Università degli studi di Padova

Dipartimento di Fisica e Astronomia "Galileo Galilei"

Master Thesis in Physics of Data

Integration of FCCee workflows in iLCDirac for large scale Monte Carlo simulations on the WLCG

Supervisor
Dr. Marco Zanetti
Università degli studi di Padova

Co-supervisor
Dr. André Philippe Sailer
CERN

Master Candidate
Lorenzo Valentini

Student ID
2062888

Academic Year
2023-2024

CERN-THESIS-2024-098
//2024

“To my family”

Abstract

The quest for deeper insights into fundamental particle physics, beyond the capabilities of the Large Hadron Collider (LHC), necessitates the construction of more powerful accelerators, and the Future Circular Collider (FCC) is currently the main candidate for this task [1, 2].

Feasibility studies for the FCC involve complex considerations, with extensive large-scale software simulations playing an important role in evaluating the collider's performance and potential scientific outcomes. The magnitude of these simulations places substantial demands on computational resources, thus necessitating the utilization of distributed computing solutions.

The Worldwide LHC Computing Grid (WLCG) is a powerful infrastructure for executing the large-scale simulations required for the FCC. WLCG's global collaboration of several countries and institutions meets the needs of the FCC simulations of accessing, processing, distributing, replicating massive datasets. On top of the WLCG framework, the DIRAC Interware (DIRAC) software provides a unified solution for managing and processing data across diverse computing resources.

In this context, this thesis focuses on the DIRAC extension tailored for the Linear Collider (LC) Community necessities: iLCDirac, and its use to conduct large-scale Monte Carlo productions specific to the FCC. The choice of iLCDirac is driven by its adaptability to the requirements of new high-energy physics experiments and the similarities between the linear collider and FCC software, making it the ideal instrument to accommodate FCC simulation needs.

This thesis aims to examine the development of iLCDirac, with a focus on integrating production workflows necessary for the FCC studies. This includes incorporating new applications for event generation, simulation or fast simulation, and reconstruction processes. Moreover, it involves refining other components of the software infrastructure to meet the requirements for FCC data analysis. Lastly, it presents an assessment of the performance of the initial FCC large-scale productions utilizing the WLCG.

Contents

Listing of figures

Listing of acronyms and abbreviations

API Application Programming Interface

CE Computing Element

CERN European Organization for Nuclear Research

CS Configuration Service

CLIC Compact Linear Collider

CLI Command Line Interface

CVMFS CernVM File System

DB Database

DIRAC DIRAC Interware

DMS Data Management System

EOS EOS Open Storage

FC File Catalog

FCC Future Circular Collider

FTS File Transfer Service

HPC High-Performance Computing

ILC International Linear Collider

JDL Job Description Language

LC Linear Collider

LCG LHC Computing Grid

LHC Large Hadron Collider

LHE Les Houches Event

LFN Logical File Name

OS Operating System

PFN Physical File Name

RMS Request Management System

SE Storage Element

SMS Storage Management System

TS Transformation System

VO Virtual Organization

VOMS Virtual Organization Membership Service

VM Virtual Machine

WLCG Worldwide LHC Computing Grid

WMS Workload Management System

WN Worker Node

1 Introduction

1.1 Thesis outline

This thesis is structured as follows.

In the following 4 sections of chapter 1 there is an overview of:

- the FCC feasibility studies and the need of computing resources,
- the WLCG,
- DIRAC as a grid software,
- iLCDirac as a specific solution for the FCC community.

In chapter 2 there is a more detailed explanation about DIRAC, its principles and architecture design, and specifically the DIRAC systems that are more relevant for understanding the functioning of iLCDirac.

The contributions of this thesis work are presented in chapter 3, in particular:

- a short explanation of the preexisting iLCDirac framework and its functioning,
- an explanation of the needs to be addressed for FCC simulations,
- the introduction of new event generator applications,
- the new workflows for fast simulations and full simulations,
- the creation of a new agent in iLCDirac for treating the production metadata.

Finally, conclusions and future developments are presented in chapter 4.

1.2 FCC feasibility studies

The FCC (figure 1.1 on the next page) project aims to achieve collision energies of 100 TeV, unlocking new opportunities of scientific discoveries: to solve unanswered questions in particle physics left by the Standard Model, there is the need for advanced colliders with higher energies.

The FCC project's implementation relies heavily on exhaustive feasibility studies, evaluating its technical, financial, and environmental feasibility. These studies cover scientific prerequisites, technical obstacles, administrative logistics, and financial projections [1, 2].

An important part of these feasibility studies are sophisticated simulations aimed at assessing detector system performance. Detector studies are necessary for refining the design and efficacy of particle detectors. Through software simulations, it is possible to evaluate the behavior of the detectors, predict detector responses to different particle types and energies, and refine setups to achieve a satisfactory performance. Moreover, these simulations facilitate the investigation of background effects, calibration procedures, and data analysis techniques, ensuring the robustness and reliability of detector systems.

By leveraging a vast amount of very detailed simulations, it is possible to study complex detector designs and physics processes with great fidelity and precision. These kinds of simulations however require vast amounts of computational power and storage space, and traditional computing systems may struggle to handle such needs. For this reason, distributed infrastructures like the WLCG are employed, leveraging interconnected nodes to distribute computational tasks and take advantage of scalable storage solutions [5].

