

A Visual Query Language for HEP Analysis

Vasco Amaral[1], Sven Helmer[2], Guido Moerkotte[3]

Fakultät für Mathematik und Informatik
Universität Mannheim

Abstract—Pheasant is the first proposal for a declarative domain specific visual query language for HEP data analysis. It has been designed based on our experience dealing with Hera-B event data and query patterns. Its main goal is to allow the physicist to describe the decay selection queries by means of visual operators, to be run against the experiments' existing storage bases and analysis frameworks.

Our visual language aims to be a simple-to-use tool with which a user can express complex decay queries in an easy way with reduced programming efforts. Indeed, the user does not have to deal with intricacies like physical storage details.

In our communication we will describe how we determined the visual primitives by looking at the query patterns. We will also describe our language in an informal manner in terms of syntax, semantics, and example queries.

Index Terms—Pheasant, HEP analysis optimization, declarative query languages, domain-specific query languages, visual query languages

I. INTRODUCTION

THERE is an ever growing need to speed up the analysis process in High Energy Physics (HEP). The task to achieve this is made difficult due to the constant rise of storage demand and complexity of the frameworks.

At the moment, the adopted tools do not distinguish between different layers of abstraction. This means, that physicists who want to analyze data have to learn many different details that are totally unrelated to physics. As examples we mention the storage layout of the data, interfaces of many different utility libraries, programming languages following diverse paradigms (procedural, object-oriented, etc.). As a consequence, scientists are distracted from their actual work. Additionally, when changing from one experiment to another, they face a steep learning curve familiarizing themselves with new tools. Often costly reprogramming efforts are necessary to adapt their analysis process to the new environment.

The major goal of our work is to optimize the HEP analysis process. One important aspect of performance is increasing the

¹This work was partly funded by the Portuguese Governmental Foundation of Science and Technology FCT in form of a research scholarship ref. SFRH / BD / 8918 / 2002

Vasco Amaral, amaral@pi3.informatik.uni-mannheim.de, URL <http://pi3.informatik.uni-mannheim.de/~helmer/>
Researcher at LIP - Laboratório de Instrumentação e Física Experimental de Partículas and Collaborator at Hera-B, DESY/Hamburg

²Sven Helmer, helmer@pi3.informatik.uni-mannheim.de, URL <http://pi3.informatik.uni-mannheim.de/~helmer/>

³Guido Moerkotte, moer@pi3.informatik.uni-mannheim.de, URL <http://pi3.informatik.uni-mannheim.de/~moer/>

user productivity. We achieve this by introducing a logical layer between end users and the underlying physical layer, which comprises the currently used standard tools. The logical layer hides implementation details from the end user, which makes it easier for him or her to become acquainted with the analysis process. Owing to this abstraction layer, a user does not have to learn details on how the data is actually stored on disk. Feeling that we could push this concept even further, we added a conceptual layer, in form of a declarative visual query language, on top of the logical layer. In this way a physicist will be able to carry out analysis on familiar terms and is not forced to learn the intrinsics of programming in several different paradigms.

Complex queries in textual representation written in a general purpose language are not easy to grasp, especially for untrained users. Therefore we propose a declarative domain-specific visual query language. Visual languages have the advantage that the structure of the query is easier to comprehend. Our language also allows a user to formulate queries in a more intuitive way, relying on concepts taken from the domain. That, and the declarative character of the language, reduce the error rate significantly.

Here we give a brief description of the visual query language used in our unifying framework Pheasant (PHysicist's EASy ANalysis Tool)[2], [3], [6]. The formal semantics of the language and the lower layers of our framework are described elsewhere.

In Section II. we present the data model and in Section III. we describe the syntax, introducing the symbolic notations and grammar. A conclusion and an outlook wraps up the paper in Section IV.

II. DATA MODEL

We have modeled the semantic model of the HEP analysis data in an ER diagram that is depicted in Figure 1. It consists of three major entities: Run, Event and Particle. The attributes of the entity Run define the parameters of the experiment, e.g. the setup of the detectors, the time span during which data acquisition took place and general quality issues. As events can only exist within a run, they are modeled as a weak entity (dependent on the entity Run). Its attributes describe properties of particles involved in an event. A particle is mainly described by its track. The remainder of the diagram consists of various entities for describing particles. Particles reconstructed by computations, i.e. whose existence has been derived from the data, are called vertices. We also have to be able to store simulation results. These are the Monte Carlo entities.

