CERN - LHCb Collaboration

New Decay Descriptor Parsing for LHCb

Author:
James Kimmins
CERN Summer Student
University of Warwick

Supervisors:
Dr. Adam Morris
Dr. Chris Burr
LHCb Collaboration

September 11, 2026

CERN, Geneva, Switzerland

Contents

1 Introduction **1**
1.1 The Gaudi Framework 1
1.2 What is a Decay Finder? 2
1.3 Decay Descriptor Syntax 2
1.4 Motivation for New Implementation 3

2 New Decay Descriptor Parsing **4**
2.1 String Parser 4
2.2 Compiler 5
2.3 Configuration 6

3 New Decay Finder **7**
3.1 Key Classes 7
3.2 Decay Finding Implementation 9

4 Conclusion **11**
4.1 Acknowledgements 11

A Complete Decay Descriptor Syntax **12**

B Descriptor Parser Grammar File **16**

1. Introduction

The LHCb Experiment is one of four major experiments operating on the Large Hadron Collider (LHC) at CERN. Originally conceived to be dedicated to studies of CP violation within the b and c physics sectors, it has since expanded to many other areas including hadron spectroscopy, lepton flavour universality, and electroweak physics. The experiment has measured many rare decays of both b and c quark mesons and baryons with branching ratios down to order 10^{-9}. [1, 2]

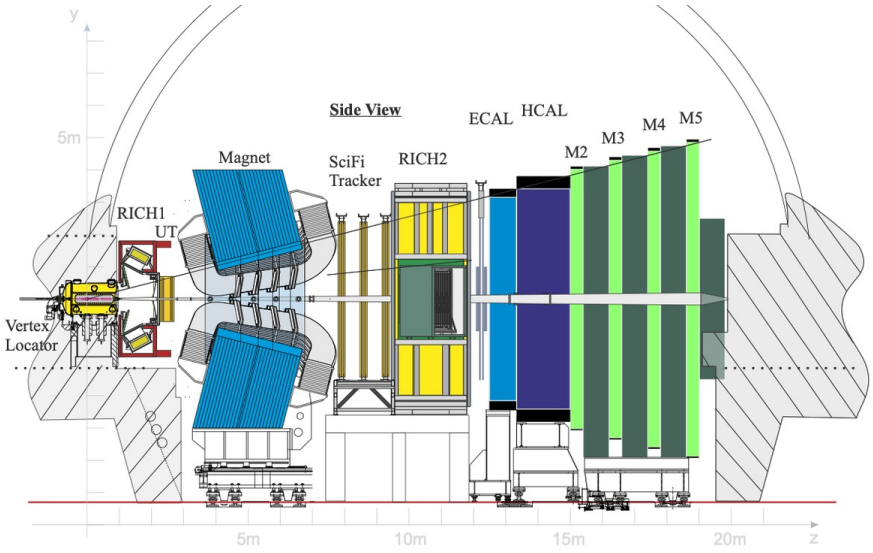

Figure 1.1: [2] A side-view schematic of the LHCb detector.

In order to perform its various experiments, simulations, and analyses, the Collaboration makes use of an extensive software stack. This stack is spread over several distinct repositories some of which are shared with other High Energy Physics (HEP) experiments. The various algorithms and tools within these repositories are tied together using the Gaudi Framework [3].

LHCb's decay finding solution was last significantly altered in 2022 with a new version based in DaVinci [4] introduced to address the challenges posed by the additional luminosity requirements resulting from LHCb's Run 3 upgrades [5]. Before this update, the decay finding implementation was based within the LoKi software package [6].

This report will explain the nature and purpose of a decay finder, discuss the motivation for a new implementation, and describe the design and code decisions made in its creation.

1.1 The Gaudi Framework

Within the Gaudi architecture, software is designed around an event loop and is organised into algorithms (which run on the event loop), services (which run separately from the event loop), and tools (which perform a specific task and are called by algorithms, services, or other tools). The purpose of Gaudi is to link and manage all of these distinct pieces of code collecting them into one cohesive system (see FIG. 1.2). [7]

The ideology of Gaudi is built upon the principle of specialisation. By allowing each developer to focus their work on only the specific tasks and problems that they are best suited to solve, the capabilities of the whole system are maximised.

Algorithms compose the core of the Gaudi architecture. Their principle functionality is reliably to

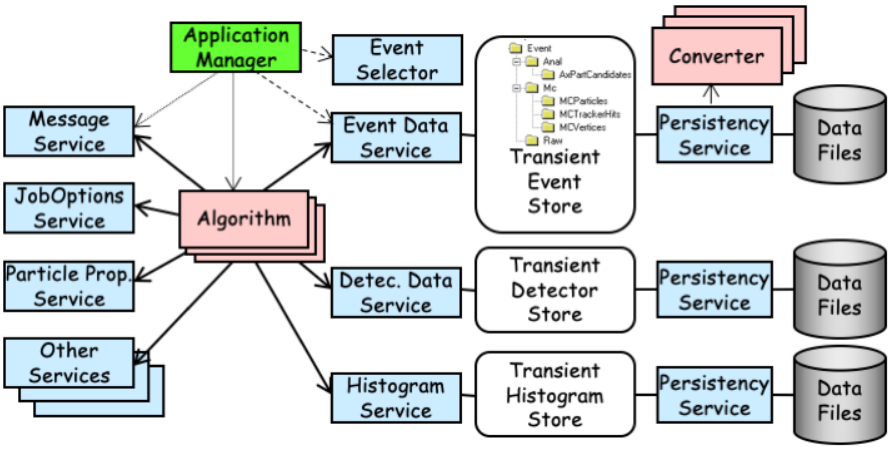

Figure 1.2: [7] An object diagram of the Gaudi architecture.

take in input data, manipulate it in some way, and produce new output data. In order to perform this task, they may make use of services (such as the Particle Property Service to use information about particles) and tools (such as the Decay Finder to identify relevant decays from a container). The independence and interoperability of tools is consequently a priority. [7]

1.2 What is a Decay Finder?

A decay finder is a Gaudi tool which searches through a provided container of particles and returns a list of only the input that matches a user-specified decay descriptor. A decay descriptor is a string representation of one or more particle decays (e.g. `"K0 -> pi+ pi- pi0"`). In a single standard analysis, a decay finder could potentially run this searching algorithm $\mathcal{O}(10^7)$ times and it represents a foundational pillar of the event analysis framework [8].