1.3 Worldwide LHC Computing Grid

The WLCG is a significant achievement in international collaboration within particle physics. Originally named the LHC Computing Grid (LCG), it was developed by European Organization for Nuclear Research (CERN) to manage the vast amounts of data produced by experiments at the LHC. [6]

The primary goal of the WLCG was to handle the data output from the LHC. By 2017, the WLCG (figure 1.2 on page 4) had become the world's largest computing grid, boasting millions of CPU cores distributed across its infrastructure. Collaborative partnerships are integral to WLCG's success, with organizations like EGI, OSG, and NeIC playing key roles in enhancing the project's reach and effectiveness. The computing infrastructure of WLCG encompasses multi-petabyte storage systems and computing clusters, enabling efficient data processing and

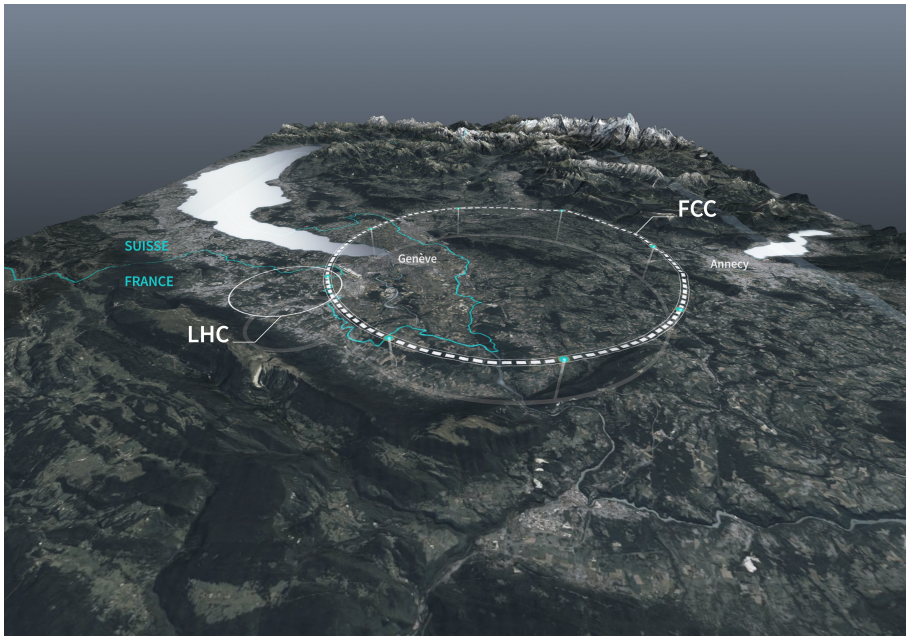

Figure 1.1: Future Circular Collider

analysis [6, 7].

In this manner, WLCG provides several advantages, including data replication, universal accessibility, and backups across multiple sites. Through data replication, copies of files are distributed across the network, ensuring redundancy and mitigating the risk of data loss. Universal accessibility enables researchers worldwide to access data from any location, fostering collaboration and facilitating widespread scientific inquiry. Additionally, backups between different sites further enhance data security and resilience, ensuring continuity of research activities in the event of localized disruptions or failures.

The grid software framework plays a crucial role as an intermediary, facilitating user access to distributed computing resources across the network.

1.4 DIRAC

Effective utilization of distributed computing and storage resources is crucial for the success of contemporary and future High Energy experiments. To address this need, DIRAC has been developed as an interware for constructing and managing distributed computing systems.

Originally designed to facilitate the utilization of distributed computing resources within the LHCb experiment at CERN for data production and analysis, DIRAC enables concurrent

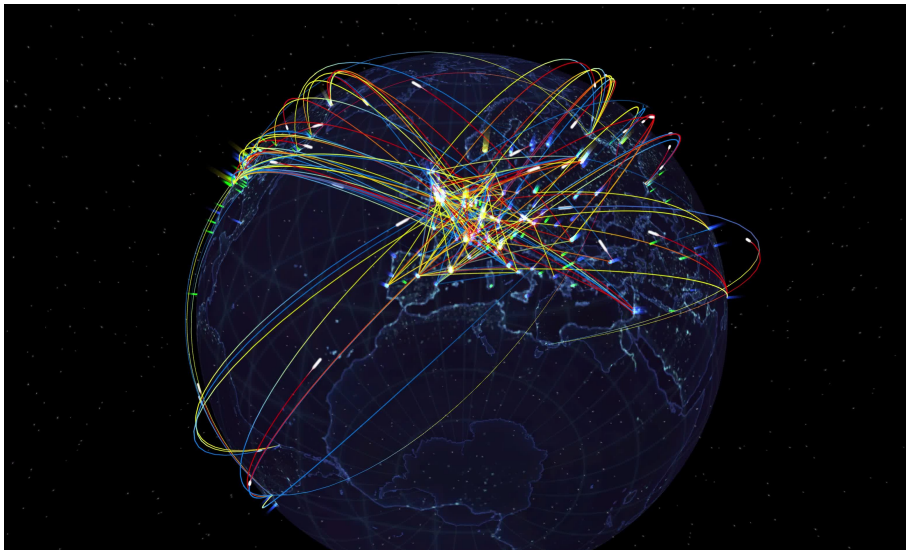

Figure 1.2: Worldwide LHC Computing Grid

use of computing resources spread across numerous sites.

The DIRAC system caters to a wide range of users from different experiments, offering flexibility in interaction based on their tasks, expertise level, and previous experience. Users can engage with the system using command line tools, Python Application Programming Interfaces (APIs), or web portals.

Continuous enhancements have been made to meet the evolving needs of DIRAC's user communities and hosting infrastructures. Examples include the incorporation of industry-standard authorization and authentication infrastructure solutions, management of diverse computing resources (cloud, High-Performance Computing (HPC), GPUs, etc.), handling of high-intensity work and data flows, and advanced monitoring and accounting techniques [8, 3, 9].

1.5 ILCDirac

ILCDirac serves as a comprehensive distributed computing solution tailored for the LC community. Built upon the DIRAC system, it became integral to all the LC detector concept studies [10, 11].

