0P4.57ON.

```
boost::regex expression(
"(" // "(...)" capturing group for
    // "b(?:m|(?R))*e"
    // group matches balanced or nested patterns where "b" is the
    // beginning of the patterns, "e" the end and "m" what can occur
    // in the middle
    // this is recursive as per the "(?R)" specifier

    "(?:\\^\\s*\\(|\\()"  // "b" = "(?:\\^\\s*\\(|\\()"
                          // beginning of the pattern is either a hat "\\^"
                          // followed by any number of whitespaces "\\s*"
                          // and a opening bracket "\\(" OR: ("|"), a
                          // single opening bracket "\\("

       "(?:[^()]+|(?R))*" // "m" = "[^()]+"
                          // character set that matches a character
                          // not in the set one or more consecutive time
                          // The negation is indicated by the "^" inside
                          // the character set
                          // Recursion gets incremented if a character from
                          // the set in encountered
                          // "*" means that pattern will repeat zero or
                          // more consecutive time

    "\\)"  // "e" = "\\)"
           // the pattern ends with a closing bracket "\\)"

")" // end of the capturing group
);
```

Figure 13: Regex expression that matches and selects sub-decays, using regular expression recursion. Used in the function `findStringWithinOuterMostBrackets` (scenario three).

[2] LHCb, *LoKi-based Decay Finders*, `https://twiki.cern.ch/twiki/bin/view/LHCb/FAQ/LoKiNewDecayFinders`, 2022. [Online; accessed 16-Sept-2022].

[3] J. Maddock, *Boost.Regex 7.0.1*, `https://www.boost.org/doc/libs/1_80_0/libs/regex/doc/html/index.html`, 2022. [Online; accessed 16-Sept-2022].

[4] H. K. Joel de Guzman, *Qi - Writing Parsers*, `https://www.boost.org/doc/libs/1_80_0/libs/spirit/doc/html/spirit/qi.html`, 2022. [Online; accessed 16-Sept-2022].

[5] LHCb, *Monte Carlo Particle Numbering Scheme*, `https://pdg.lbl.gov/2007/reviews/montecarlorpp.pdf`, 2006. [Online; accessed 16-Sept-2022].

[6] LHCb, *Grammar in short: Arrows*, `https://twiki.cern.ch/twiki/bin/view/LHCb/FAQ/LoKiNewDecayFinders#Arrows`, 2022. [Online; accessed 16-Sept-2022].

[7] H. Krekel *et al.*, *pytest*, https://docs.pytest.org/en/7.1.x/, 2022.

[8] LHCb, *Tests for parsing of decay descriptors*, https://gitlab.cern.ch/lhcb/Rec/-/blob/9821a22a9cd584a210e59eb32df8816af57864ca/Phys/DaVinciKernel/tests/decays/test_descriptor_parsing.py, 2022. [Online; accessed 16-Sept-2022].