The decay finder is utilised at various points throughout the LHCb software stack including generator cuts, FunTuples for offline data processing, and selection of LHCb events [9]. It is largely as a consequence of its widespread and varied usage within the Collaboration that a decay finder is usually required to support a wide range of descriptor syntax.

1.3 Decay Descriptor Syntax

Decay descriptors can be incredibly simple resolving only to a single straightforward decay channel however, they can also be highly complex. Various features, specifiers, and types can be utilised to inform the decay finder some of which are unique to specific classes of particle. This section will describe the basic format of a decay descriptor; for a complete description of the available syntax range, see Appendix A.

The foundational syntax of a decay descriptor consists of a head, arrow, and daughters (i.e. `"head -> daughter1 daughter2 etc"`). The head and daughters are ordinarily distinct particles (e.g. `B0`, `pi+`) however, they can also be predicate terms which can match multiple potential particles (e.g. `Meson`, `Charged`). There are additionally derived predicates which are similar in operation but allow the user to specify a particular argument rather than relying solely on pre-defined behaviour.
Basic boolean operators (i.e. `AND`, `OR`, and `NOT`) are supported both on the level of individual terms as well as whole decays and daughters can themselves interact further as a subdecay.
Provided each decay ultimately resolves to one head transforming into several daughters, it is likely valid.

There are additionally multiple available arrow types which allow the user to specify a particular

method for searching through available daughters such as being inclusive to an arbitrary number of additional photons or ignoring intermediate resonances. (The full grammar details are included in Appendix A however, it should be noted that some syntax is reserved for simulated particle searches as they depend on truth information.)

1.4 Motivation for New Implementation

For LHC Runs 1 and 2, LHCb's decay finding implementation was unified and based within the software package LoKi [6]. This finder had full syntax coverage and was well utilised throughout the Collaboration's software stack. However, the upgrades to the LHCb detector made in preparation for Run 3 entailed a significant overhaul to the trigger and data-processing software [10]. The additional requirements of this upgrade (namely a factor thirty increase in the rate of data-collection) motivated the development of a new decay finding implementation which was developed and implemented within DaVinci [4].

The LHCb software stack did not fully move over to this new version however. The new decay finder was significantly more limited than the LoKi-based version it replaced. It only covered a small portion of the available decay descriptor syntax and it was unable to interface with generator particles [11]. Additionally, it took a different approach to its string parsing algorithm by making use of regular expressions. Whilst there was fair rationale for this decision, ultimately, regular expressions are not an appropriate tool for parsing grammar that can be arbitrarily recursive. In the case, a dedicated parsing module would have been significantly preferable.

Because of these limitations, various parts of the LHCb stack continue to use various implementations of the legacy decay finder as opposed to the more-modern version. LHCb's simulation software Gauss uses LoKi for generator-level cuts [12] even when producing Run 3 simulations and other parts of the stack (such as Moore and DaVinci) use ThOr functors for event selection whilst utilising the newer DaVinci-based finder for ntupling [13].

Whilst this setup has allowed for the Collaboration to continue its work effectively, the duplication and isolation of fundamentally equivalent work is unnecessary and can induce developmental confusion. Unifying all of the decay finding implementations would eliminate this inefficiency whilst making future development work significantly easier. Additionally, the decision has been taken to fully extract and remove the LoKi package from the LHCb stack.

These factors combined made a compelling case for continuing development and producing a new decay finder. The principle goals of this work were to:

- Unify all decay finding implementations across the LHCb stack creating a common tool for all use-cases,
- Enable standalone parsing for external packages improving flexibility, and
- Achieve full feature parity with the LoKi-based decay finder whilst supporting as much syntax as is practically achievable in order to improve user experience.

Ultimately, the new implementation was successful in all of these goals. The rest of this report discusses how this was achieved and the specific design choices made.

2. New Decay Descriptor Parsing

Before any decays can be found, the user-provided decay descriptor must first be parsed such that the algorithm extracts all of the necessary information about the desired decay from the string representation.

The LoKi and Run 3 DaVinci decay finders utilised the `Boost::Spirit` and `Boost::Regex` libraries respectively for their decay descriptor parsing. These are both entirely implemented within `C++`. In order to simplify future development as well allow more flexibility with the potential for standalone parsing algorithms, it was decided to decouple the descriptor parsing from the decay finder and implement it separately within Python.