ILCDirac offers a unified interface to distributed resources for the International Linear Collider (ILC) Virtual Organization (VO), streamlining access to various computing assets through a simplified API. It facilitates integration of beam-induced backgrounds while minimizing dis-

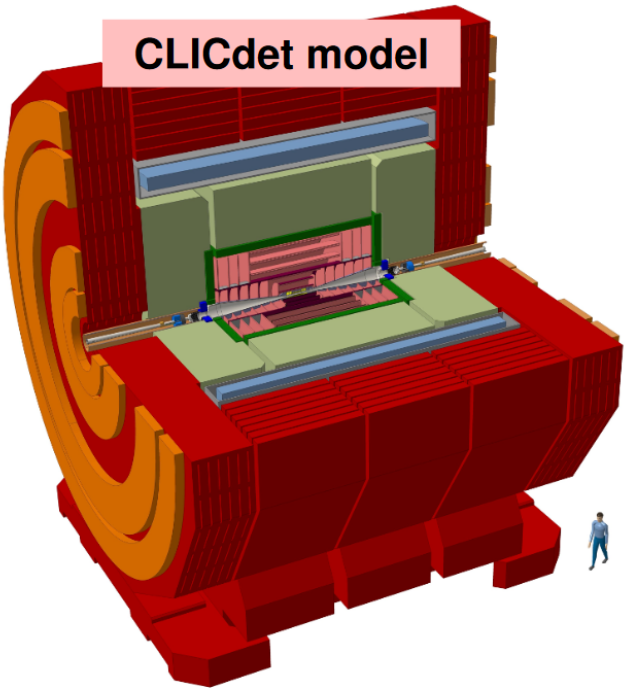

Figure 1.3: CLIC detector

ruption to Storage Elements (SEs) through judicious job scheduling. Its effectiveness has been demonstrated in pivotal projects such as the Compact Linear Collider (CLIC) Conceptual Design Report and the ILC SiD Detailed Baseline Design, leading to its adoption as the official Grid production tool by the ILC community. Furthermore, members of the CALICE collaboration used iLCDirac within their VO, attesting its utility and applicability [10, 11].

Throughout this thesis, iLCDirac has undergone upgrades to cater to FCC studies. ILCDirac now integrates the FCC VO and custom code to manage new FCC-specific workflows.

The development of iLCDirac for FCC studies has focused on refining production workflows essential for comprehensive FCC research. This includes incorporating new applications for event generation, simulation, fast simulation, and reconstruction processes. Additionally, adjustments have been made to various software components to ensure compatibility with the unique data analysis requirements of the FCC.

2 DIRAC

2.1 DIRAC in a nutshell

The DIRAC interware is a complete Grid solution for one, or more than one, community of users that needs to exploit distributed heterogeneous resources. It forms a layer between a community and various compute resources to allow optimized, transparent and reliable usage. The types of resources that DIRAC can handle include:

- Computing Resources, including Grids, Clouds, HPCs and Batch systems,
- Storage Resources,
- Catalog Resources.

So, DIRAC provides a comprehensive solution for the Workload, Data, and Production Management tasks of large scientific communities. A single installation of DIRAC serves as a complete solution for the distributed computing needs of one or multiple collaborations, integrating sites within a Computing Grid like the WLCG or standalone computing clusters under a unified management structure.

The DIRAC system currently serves a wide range of users from different experiments, offering flexibility in interaction based on their tasks, needs, and expertise level, interacting with the system using command line tools, APIs, or web portal.

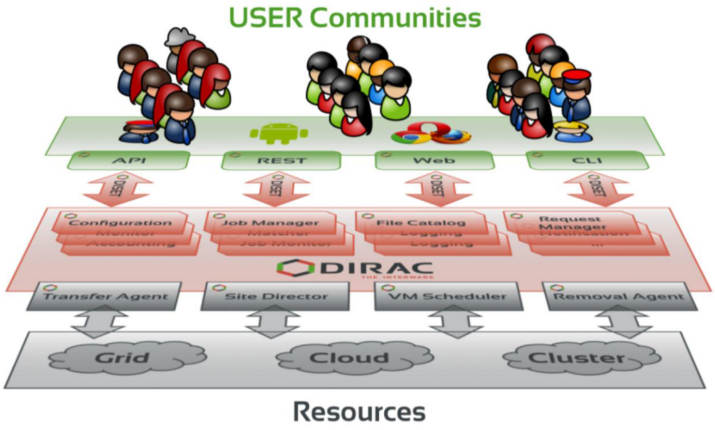

Figure 2.1: Datasets management: DIRAC Transformation, Request Management and Data Management Systems

2.2 DIRAC Design Principles

Here are presented some design patterns that are widely used in the DIRAC system in order to increase its efficiency and resilience to failures.

2.2.1 Redundancy

The inherently volatile nature of distributed computing environments necessitates substantial efforts to enhance service availability. This is particularly crucial in grid environments, where more robust means of monitoring and control are available. Despite diligent efforts to mitigate failures resulting from software, hardware, and human errors, it remains impossible to entirely eliminate all sources of potential disruptions.

Therefore, effective distributed computing systems must be designed to minimize the impact of temporarily unavailable services. It is unacceptable to lose valuable results due to the consumption of significant computing resources, or to have the entire system paralyzed by the failure of a single component. In such scenarios, a distributed computing system should persist in its operation, even if at a reduced capacity. This resilience can be achieved by incorporating redundancy into operations, enabling retries either with an alternative service instance or at a later time when the faulty service has been restored.

In DIRAC, redundancy is implemented through multiple strategies. Firstly, critical information vital for seamless system operation is duplicated across several services, ensuring the availability of at least one copy to fulfil client requests. This redundancy is applied to the DIRAC Configuration Service and the FC, each of which maintains several synchronized mir-

rors alongside the master instance. This redundancy not only safeguards against data loss but also facilitates load balancing, distributing the services load [3].