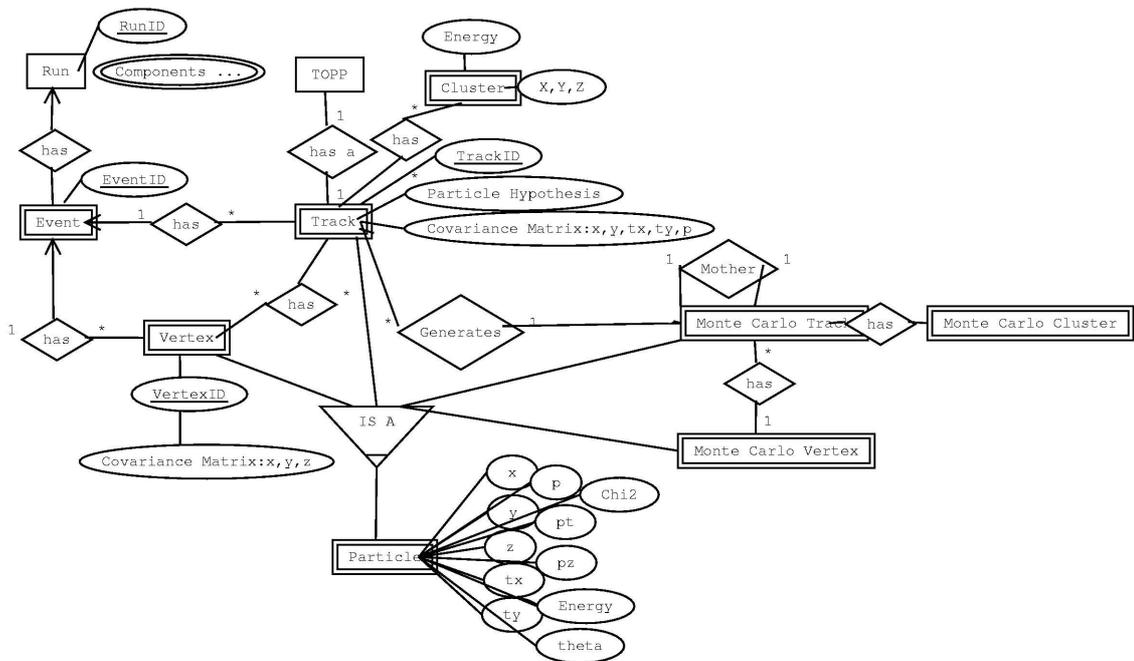


Fig. 1. Patterned ER model of the HEP analysis data, which excludes detector related data

III. DESCRIPTION OF OUR VISUAL LANGUAGE

In designing a consistent VQL, we took into account – based on previous works in the area of VL [4] – that it should satisfy three major user activities: understanding the structure of the data, query formulation and result set visualization. In HEP analysis, the structure of the data does not change much, so we refrained from explicitly showing the whole domain, like in other VLs. We hide the schema to avoid a confusing layout of the interface, as we expect the users to be very familiar with it. This means that each operator contains browsable and editable data, which usually is not visible. Basically, a query has a tree-like structure and its result can be visualized with the help of different histogram layouts, which is the usual approach in physics to handle statistical data.

Furthermore, we did not limit ourselves to retrieving a subset of existing data or selective zooming of it, but also allow the computation of new inferred data (usually seen by the physicists as objects, like new particles) at run time. As we wanted to provide a language where the user has only to worry about what computations are performed and not how they are carried out. This declarative approach leaves the optimization details up to the underlying system.

A. Informal description of the Syntax

1) *Basic Operators:* We are going to introduce the basic building blocks of our language with the help of a running example. Let us start with the thoughts of a physicist doing analysis. Figure 2 shows the schematics of a typical decay chain that can be the center of an investigation. Particle D^+ (on the left hand side) eventually decays into particles Π^+ and Π^- (on

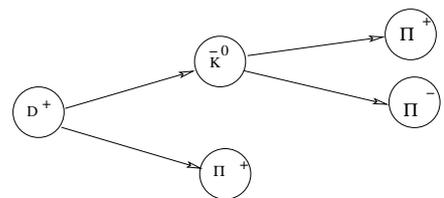


Fig. 2. A typical query

the right hand side). A typical query of medium complexity is to find the D^+ particle with the highest energy level for each event. The particles on the right hand side are the ones whose presence is directly recorded by a detector. This is used as a starting point to search for the original particles, which are too short-lived or too small to be detected.