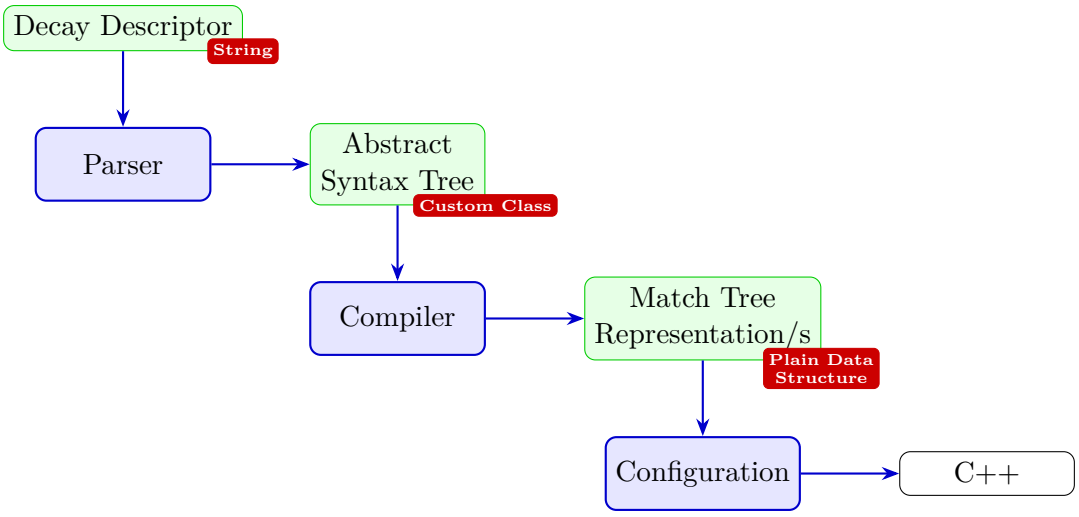

Figure 2.1: Schematic overview of the Python-side decay descriptor parser.

The purpose of the descriptor parser is to take in a user-provided decay descriptor, parse it, and output a plain-data structure which can then be used to configure the decay finder which remains in `C++`. To encourage flexibility and ease of future development, a highly modular approach was taken with the parser being divided into two distinct components: a string parser and a compiler (see FIG. 2.1).

2.1 String Parser

Within the string parser, the actual parsing implementation is delegated to an external Python module named Lark [14]. In order to perform its parsing, Lark requires a grammar file which is provided by the developer. It is the purpose of the grammar file to define the various syntax paths of the potential string input. This if far from a trivial task with descriptor grammar often being highly recursive and complex. Overall, the approach is beneficial, however, as the resulting parser is highly efficient and reliable. The grammar file produced for use by the string parser is discussed in Appendix B.

Once it has parsed the provided string, Lark outputs a Lark tree object. This is then passed through a series of transformer functions (collectively called the transformer) which ultimately result in an Abstract Syntax Tree (AST) containing the full logic and information of the inputted string whilst remaining entirely independent of Lark (see FIG. 2.2).

It is worth noting that the parser has no understanding of the actual string components it is parsing. It does not interface with Gaudi and does not resolve particle or predicate terms. This logic is deferred to the Compiler allowing for considerably flexibility. The string parser does have a complete definition

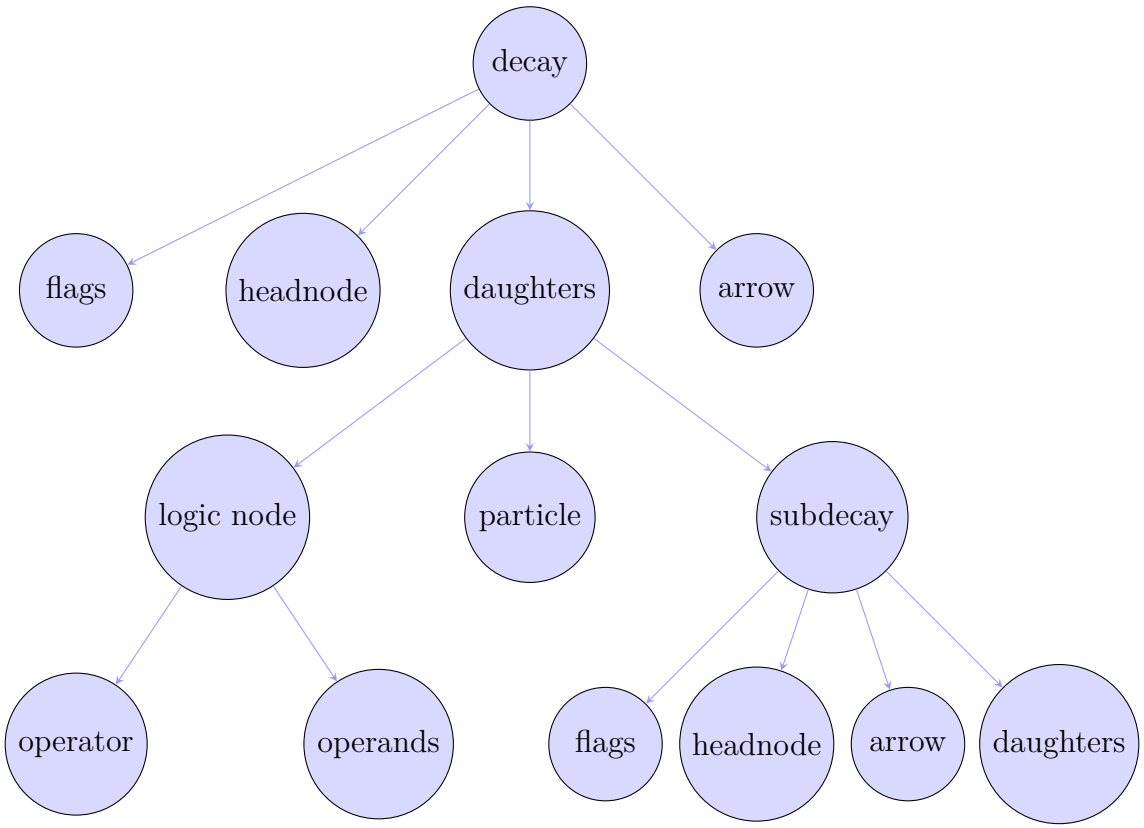

Figure 2.2: An incomplete simplified representation of an example Abstract Syntax Tree (AST).

of the decay descriptor syntax, however, which allows it to immediately throw an error upon receiving an invalid input.