Secondly, crucial operations that are indispensable for the system's uninterrupted functionality are executed within a failover recovery framework. This framework allows for the automatic retrying of failed operations. All pertinent information required for executing operations is stored in a DB; special agents operating in close proximity to the request DBs persistently attempt to complete stored operations until successful [3].

For data management operations, such as the initial uploading of data files to a grid storage, in the event of a failure, the files are temporarily stored in an alternative storage element. A failover request is then initiated to transfer the data to its final destination once it becomes available. This multifaceted approach to redundancy in DIRAC ensures both data integrity and operational continuity in the face of distributed computing challenges [3].

2.2.2 System state information

Maintaining accurate information about the system state is crucial for successful operations. It is essential to distinguish between slowly changing static information, such as system configuration data, and rapidly changing dynamic information, such as the available capacity of computing resources. Combining static and dynamic data within the same information system should be avoided.

In DIRAC, the static configuration data is accessible to all clients through the Configuration Service (CS). Additionally, this information can be cached on the client side for short periods without risking client misbehavior. Separating static and dynamic information mitigates the risk of compromising static data due to system overloading [3, 4].

Dynamic information is typically sought directly from its source. For instance, the current state of a computing resource is obtained directly from the batch system or the computing element to prevent reliance on outdated data [3, 4].

2.2.3 Requirements to sites

The primary role of sites is to furnish resources for collective use within a grid. Site managers oversee these resources, which are made accessible through middleware services, such as Computing Elements (CEs) and SEs. In some instances, sites, especially those newly joining the grid community, may possess limited expertise in middleware. Simplifying requirements for site operations is crucial to reducing entry barriers for new grid participants, thereby increas-

ing the pool of available resources. DIRAC emphasizes keeping site operation requirements straightforward, minimizing specific demands from various VO [3].

2.3 DIRAC Architecture

DIRAC Interware (DIRAC) comprises various "systems" such as the Workload Management System (WMS), the Data Management System (DMS), the Transformation System (TS), the Request Management System (RMS), and others. Each of these DIRAC systems consists of components, including services and agents, with Databases (DBs) employed to persist the necessary information, as shown in figure 2.3 on page 14.

2.3.1 Services

Services within DIRAC operate as passive components, actively listening to incoming requests incoming connections from a variety of clients, including user interfaces, agents, or running jobs. Typically, each service utilizes a MySQL DB backend to store relevant state information, and responds to the requests by either serving requested information from the backend or inserting requests into the backend. Services often interact with other Services within the same DIRAC System or even across different Systems.

2.3.2 Agents

Agents are lightweight software components designed for easy deployment, that operate as independent processes to carry out one (or occasionally multiple) system functions. They are all constructed within the same framework, which organizes a primary execution loop and provides a standardized approach for deployment, configuration, control, and logging of agent activities. Agents function by executing actions periodically, monitoring service states for changes, and responding by triggering tasks such as job submissions or result retrieval. During each cycle, agents typically interface with services or inspect a DB to identify pending actions, execute the necessary tasks, and report the outcomes. An agent operates in a repetitive loop, executing a designated function every X seconds. It primarily comprises two methods: initialize and execute. Upon startup, the agent executes the initialize method, which typically sets parameters such as the frequency of the execute method. Subsequently, the execute method is triggered. After its completion, the agent waits for a specified amount of time before executing the method again,

or, if the execution took longer, it restarts immediately. This loop continues until the agent is terminated or a specified number of iterations have been completed.

2.3.3 Executors

Executors are another set of dynamic components within DIRAC. They are invoked upon request and are primarily utilized within the DIRAC WMS.

2.3.4 Interfaces

Application Programming Interfaces (APIs) are one of the main communication channels for interacting with the DIRAC system.

The DIRAC functionality is accessible to both system developers and users through various means. Python serves as the primary programming language for DIRAC, with APIs offered in this language. Each service is accompanied by a corresponding client class, which, in the simplest form, acts as a thin layer translating service interface calls.

For developers utilizing the DIRAC system, the functionality is presented as a Python API consolidating essential methods into a minimal number of classes for constructing higher-level applications.

Users of the DIRAC system have access to functionality through a Command Line Interface (CLI), with some DIRAC subsystems featuring specialized shells. These shells offer the added benefit of online help assistance and maintain user session states, proving valuable for tasks like meta catalog browsing.

In addition to command line access, DIRAC provides web interfaces for users and system managers to monitor system behavior and control ongoing tasks.

2.3.5 Databases

DBs keep the persistent state of a System. They are accessed by Services and Agents as a kind of shared memory.

2.4 Data Management System

The DMS and the Storage Management System (SMS) jointly offer essential functionalities to execute and oversee all operations related to the users' data. The DMS handles fundamental

tasks such as uploading a local file to a Storage Element (SE) or removing files from SEs, and registering corresponding replicas in the configured DIRAC File Catalog (FC). Various SE endpoint types and FC types are supported.

DIRAC also facilitates extensive data replications, including the use of the File Transfer Service (FTS), or retrievals of archived data on Tape for subsequent processing. These capabilities are realized through the DIRAC RMS and DIRAC TS.

The entire DIRAC DMS relies on a few key concepts:

- Storage Element (SE): An abstraction of a physical storage endpoint, described in the DIRAC Configuration System, including all the necessary configuration for physical file access.

- Logical File Name (LFN): The name of a file or path, uniquely identifying a file throughout a DIRAC namespace. A file may have one or several replicas.

- Replica: The physical copy of an LFN stored at a SE. The couple (LFN, SE) uniquely identifies a physical copy (also known as Physical File Name (PFN)).

- Catalog: The DMS namespace listing files and their metadata. Catalogs serve as a namespace where files and metadata (size, checksum, SEs where they are stored, etc.) are listed. Multiple catalogs are supported, and operations on one catalog are generally mirrored in others.