Fig. 3. Collecting the data in step 1

We are now going to translate these thoughts into a query. First of all we have to decide which run and event data to use. This is the task of the collection operator, which is represented by a small disk symbol (see Figure 3). Associated with this operator is a list of attributes and a list of filter predicates. Assume for a moment that we are only interested in the data from the third run, so in a first step, we have a collection operator that selects this data for us. Collections can be combined using the standard set operators \cap , \cup , and \setminus .

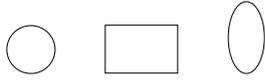


Fig. 4. Selection, Aggregation, Transformation

For the second step we need three more operators: Selection, Aggregation, and Transformation (see Figure 4 for their symbols). Selection works on the data retrieved by a collection operator and selects actual particles detected during these runs and events according to predicates that refer to particles. Aggregation and Transformation operators work on the results of selection operators. An aggregation sums up information on particles per event, i.e. we get one result for each event. Transformation combines the results of two (or more) selections according to predicates that refer to attributes of all participating selections. Usually this results in the construction of a particle higher up in the decay chain. So, the transformation operator creates new particle objects with the data from previous selections.

We now need a way to connect the objects. For this we use a simple line with an arrow that describes the data flow from one operator to another.

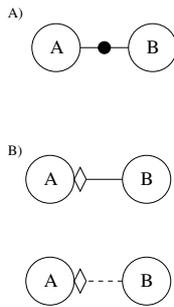


Fig. 5. A)Comparison B)Minimal Distance

Our language supports two more complicated primitives to relate selection objects: the comparison and the minimal distance operators (see Figure 5). Both of them relate two different selection objects and apply a restrictive selection based on a criteria.

The first one, comparison (see Figure 5(A)), basically compares attribute values of one selection object to those of another selection object. In doing so, it filters out particles that do not satisfy the condition of the comparison operator.

For the second operator, minimal distance (see Figure 5(B)), the user can define a threshold value. For each particle x in A we find the particle y in B that has the smallest distance to x . If this distance is smaller than the threshold, both particles, x and y , are kept. All particles in B not hooked up to one in A are discarded. If the distance is larger than the threshold, the behavior of the operator depends on its mode. The first mode, mandatory (represented by a solid line), filters out all particles in A that do not find a match in B. The second mode, non-mandatory (represented by a dashed line), keeps all particles in

A, so just filters out particles in B. Note that this operator is not commutative (as indicated by the position of the diamond).

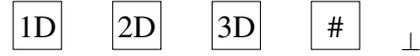


Fig. 6. Result Set Specification:1D,2D,3D,Value Result and operator omission

And last, but not least, we have to describe how to visualize the result of the query in the third step. We provide four different operators for the description of the result (see Figure 6): three operators to create one-, two-, and three-dimensional histograms, and one operator to output numeric values. In case of absence of a result operator (in this case we will represent it textually by \perp), the result will be a set of tuples that match the decay description that will be used to feed some other analysis frameworks, external to our own one.

Figure 7 shows the complete query. The two operators in the upper part of Figure 7 tell the system that we are interested in the data from the third run and want a one-dimensional histogram output of the result. We begin on the right hand side with extracting all Π^+ and Π^- particles from the events of the third run. With the help of a transformation operator (T_1) we reconstruct a \bar{K}^0 particle. Another transformation operator (T_2) helps us find D^+ particles. The condition operator between Π^+ and Π^- guarantees that they have the same mass. A minimal distance operator is used to select the selection object PV, (primary vertex), that is closer to the computed D^+ particle, if none exists, the decay chain is also not selected. Finally an aggregation operation filters out the particles with the maximal energy level for each event.

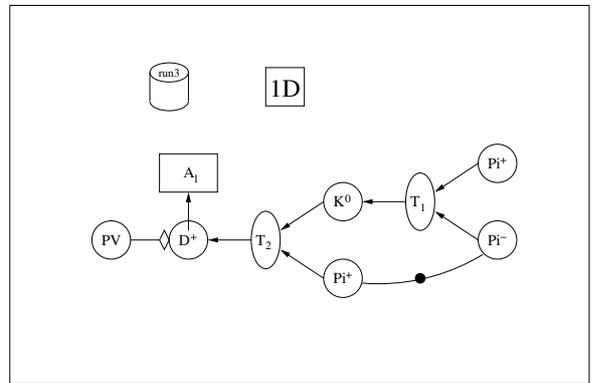


Fig. 7. Example of a Complete Query: the D^+ decay

2) *Implicit graph structures:* There are some non-visible implicit operators in our language that we will specify in this section. These hidden structures deal with the collection objects, result objects, and so-called object references.