The benefits of this approach are numerous. Firstly, syntax errors are caught early (within configuration). In the previous decay finder implementations, an invalid descriptor would have made it through the configuration ultimately failing with the first execution of the `findDecay` method. Now, the program is able to fail immediately with the configuration failing to reach completion until a valid descriptor is provided.
Secondly, future development work on either the parser or other elements of the decay finding algorithm will be easier. The grammar file is effectively human readable and the Lark-object to AST transformer is a straightforward linear process. The string parser is also entirely independent from any other aspect of the parsing or decay finding system allowing for siloed development work (as opposed to the current situation where a developer is unable to consider the parsing in isolation from the decay finding and vice versa). This could be extended further to allow the parser to more easily be adapted to non-Gaudi software such as DIRAC, RapidSim, DecayLanguage, and offline analysis code.

2.2 Compiler

Once the parser has completed its construction of the Abstract Syntax Tree (AST), this decay representation is passed to the compiler. The purpose of the compiler is to process the AST and convert it into a plain-data structure that, to the fullest extent possible, is a direct copy of the representation needed by the decay finder's configuration. This also has the side-effect of being human-readable and, thus, relatively straightforward to understand. (During development and within this report, this plain-data structure is often referred to as the Match Tree Representation (MTR).)

The compiler performs this task using the Python `singledispatch` mechanism. As each node within the AST is of a known type (inherited from the string parser), the compiler can perform the necessary

logic on each type independently with recursive calls accounting for nested nodes. This allows the compiler to transform ASTs of arbitrary complexity.

Within the compiler's logic, decay terms are resolved by comparing the provided string against LHCb's master list of particles [15]. As of the submission of this report, particles that do not match against any item on this list are assumed to be predicate terms. The compiler then checks if the term satisfies the structure of a derived predicate (containing a closed set of brackets) and, if so, extracts the argument. In future, adding a check against a similar list of valid predicate terms would be beneficial such that incorrect terms can be allowed to fail within the configuration.

The MTR output of the compiler is simply a nested Python `dict`. The importance of this format is that it can be serialised by JSON [16] which is necessary for the configuration of the decay finder in `C++`. (Gaudi already has full support for JSON serialised objects making it the obvious choice for passing the tree representation to the finder.)

2.3 Configuration

Whilst not constituting part of the descriptor parser in the technical sense, it is worth briefly considering the requirements of the algorithm's configuration.

Under the new implementation, the user will create an instance of the `DescriptorCompiler` class providing the chosen decay descriptor as part of the initialisation. It is then possible to extract the MTR from the compiler instance which is passed into the decay finding algorithm after being serialised by JSON.

3. New Decay Finder

Once the descriptor has been parsed and outputted as a JSON [16] object, it is passed into the decay finder using Gaudi [3] via the user's configuration file.

As the decay finder is a Gaudi tool as opposed to a standalone algorithm, the precise nature of how it is called will vary with the user's desired analysis however, the entry point will always involve the use of the constructed JSON object to initialise a `Tree` class instance.

The existing DaVinci-based decay finder was used as the basis for the new development. Whilst many changes were made to adapt the finder to the necessary level of syntax support, the overall structure and methods of the decay finder were entirely sound and were thus left as untouched as was reasonably feasible.

3.1 Key Classes

The two primary classes within the Decay Finder are the `Tree` and `Node` classes (both contained within the `Decay` namespace). These contain the bulk of the decay information and the large majority of the actual decay finding implementation is located within `Tree`.
In addition, there are also two secondary classes: `Arrow` and `Predicate`. These are both enumerated classes containing simple logic for their respective types.

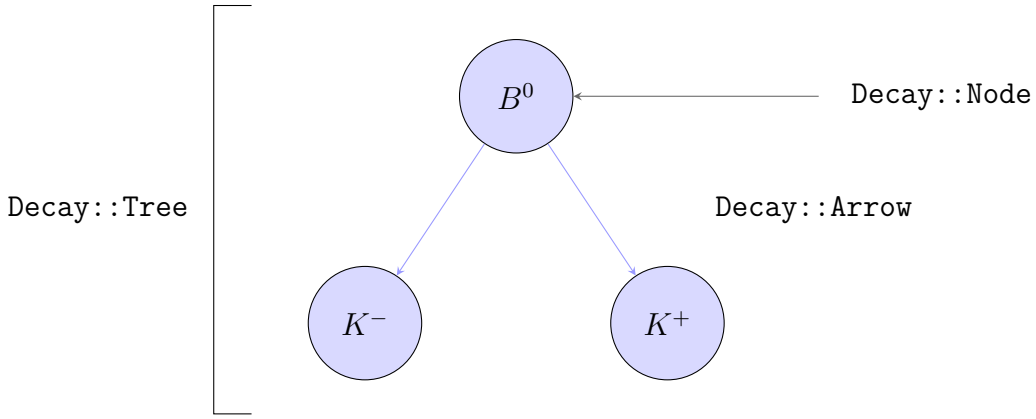

Figure 3.1: [5] A representation of the class structure of the parsed decay descriptor `"B0 -> K- K+"`.

3.1.1 Tree

The core of the `Tree` class is its private member variables (see TAB. 3.1).

As previously stated, the bulk of the decay finding implementation is contained within the `Tree` class. This is discussed separately in Section 3.2.

3.1.2 Node

Similarly to the `Tree` class, the core of `Node` is contained within its member variables (see TAB. 3.2).

The resolution of predicate nodes poses an additional layer of complexity for the `Node` class. Whilst, in principle, a particle can also be considered a predicate as any comparison is just resolved against

Variable Name	Variable Type	Description
m_headNode	Node	A pointer to the node of the decay head
m_arrow	Arrow	The decay's arrow type
m_daughters	std::vector<Tree>	A vector of decay daughters each of which is its own tree
m_hasCC	bool	A flag for if the Tree has CC
m_isMarked	bool	A flag for if the Tree is marked
m_isOptional	bool	A flag for if the Tree is optional
m_isInclusive	bool	A flag for if the Tree is inclusive
m_treeNot	bool	A flag for if the Tree is the subject of a NOT operator
m_hasMatched	bool	A flag for if the Tree has been matched by the decay finder

Table 3.1: An overview of the private member variables of the Tree class.

the node's PDG code, the combination of predicate terms alongside the potential for user-defined arguments and boolean logic make predicates much more complex.