The DIRAC DMS exclusively deals with LFNs, and, users or systems, when dealing with files, need only concern themselves with LFNs. In the rare event that specific replicas must be addressed, the couple couple (LFN, SE) should be used. Importantly, there is no leakage of protocols or URLs at any point in the DMS's low-level operations [8].

2.5 Transformation System

In the context of the DIRAC system, a "transformation" refers to a predefined set of actions or processes applied to datasets or files. Transformations encapsulate specific instructions or workflows that dictate how input data should be processed or manipulated to produce desired output results. They may involve various operations such as data replication, simulation job generation, data reprocessing, or dataset management tasks.

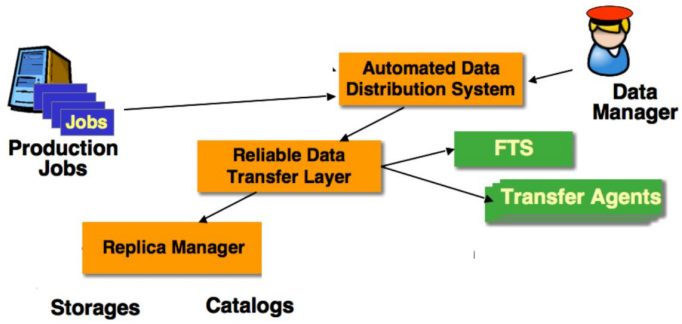

Figure 2.2: DIRAC DMS: handles uploading, removing and replicating data, hiding the underlying structure [3].

For example, a transformation could involve processing raw data files from an experiment using a simulation program to generate simulated events. Another transformation might replicate data files to multiple storage locations for redundancy, or distribute them across different computing resources for parallel processing.

The DIRAC TS serves the purpose of automating routine tasks associated with production activities. To illustrate, the TS can manage tasks such as generating Simulation jobs or Data Re-processing jobs when a 'pre-defined' dataset becomes available. Additionally, it can initiate Data Replication to 'pre-defined' SE destinations as soon as the first replica is registered in the Catalog [8].

The TS functions as a system for queuing similar operation types on specific datasets and directing them to the appropriate systems (the DIRAC WMS for productions and the RMS for dataset management operation types).

In practical terms, the TS plays an important role in DIRAC for overseeing datasets and job productions, featuring a highly flexible design (figure 2.3 on the following page). Key parameters of a Transformation include:

- Type: Simulation, Data Processing, Removal, Replication;
- The option of having (or not) Input Files;
- A Plugin, defining how input datasets are segmented into groups (e.g., by size, destination, metadata, or other coded criteria).

Within the TS, a user can, for instance, generate multiple identical tasks with slight variations

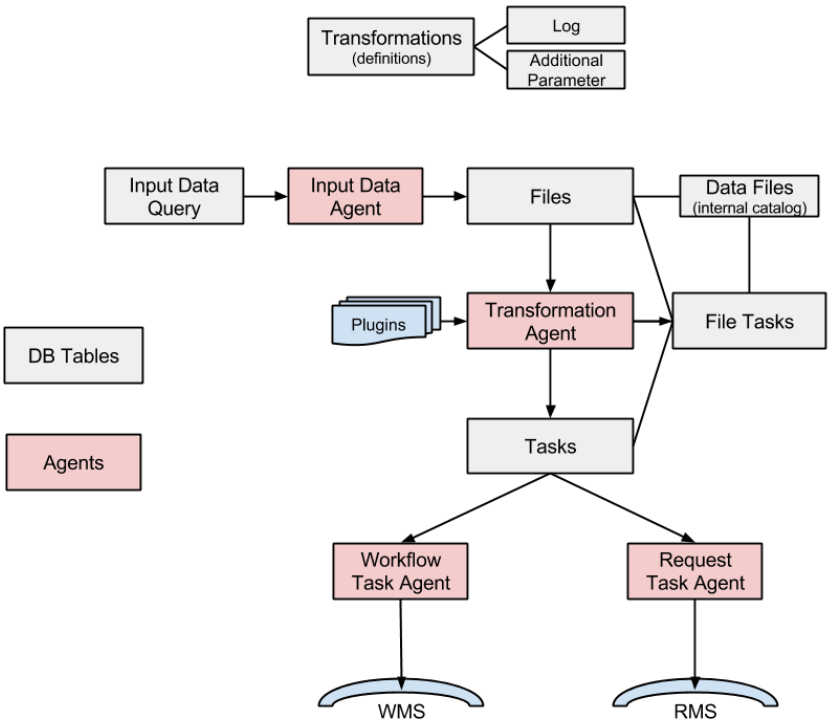

Figure 2.3: DIRAC TS. As a standard DIRAC system, it is composed by Services, DBs, Agents [4].

in parameters (e.g. Input Files list), increase the number of tasks, and associate a single high-level object (the Transformation) with a specific production for comprehensive monitoring. Two simple examples further illustrate its capabilities:

- Dataset management example: Retrieve all the data from 2018 with tag=delphes, ensure one copy on tape and one on disk, distribute them across sites based on free space, and group operations in sets of at most 100 files.

- Jobs production example: Retrieve all data from 2018 with tag=kkmc, process them in a Delphes fast simulation, using only Tier2 sites.