The collection objects are implicitly connected to the leaves of the query tree. They “feed” the rest of the query operators, i.e., they can be seen as the source of the data flow.

The result operators implicitly stand at the very end of the query tree. The output of the last operator is sent to the output

operator, i.e., it can be seen as the sink of the data flow.

Object references are used to connect measured data (for instance *Tracks*) to simulated data (for instance *MonteCarlo Tracks*). This reference operator also comes in two flavors: mandatory ($\circ \xrightarrow{mand.ref} \circ$) and non-mandatory ($\circ \xrightarrow{ref} \circ$). The mandatory version eliminates all measured data for which no matching simulated data can be found. The non-mandatory version will keep the measured data regardless of the presence of matching simulated data. The references from measured data to simulated data are usually not visible to the users and can be reached by accessing hidden sub-menus.

B. The grammar

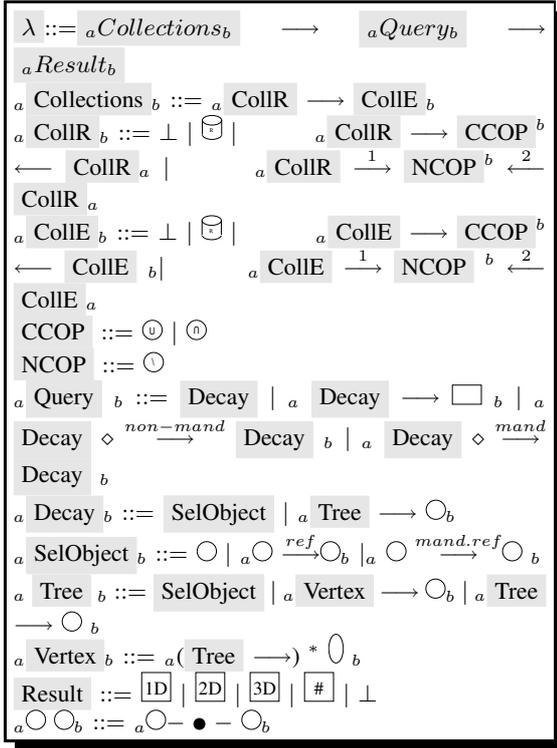


Fig. 8. Grammar of Visual elements of PheasantQL

In order to proceed with the definition of the syntax of our language, we have to describe how symbols may be formed into valid phrases of the language. We do this with the help of a graph grammar (see Figure 8). Grammars are a standard way in computer science to describe formal languages. This grammar is context-sensitive since it allows left and right graphs of a production to have an arbitrary number of nodes and edges.

PheasantQL's grammar comprises four parts $\langle \Sigma, N, P, S \rangle$ where:

- Σ is a finite set of terminal symbols. This is the alphabet of the language, from which we compose the possible words of the language. We decided to use the symbols of the visual language itself as terminals in the grammar, so there would be no problem to recognize the components introduced in the last section.

- N is a finite set of nonterminal symbols (disjoint from Σ). In our description, non-terminals have a grayish background, while for the terminals the regular background is used.
- S is the start symbol λ or null graph.
- P are the production rules stated (summarized for our language in Figure 8). We start with λ and replace non-terminals in the current string with the right hand sides of production rules if they match the left hand side of that rule. Sometimes there are several alternatives on the right hand side divided by $|$ (in this case we have to choose one). We do this step by step until we arrive at a string containing only terminals. This is called a word of the language. The set of all derivable words describes the whole language.