3.1.3 Arrow

Unlike the previous two classes, Arrow is enumerated meaning each instance is initialised as one of a distinct, pre-defined list of types. These types are the different arrow types as set out in Appendix A. Whilst the class also contains some helper functions in order to test against, and convert between arrow types and strings, no decay finding logic is contained within Arrow.

3.1.4 Predicate

Like Arrow, Predicate is an enumerated class however, unlike Arrow, it contains actual finding logic. Primarily, it is composed of a large list of valid predicate terms (see Appendix A) and a compile-time created map between these terms and their corresponding string representations. Each predicate has one primary representation but some have additional aliases (although the behaviours of these are identical).

In addition to the basic conversion methods that translate between string representations and predicate types, there is an additional method: checkPredicate. It is this method that contains the full predicate resolving logic for the decay finder. It primarily makes comparisons using the logic from LHCb::ParticleID however, some predicates require addition interfacing and the use of the Particle Property Service.

Variable Name	Variable Type	Description
m_isPredicate	bool	A flag that describes if the node is a predicate or simple particle
m_pid	int	The PID code of the particle (0 for predicates)
m_predicateTerms	vec_pre_terms	A representation of predicate terms (empty for simple particles)
m_hasCC	bool	A flag for if the Node has CC
m_hasOS	bool	A flag for if the Node has OS
m_hasNOS	bool	A flag for if the Node has NOS
m_isOptional	bool	A flag for if the Node is optional
m_isMarked	bool	A flag for if the Node is marked
m_nodeNot	bool	A flag for if the Node is the subject of a NOT operator

Table 3.2: An overview of the private member variables of the Node class. The type vec_pre_terms is a custom alias declaration for a std::vector containing layered predicate information.

3.2 Decay Finding Implementation

3.2.1 Polymorphism

In order to create a decay finder than can operate on several different particle classes, a polymorphic approach was necessary involving the use of `C++` templates. Two concepts were created, `hasParticleTraits` and `hasTruthTraits`, where the latter is a superset of the former. Each concept requires that its templates have a specific series of defined methods. Thus, by calling the appropriate concept method, it is possible to achieve the same functionality within the decay finder for various particle classes despite the underlying implementations having the potential to be entirely distinct.

This approach required a short code file for each supported class. This simple `struct` translates between a class' own methods and those of the concept. Such a code file was created for `LHCb::Particle` and `LHCb::MCParticle` as well as `HepMC3::GenParticle` and `HepMC3::ConstGenParticlePtr`. Of these classes, only `LHCb::Particle` is solely a member of `hasParticleTraits` with the rest also being members of `hasTruthTraits`. Despite the two `HepMC3` classes being members of `hasTruthTraits`, neither have an implementation for x-type arrows (e.g. `->`, see Appendix A) thus, these arrows may not be used in combination with generator particles.

3.2.2 Initial Tree Construction

When the decay finder is initially called, the first step is always to initialise the `Tree` instance/s which will represent the user-provided decay that container particles will be matched against. The `Tree` class has a constructor from a JSON object thus, this process is usually as simple as a single pass however, there can be additional complications when a single decay descriptor results in multiple valid match tree representations. (This logic is handled by the public `getDecayPossibilities` method.)

The simplest case in which this can happen is by the use of the `CC` flag either on an individual term/s or on the whole decay. Any use of this flag corresponds to an `OR` operator between the named particle and its charge-conjugate. This is simple to resolve.

What is more complicated to account for is the use of logical operators. These can be potentially utilised in combination with each other and can be arbitrarily nested. Fortunately, the organisation of the logic is handled earlier in the stack during the parsing stage. The operators have been flattened and the result is a list of lists of JSON tree representations. The upper layer list represents different alternative decay paths; only one of these need to be matched in order for the whole decay to match (i.e. it is equivalent to the `OR` operator). The internal list is then a list of distinct decay paths connected by `AND` operators.

Theoretically, it is possible to continue this chain of nested lists each alternating between `AND` and `OR` equivalence beyond these initial layers (and indeed, in implementation, the resolution of the `CC` flag does exactly this) however, in practice, decay descriptors can always be flattened into just two layers by the parser.

An identical process occurs for boolean logic on individual decay terms although without the production of entirely new `Tree` class instances.

3.2.3 Decay Matching

Once the tree possibilities have been produced, the decay finding implementation proceeds in two stages applied consecutively to each container particle in turn. This process is subsequently repeated for each identified decay possibility.

Firstly, the finder will iterate through the various layers of the container particle (i.e. headnode, immediate daughters, granddaughters, etc) as far as the decay possibility requires it to check. (In implementation, this is executed via recursive calls.) At each stage, it verifies if each node matches

an equivalent node from the decay possibility. If it does, that node on the decay possibility is marked using the internal `m_hasMatched` flag.

At various stages throughout this process, there are many checks both out of the necessity of the finding logic and to increase efficiency. For instance, before the looping over daughter nodes begins, the total number of container particle daughters is checked against those of the decay possibility and the particle is immediately rejected if the former has more daughters than the latter (unless the possibility is marked as inclusive). (This check does not verify that the number of daughters match exactly as the decay possibility might have optional daughter nodes.)

Once this marking has completed, the second stage begins. The finder loops over the decay possibility and checks if every node has either been marked as having matched or as being optional. If all nodes return true, the finder has found a valid match for the initial decay descriptor and the pointer to the head of the container particle is appended to the output vector.

This approach to decay matching is almost identical in principle to that of the existing DaVinci-based finder. Some changes were made to the implementation in order to account for additional features and a slightly different searching algorithm was introduced for descriptors making use of arrows ignoring intermediate resonances (see Appendix A) however, the overall flow of logic was left unchanged.