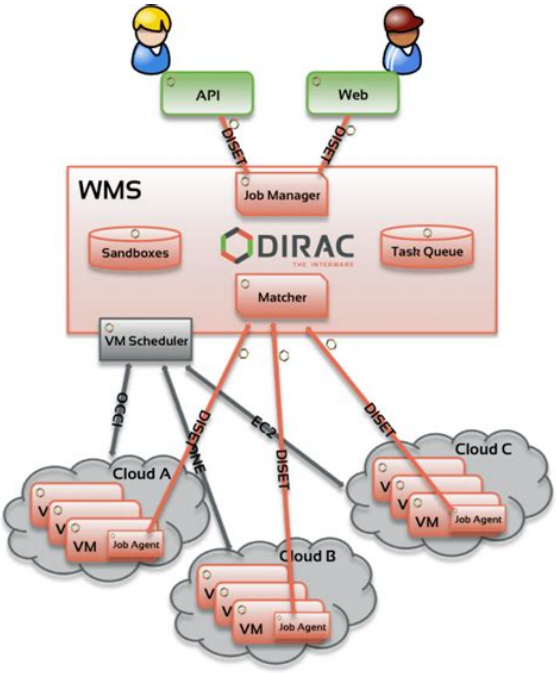

Figure 2.4: DIRAC WMS overseeing the utilization of distributed computing resources.

2.6 Workload Management System

The DIRAC WMS (figure 2.4) oversees the utilization of distributed computing resources, specifically managing jobs and pilot jobs.

The DIRAC Grid model operates with pilots—startup scripts that land at every Worker Node (WN), perform checks, and pull payload jobs from a central queue based on WN capabilities. The pilot job concept has become standard, enhancing user job throughput efficiency and providing a uniform presentation of heterogeneous computing resources to central services. DIRAC supports the integration of non-grid resources, facilitating the exploitation of all available CPU or GPU cycles. The DIRAC pilot serves as the federator, ensuring transparent access to underlying resources [8].

The DIRAC WMS's main role is to leverage different computing resource types, such as Grids, clusters behind batch systems (with support for SSH/GSISSH tunnels), vacuum-type resources, Virtual Machine (VM) schedulers, and High-Performance Computings (HPCs) sites with specific configurations. It manages both single jobs and single pilots, while collections of jobs are typically referred to as "Productions".

The WMS provides high user jobs efficiency, hiding the heterogeneity of the the underlying

computing resources.

Within DIRAC jobs, users specify at least an executable, and maybe some argument, that DIRAC will start on the Worker Node. Jobs are not sent directly to the Computing Elements (CEs), nor to any Computing resource. Instead, their description and requirements are stored in the DIRAC WMS DB (using the Job Description Language (JDL)) and added to a Task Queue of jobs with same or similar requirements. Jobs will start running when their JDL is picked up by a pilot job [4].

Pilot jobs are submitted to computing resources by specialized Pilot Directors. After the start, Pilots check the execution environment and form the resource description (Operating System (OS), capacity, disk space, software, etc.). The resources description is presented to the Matcher service, which chooses the most appropriate user job from the Task Queue. The user job description is delivered to the pilot, which prepares its execution environment and executes the user application.

One evident advantage is that the users' payloads are starting in an already verified environment. The environment checks can be tailored for specific needs of a particular community by customizing the pilot operations.

For the users all the internal WMS/pilots machinery is completely hidden. They see all the DIRAC operated computing resources as single large batch system.

The DIRAC WMS basically follows this workflow:

- Users define and submit jobs. Jobs have requirements. Job descriptions are stored in DIRAC's Job DB.
- A DIRAC Agent submits pilot jobs to CEs.
- Pilots will try to match the WN capabilities to Jobs requirements.
- Jobs are started on WNs. DIRAC monitors its progress.

The basic flowchart describing the evolution of a job's status is shown in figure 2.5 on the facing page

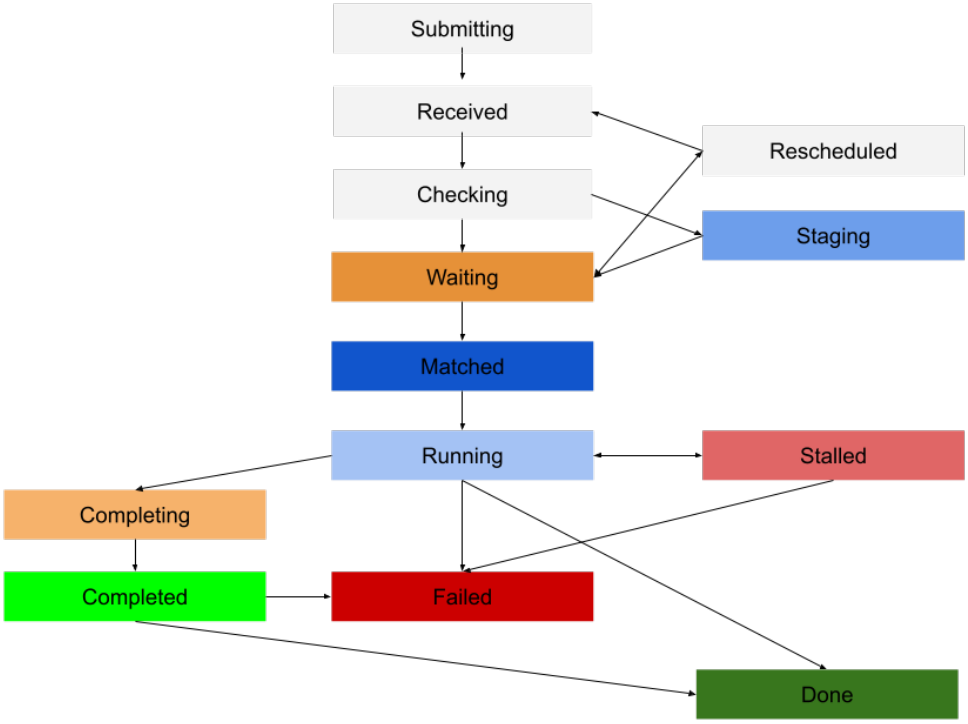

Figure 2.5: The job status of a successful job proceeds in the following order: Submitting, Received, Checking, Waiting, Matched, Running, Completing, Completed, Done. Jobs which return no heartbeat have a status of Stalled and jobs where any workflow modules return an error status are classed as Failed [4].