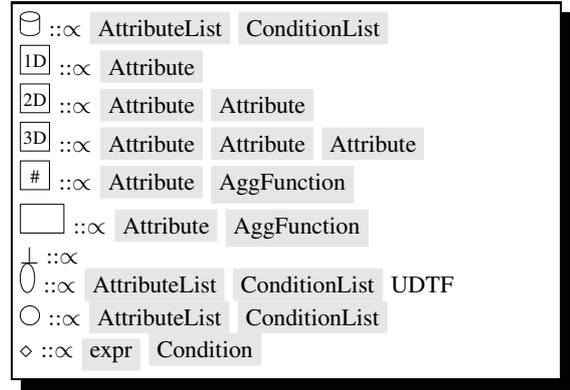


Fig. 9. Terminals definition

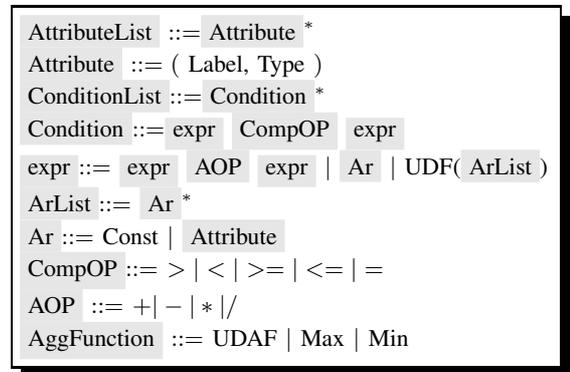


Fig. 10. Grammar of the textual elements of PheasantQL

Let us give some extra explanatory notes. In our production rules we define a and b as connection points to the rest of the graph, and they are used to keep the graph orientation after applying the rule (meaning that the data flow goes from a to b).

Associated with each operator is some additional data, like attribute lists and condition lists. During query construction, this information is hidden most of the time. Therefore, we describe this hidden data associated with each operator with the symbol

:: ∞ (see Figure 9). There are further rules describing the hidden data in Figure 10.

Furthermore, we distinguish between two different collection types: run collections and event collections. When no collections are given in a query, the query considers all available data. If we define a run collection, only data from the selected runs will be considered. We can restrict this further by additionally supplying a description of an event collection. Then only a subset of the events of the run collection will be taken into account. When specifying an event collection without any run collection, we regard the relevant events from all runs. When connecting collections via set operators, we have to differentiate between commutative operators (\cup , \cap) and non-commutative operators (\setminus).

We consider the fact that users want to be able to extend the language with UDFs (user-defined functions). We support UDSFs (user defined scalar functions); UDAFs (user-defined aggregate functions), and UDTFs (user defined table functions). For instance, users can integrate their own aggregation functions (the system currently provides a max- and min-function UDAF) into an aggregation operator. To connect selection objects via a transformation operator, the user can also supply his or her own transformation function (usually a function to reconstruct vertices UDTF). Some expressions and conditions can also be composed by UDFs.

Due to space constraints we are not able to present the formal semantics of the language. We have done this via transformational semantics mapping our operators to operators of the relational algebra (details can be found in [3]).

IV. CONCLUSIONS

In this paper we introduced a declarative domain-specific visual query language for HEP analysis. It hides the complexity of data storage and programming details from the end users.

For future work, we plan to do a thorough evaluation of the language involving feedback from potential users. In this way we can make sure that we satisfy the needs of the users established during the requirements analysis. When the need for changes or extensions arises during this evaluation, we will incorporate these suggestions into our framework Pheasant. We will also try to specify the semantics of the language via monoid comprehension [5] to be able to introduce object-oriented elements without losing meaning during the translation process (causing a so-called impedance mismatch).

REFERENCES

- [1] V. Amaral, G. Moerkotte, S. Helmer, A. Amorim, "Studies for optimization of data analysis queries for HEP using HERA-B commissioning data" Proc. of CHEP'01, Beijing, China, Science Press 3-10:154-155, 2001
- [2] V. Amaral, S. Helmer, G. Moerkotte, "A Domain Specific Visual Query Language for High Energy physicists", OOPSLA 2003, Workshop in Domain Modelling, ACM
- [3] V. Amaral, S. Helmer, G. Moerkotte "Pheasant: A Physicist's Easy Analysis Tool" Technical Report of the University of Mannheim: 8/03, 2003
- [4] T. Catarci, M. Costabile, S. Levialdi, C. Batini, "Visual Query Systems for Databases: A Survey", Journal of Visual Languages and Computing, 1995

- [5] L. Fegaras, D. Maier "Optimizing Object Queries Using an Effective Calculus" ACM Transactions on Database systems, Vol. 25, N 4, December 2000, pp. 457-516
- [6] Pheasant: <http://pi3.informatik.uni-mannheim.de/~amaral/pheasant>
- [7] HERA-B: <http://www-hera-b.desy.de>