4. Conclusion

This report aimed to document the motivation for and subsequent development of a new decay finding solution for the LHCb Collaboration's software stack. The resulting implementation is a genuine improvement over the previous implementations combining the complete decay descriptor syntax coverage of the LoKi-based finder with the improved capacity for data-collection rates of the DaVinci version.

Initial testing of both the decay descriptor parsing and the decay finding implementations has already been completed with all tests showing identical outputs to the LoKi version (which was used as the standard of comparison for testing). Whilst further testing is almost certainly necessary before the new finder can be rolled out to the LHCb userbase, when this does occur, the goal of the unification of various decay finding implementations will have been achieved. In addition, any future development work in this area will be made significantly easier as a result of the modularised nature of the new system.

Whilst this report did not cover all of the precise implementation details, all of the salient characteristics of the new parser and finder have been discussed and explained. For further information, the full software code can be found on the CERN LHCb GitLab [17].

4.1 Acknowledgements

I would like to first and foremost thank my project supervisor Dr. Adam Morris for his support, advice, and guidance over the summer. This project simply would not have been possible without his input.

I would also like to thank my additional supervisor Dr. Chris Burr, the Summer Student Office team, as well as the whole Computing Systems section of the LHCb Collaboration.

Finally, a massive thank you and the best of wishes to my colleagues within the LHCb Summer Student Office and to all those whom I encountered across the CERN Summer Student Programme. You made this summer what it was and it means the world to me.

A. Complete Decay Descriptor Syntax

As previously stated, the available syntax range for decay descriptors is vast and can vary between different input particle types. What follows is an overview of the full available syntax which was intended to be backwards compatible with both the LoKi and Run 3 DaVinci decay finders. This description is primarily taken from the existing LoKi Decay Descriptor Grammar webpage [6].

A.1 Arrows

Arrow	Supported Particles	Description
`->`	RECO, MC, GEN	Search within the direct daughters
`->`	RECO, MC, GEN	Search in decay "sections" (ignoring intermediate resonances)
`=>`	RECO, MC, GEN	Search within the direct daughters allowing arbitrary numbers of additional photons
`==>`	RECO, MC, GEN	Search in decay "sections" (ignoring intermediate resonances) allowing arbitrary numbers of additional photons
`-x>`	MC	Search within the direct daughters including "non-decay" daughters
`-x>`	MC	Search in decay "sections" (ignoring intermediate resonances) including "non-decay" daughters
`=x>`	MC	Search within the direct daughters including "non-decay" daughters allowing arbitrary numbers of addition photons
`==x>`	MC	Search in decay "sections" (ignoring intermediate resonances) including "non-decay" daughters allowing arbitrary numbers of addition photons

Table A.1: The full list of allowed arrow types. The supported particle types RECO, MC, and GEN refer to `LHCb::Particle`, `LHCb::MCParticle`, and `HepMC3::GenParticle` respectively.

The list of available arrow types given in TAB. A.1 are fully supported by the new decay finder.

A.2 Optional & Inclusive Components

An optional decay product is a daughter that can be present within a decay but will not prevent a `Tree` from being marked as matched if it is not present. They are denoted using braces (e.g. `"B0 -> pi+ pi- { pi0 }"`).

An inclusive decay is one that will pass if all of its defined daughters have matched regardless of how many addition daughters there might be. They are denoted using ellipses (e.g. `"B0 -> pi+ pi- ..."`) and this may be placed anywhere within the descriptor daughters.
It should be noted that making a decay inclusive makes the finder ignore the use of optional daughters so the two should not be used simultaneously.

A.3 Mixing

Oscillation flags can be specified for Monte-Carlo and Generator particles using the `[ ]os` and `[ ]nos` specifiers which refer to *oscillated* and *non-oscillated* respectively. These can be applied to individual nodes but not to whole decays.

A.4 Logic Operators

The `AND`, `OR`, and `NOT` operators are fully supported using standard syntax `&&` (or `&`), `||` (or `|`), and `!` (or $\sim$). (Note that $\sim$ can only be used as a `NOT` operator in combination with brackets in order to avoid confusion with particles such as $\sim$`c_L`.)

Logic operators can be applied at both the term and decay level and to both particles and predicates. Additionally, list syntax is supported as an alias for `OR` operators. For example: `"[( B0 -> pi+ pi- ) , ( B+ -> pi+ pi0 ) , ( B- -> l- ... )]"`.

A.5 Marked Components

The marked components of the decay descriptor are indicated using the ^ symbol which can be applied to both individual nodes as well as whole decays. (Marking a decay is equivalent to marking the decay's headnode.) If no decay component is marked, the headnode is considered marked automatically.

A.6 Charge Conjugation

Both nodes and decays can be given the charge conjugation specifier `[ ]CC` (case-insensitive). Such an expression simply means that the corresponding charge conjugated particles are also considered automatically. For instance, `"[B+ ==> pi+ pi+ pi-]CC"` is equivalent to `"(B+ ==> pi+ pi+ pi-) || (B- ==> pi- pi- pi+)"`.

A.7 Predicate Terms

There are a host of available predicate terms that allow a user to match a given decay descriptor to multiple potential decay channels. Most of these are standard however, some are derived (meaning they also take a specified argument). They are primarily resolved simply by making use of `LHCb::ParticleID` however, some are more involved requiring the use of the `PartPropSvc`.

A complete description of standard and derived predicates can be seen at TAB. A.2 and TAB. A.3 respectively.