3

FCC on iLCDirac

3.1 iLCDirac

ILCDirac is an extension of the DIRAC Interware (DIRAC) framework. Its core functionality is built on top of the workflow Application Programming Interface (API) from DIRAC, with dependencies on core DIRAC services such as the File Catalog (FC) and the Transformation System (TS). Currently, several applications used in the Linear Collider (LC) and Future Circular Collider (FCC) community are supported natively by iLCDirac [10].

The approach to designing job and application interfaces is rooted in considerations of ergonomics and flexibility: the guiding principle is to avoid tight coupling between job and application elements. Applications are designed to be definable independent of jobs, with no intertwining of properties. The diverse interfaces of various applications are concealed behind a uniform interface, simplifying usage. This unified interface presents standard application parameters such as name, version, number of events, steering files, input file names, output file names; typically, an application also requires in input a steering file or an option file. Applications, when executed, produce log files and output files, with the latter that may serve as the input for subsequent applications, necessitating a mechanism to connect applications seamlessly.

Applications in general belong to the three main classes: generation, simulation and reconstruction, with some tweaking in the case of fast simulations, that may perform simulation and

reconstruction in a unique process. Any application must be uniquely identifiable by name and possibly version.

All applications in iLCDirac inherit from a parent application class, as shown in figure 3.1 on the next page. This design choice, exemplified by the submission algorithm, includes checks to ensure the sanity of application definitions, such as verifying the existence of the specified version in the central configuration, and the presence of some of the inputs necessary for that specific application.

The Job classes in figure 3.2 on page 22 are also built with generality in mind. A foundational Job class, inherited from the DIRAC Job class, allows for the definition of general job requirements, such as CPU time and required platform. Two primary subclasses enhance this design. The first, UserJob, permits the specification of input data and output data and is tailored for user-initiated tasks. This job type is distinct from ProductionJob, the second subclass. ProductionJob lacks explicit input/output definitions, and these jobs cannot be directly submitted to the DIRAC Workload Management System (WMS). Instead, they are intended for submission to the DIRAC TS, which handles job preparation, definition of the input files, and submission. Despite these differences, both job types share a similar application interface, featuring an 'append' method that accepts an Application instance as an argument. Consequently, the checks applied to user jobs are equally applicable to production jobs, ensuring a uniform application definition experience across contexts. Examples of UserJob are provided in section 3.2 on the next page, while the ProductionJob process is explained more in detail in section 3.6 on page 34.

This modular design makes the integration of FCC necessities into the existing codebase straightforward.

The main addition to the codebase was the creation of the new applications required for FCC detector studies. In particular, it was necessary to add new generators (as described in section 3.2 on the next page), an application for Delphes fast simulations (section 3.4 on page 25) and an application for the Gaudi wrapper (section 3.5 on page 31) used during the reconstruction phase of full simulations. On the contrary, it was not necessary to create new applications for Geant4 based simulations, since the simulation step for FCC full simulations, DDSim (section 3.5 on page 31) stays the same as the one of Compact Linear Collider (CLIC) detector studies, with all the changes happening in the steering files provided in its input.

The presence of multiple possibilities for the choice of generator, the choice between full simulations and fast simulations, and the choice between the different fast simulations inputs, created the necessity of modifying the mechanism responsible for the creation of the entire

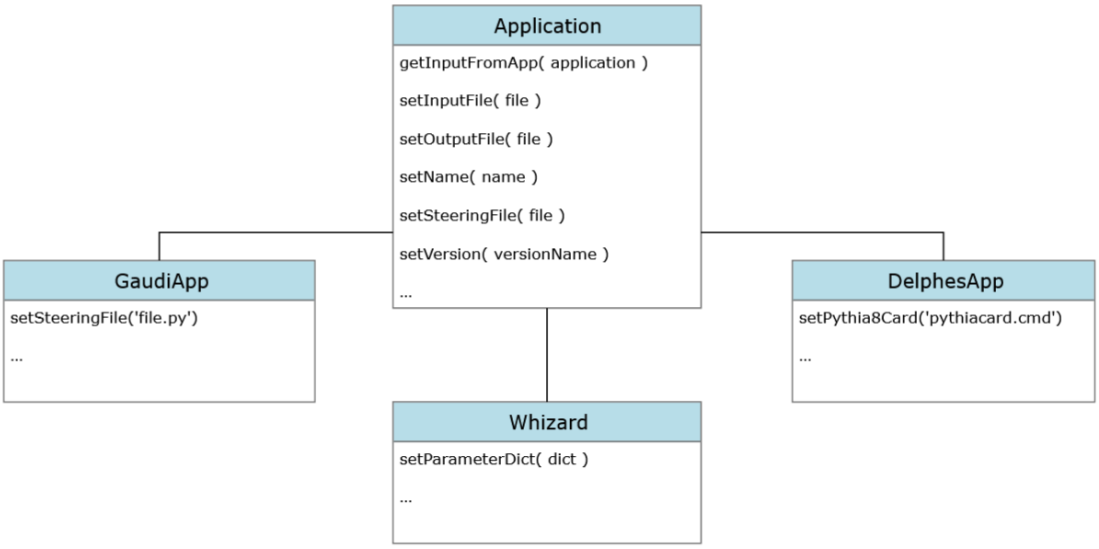

Figure 3.1: iLCDirac application class diagrams. All applications inherit from a parent application class

production chain (section 3.6 on page 34). While for CLIC the production chain is always composed by the same three predefined applications for generation-simulation-reconstruction, for FCC it was necessary to make it more flexible, allowing the choice of different production chains, and dealing with all the now configurable parameters (as for example for input/output file formats, or content of application logs).

Another difference with respect to iLCDirac for CLIC was the new Logical File Name (LFN) convention adopted for FCC productions (section 3.6 on page 34). This was done due to both the presence of more kinds of application outputs and the choice of different paths with the purpose of a better organization of the produced data.