Predicate	Aliases	Definition
`Any`	`X`	Matches any particle
`Hadron`		Matches any hadron
`Meson`		Matches any meson
`Baryon`		Matches any baryon
`Nucleus`		Matches any nucleus
`Lepton`		Matches any lepton
`Ell`	`l`	Matches any charged lepton
`EllPlus`	`l+`	Matches any positively charged lepton
`EllMinus`	`l-`	Matches any negatively charged lepton
`Neutrino`	`Nu`	Matches any neutral lepton
`Neutral`	`X0`	Matches any neutral particle
`Charged`	`Xq`	Matches any charged particle
`Positive`	`X+`	Matches any positively charged particle
`Negative`	`X-`	Matches any negatively charged particle
`PosID`		Matches any particle with a positive ID
`NegID`		Matches any particle with a negative ID
`Down`	`Xd`	Matches any particle with a d-quark
`Up`	`Xu`	Matches any particle with a u-quark
`Charm`	`Xc`	Matches any particle with a c-quark
`Strange`	`Xs`	Matches any particle with an s-quark
`Beauty`	`Bottom`, `Xb`	Matches any particle with a b-quark
`Truth`	`Top`, `Xt`	Matches any particle with a t-quark
`Scalar`		Matches any particle with $j = 0$
`Spinor`		Matches any particle with $j = 1/2$
`Vector`		Matches any particle with $j = 1$
`ThreeHalf`		Matches any particle with $j = 3/2$
`Tensor`		Matches any particle with $j = 2$
`FiveHalf`		Matches any particle with $j = 5/2$
`ShortLived`		Matches any particle with nominal proper lifetime (in $c\tau$ units) less than 0.1μm
`LongLived`		Matches any particle with nominal proper lifetime (in $c\tau$ units) greater than 0.1μm
`Stable`		Matches any particle with nominal proper lifetime (in $c\tau$ units) greater than 1m
`StableCharged`	`Trackable`	Matches any charged particle with nominal proper lifetime (in $c\tau$ units) greater than 1m

Table A.2: The full list of non-derived predicate terms. An alias simply resolves to the primary term.

Predicate	Argument Type	Definition
`ShortLived_ ( x )`	`float`	Matches any particle with nominal proper lifetime (in $c\tau$ units) less than the specified threshold
`LongLived_ ( x )`	`float`	Matches any particle with nominal proper lifetime (in $c\tau$ units) greater than the specified threshold
`Light ( x )`	`float`	Matches any particle with nominal mass less than the specified threshold
`Heavy ( x )`	`float`	Matches any particle with nominal mass greater than the specified threshold
`JSpin ( x )`	`positive int`	Matches any particle spin j such that $2j+1$ equals the specified value
`LSpin ( x )`	`positive int`	Matches any particle spin l such that $2l+1$ equals the specified value
`SSpin ( x )`	`positive int`	Matches any particle spin s such that $2s+1$ equals the specified value
`HasQuark ( x )`	`string`	Matches any particle with the specified quark

Table A.3: The full list of derived predicate terms. The `x` is replaced with the user's chosen argument.

B. Descriptor Parser Grammar File

As part of the decay descriptor parser, the Lark module [14] was utilised. The full creation of the parser is delegated to Lark however, the developer does still have to define an appropriate grammar file the full contents of which is shown in LST. B.1.

```
%import common.WS_INLINE
%ignore WS_INLINE

?start: "(" start ")"
      | decay_node
      | standard_decay | cc_decay
      | descriptor_chain | descriptor_list

descriptor_chain: start (LOGIC_OP start)+
descriptor_list: (start ("," start)+) | ("[" start ("," start)+ "]")

standard_decay: decay | (HAT "(" decay ")")
cc_decay: ("[" decay "]" _CC) | (HAT "(" "[" decay "]" _CC ")")
?sub_decay: ([HAT] "(" decay ")") | cc_decay | ("(" cc_decay ")")

decay: decay_node ARROW decay_daughter+

decay_daughter: decay_node -> daughter
              | ([(NOT_OP_MARK | NOT_OP_TILDE)] sub_decay) -> subdecay_daughter
              | ([(NOT_OP_MARK | NOT_OP_TILDE)] _LBRACE (decay_node | sub_decay)
                  _RBRACE) -> optional_daughter

decay_node: decay_subnode
          | ([HAT] "(" decay_subnode (LOGIC_OP decay_subnode)* ")")
          | ([HAT] "[" decay_subnode (LOGIC_OP decay_subnode)* "]" TAG)

decay_subnode: ([NOT_OP_MARK] decay_term) | ( [(NOT_OP_MARK | NOT_OP_TILDE)] "("
    decay_term ")" )
decay_term: [HAT] PARTICLE


// Terminals
NOT_OP_MARK: "!"
NOT_OP_TILDE: "~"
LOGIC_OP: "&&" | "&" | "||" | "|"

_LBRACE: "{"
_RBRACE: "}"

HAT: "^"
_CC: "cc"i
TAG: "cc"i | "os"i | "nos"i
ARROW: "->" | "-->" | "=>" | "==>" | "-x>" | "--x>" | "=x>" | "==x>"
PARTICLE: /~?[a-z0-9.][a-z0-9\/+*_.'~-]*(\(\s*[a-z0-9\/+*_.'~-]+\s*\))*[a-z0-9\/+*_
    .'~-]*/i
```

Listing B.1: The Lark grammar file (`DescriptorGrammar.lark`) used by the decay descriptor parser.

It is important to understand that the purpose of the initial Lark parser is not to produce the most human-readable form of the initial decay descriptor. On the contrary, the resulting Lark object is never outputted anywhere a user might be able to view it. The principle purpose of the Lark parser is to produce the form that is the simplest for the following transformer to convert into an Abstract Syntax Tree representation.

B.1 Syntax

Before explaining the choices made specific to decay descriptor parsing, first the syntax of Lark grammar must be explained.

`%` denotes commands to the parser constructor. Specifically, the first two lines of the grammar file instruct the parser to ignore inline whitespaces. This does not mean that this is equivalent to if no whitespaces were entered by the user initially as space separated items are still lexed distinctly. This is effectively just a method to clean up the grammar file as, without this inclusion, each rule would have to continually make allowances for whitespaces.

Rules and terminals compose the bulk of the grammar file and they are defined by a string followed by a colon (e.g. `decay:`). A lowercase string denotes a rule whilst an uppercase string denotes a terminal. Rules may break down into other rules or terminals (called "grammar paths") and terminals may only be defined as string literals or regular expressions.

A `?` before a rule and a `_` before a terminal mean almost the same thing. Lark will consider this definition a valid grammar path but will not include it in the returned Lark object (matched sub-rules are still included). This is tool for cleaning up the result and to make the Lark object to Abstract Syntax Tree (AST) transformation easier. The only difference between these two symbols is that rules marked in this way may still be included in the Lark object if they return multiple terms. (For example, `?rule: term1 term2` would always still be returned.)

The `|` symbol has its standard meaning as the `OR` boolean operator; the `*` and `+` symbols mean "zero or more" and "one or more" respectively; and optional terms are denoted using `[ ]`.
Some rules have alias definitions denoted by `->`. This is just a shorthand and is again intended to make the following AST transformation simpler. Finally, the addition of an `i` at the end of a terminal makes the terminal string/expression case-insensitive.

B.2 Grammar Rules

The grammar file uses the default Lark entry rule `start` which is where the grammar begins although, it should be noted that the parser constructor builds the object from the bottom-up rather than the top-down (i.e. starting at the terminals and ending at `start`). The rule can recursively call itself to remove redundant brackets but otherwise splits into the tree main grammar paths for descriptors: a single node, a single decay, and multiple descriptors. The single node skips to near the end of the grammar file whilst the multiple descriptors funnels back into the `start` rule leaving a single decay as the main grammar path.

With a rule to catch `CC`-flagged decays, the grammar breaks into the headnode, the arrow, and the daughters (which can be sub-decays). The daughters account for the possible inclusion of `NOT` operators and the potential "hats" (`^`) marking components are accounted for throughout. A daughter may also be optional at this stage.

Each daughter may be a `decay_node` which might be a simple particle or a series of predicate terms connected by boolean operators (any of which might also be the subject of a `NOT` operator). These break down into instances of `decay_term` which finally result exclusively in terminals.

B.3 Grammar Terminals

The large majority of terminals are self-explanatory. (The two `NOT` operator symbols `!` and $\sim$ must be treated separately as the latter cannot be used except in front of brackets.) The exception to this simplicity is the `PARTICLE` terminal.

The `PARTICLE` terminal is a regular expression unlike all of the other terminals as it would be im-

possible to know at this stage whether a given string exactly matches a known particle or predicate (without probing the Particle Property Service which is not desired within the parser). However, known restrictions require some matching criteria which is implemented by the regular expression. An explanation of the expression is included within the file itself and is also shown in LST. B.2 for convenience.

```
// ~?[a-z0-9.] ensures that the first character is alphanumeric (or '.') or it is a
    '~' immediately followed by an alphanumeric (or a .)
// [a-z0-9\/+*_.'~-]* is the main particle text
// (\(\s*[a-z0-9\/+*_.'~-]+\s*\))* allows for brackets in particles and requires that
     they always come in closed pairs
// i makes the whole expression case-insensitive
```

Listing B.2: The explanatory comments for the `PARTICLE` terminal's regular expression.

Bibliography

[1] I. Belyaev *et al.*, The European Physical Journal H **46** (2021).

[2] LHCb Collaboration, LHCb Detector, `https://lhcb-outreach.web.cern.ch`, [Accessed: 07 September 2026].

[3] G. Barrand *et al.*, Computer Physics Communications **140**, 45 (2001), cHEP2000.

[4] A. Abdelmotteleb *et al.*, Computing and Software for Big Science **9**, https://doi.org/10.1007/s41781-025-00144-5 (2025).

[5] S. Barré, *Decay finder algorithm for reconstructed particles in Run III at the lhcb experiment*, Tech. Rep. (LHCb Collaboration, 2022).

[6] T. Pajero, LoKi-Based Decay Finders, `https://twiki.cern.ch/twiki/bin/view/LHCb/FAQ/LoKiNewDecayFinders` (2022), [Accessed: 27 August 2026].

[7] LHCb and ATLAS Collaborations, Gaudi Framework, `https://gaudi-framework.readthedocs.io` (1998-2019), [Accessed: 09 September 2026].

[8] J. P. Baud *et al.*, Journal of Physics: Conference Series **396**, 032023 (2012).

[9] A. Mathad *et al.*, Computing and Software for Big Science **8**, 10.1007/s41781-024-00116-1 (2024).

[10] LHCb Collaboration, Journal of Instrumentation **19** ("5"), 1, publisher Copyright: © 2024 CERN for the benefit of the LHCb collaboration.

[11] A. Buckley *et al.*, Computer Physics Communications **260**, 107310 (2021).

[12] I. Beiyaev *et al.*, in *IEEE Nuclear Science Symposium & Medical Imaging Conference* (2010) pp. 1155–1161.

[13] A. Mathad *et al.*, Computing and Software for Big Science **8**, https://doi.org/10.1007/s41781-024-00116-1 (2024).

[14] E. Shinan, Lark Documentation, `https://lark-parser.readthedocs.io/en/latest/` (2020), [Accessed: 27 August 2026].

[15] LHCb Collaboration, Particle Table, `https://gitlab.cern.ch/lhcb-conddb/DDDB/-/blob/master/param/ParticleTable.txt` (2021), [Accessed: 03 September 2026].

[16] Python Software Foundation, json - JSON encoder and decoder, `https://docs.python.org/3/library/json.html` (2001), [Accessed: 27 August 2026].

[17] LHCb Collaboration, LHCb Repository, `https://gitlab.cern.ch/lhcb/LHCb` (2026), [Accessed: 07 September 2026].