Finally, since additional metadata was requested for the analysis of the produced data, it was necessary to create an agent (described more in depth in section 3.7 on page 39) to iLCDirac for this task, following the standard design of DIRAC agents (section 2.3 on page 10)

3.2 Monte Carlo Generators

Monte Carlo generators are computational algorithms employed in particle physics simulations to model the dynamics of particle interactions and decays within high-energy collision experiments. At their core they utilize quantum field theory to simulate the evolution of particles during collision events, where a broad range of processes happen, including particle creation,

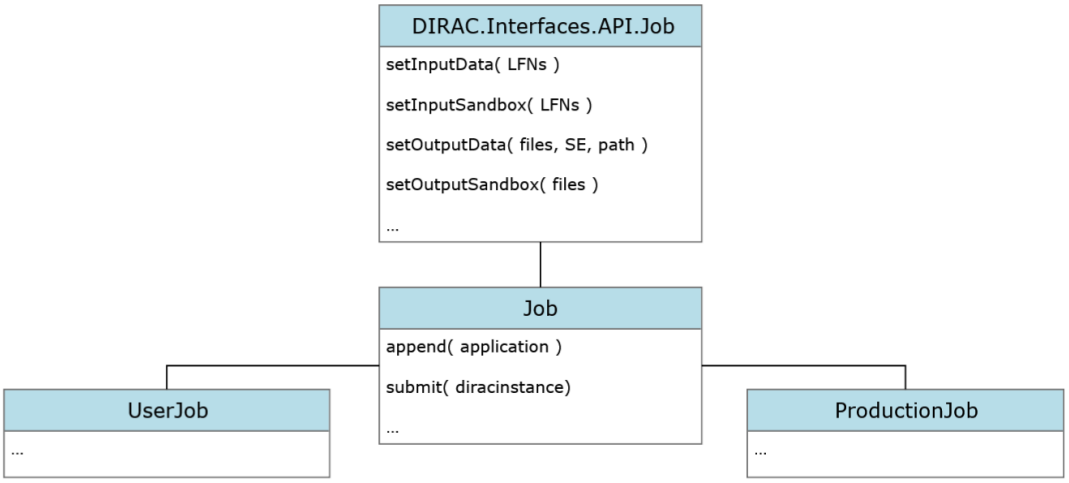

Figure 3.2: iLCDirac job class diagrams. A foundational Job class, inherited from the DIRAC Job class, allows for the definition of general job requirements. Two primary subclasses enhance this design: UserJob, tailored for user-initiated tasks, and ProductionJob, tailored for production jobs on Worldwide LHC Computing Grid (WLCG).

annihilation, scattering, and decay.

By generating a vast ensemble of simulated collision events, Monte Carlo generators provide a comprehensive representation of the potential outcomes observed in experimental settings. These simulations model only the interaction between particles, while their interactions with detector materials and the resulting signatures detected by experimental apparatus is simulated during the phases of simulation and reconstruction (section 3.5 on page 31), or fast simulation (section 3.4 on page 25).

Different generators are developed independently by different research groups or different universities. They can cover different kinds of processes, make use of different approximations, and produce results with specific accuracies depending on several variables, including for example the energy of the process. For this reason, it was requested to include a set of several generators among the available applications for the generation process of FCC detector studies.

The available standalone generators are the following:

- KKMCee, an adaptation of the KKMC Monte Carlo generator to the case of FCCee;
- Whizard-2, a multipurpose event generator that covers all parts of event generation;
- Babayaga, a Monte Carlo generator of two-photon events used at LEP;
- Bhlumi, a Monte Carlo generator of Bhabha events used at LEP for luminosity studies;
- any other generator available as a Gaudi [12] algorithm.

In the case of fast simulations, events can also be generated through the generators coupled

with Delphes executables, as will be shown in section 3.4 on page 25.

Being the first step of the generation-simulation-production pipeline, generators do not receive input data. The generation process can be based simply on a random seed, energy of the beam, and information about the process to generate. As an alternative, it may rely on a more complex input card, like in the case of Whizard-2, that requires a Sindarin file describing the process (figure **??** on page ??).

```
def run():
  """Run The Job."""
  job = UserJob()
  job.setConfigPackage("fccConfig", 'key4hep-devel-2')

  babayaga = Babayaga()
  babayaga.setVersion('key4hep_230408')
  babayaga.setNumberOfEvents(1000)
  babayaga.setEnergy(91.188)
  babayaga.setOutputFile('events.lhe')
  job.append(babayaga)

  job.submit(DiracILC(), mode='local')
```

Figure 3.3: Babayaga user job. Babayaga takes in input the key4hep softwareVersion, the specified energy of the beam, and the number of events to generate. At that point the job containing the application is ready to be submitted locally.

Figure 3.3 shows in detail how a generator application interface works, with the example of Babayaga. The parameters set for the application are then used to automatically generate a bash script (figure 3.4 on the following page). According to the choice of software version, the appropriate release of the key4hep stack ([13]) is sourced, ensuring the reproducibility of the environment, which in each transformation is also logged in a file. The generator's executable available from key4hep is then executed to produce the events, which, in case the application succeeds, will be saved on the appropriate storage element, and added to the DIRAC file catalog with the appropriate metadata.

When available in the logs of the application, metadata of interest (luminosities, cross sections, efficiencies) are collected and added to the metadata: figure 3.1 on the next page shows an example of generation log file that can be parsed to retrieve this information.

The case of a generator like Whizard-2, requiring in input a steering file, is not discussed here, since in section 3.4 on page 25 is discussed a similar case for the inputs of the Pythia generator coupled with the Delphes executable for fast simulations.

```
#!/bin/bash
######################################################################
# Dynamically generated script to run a production or analysis job. #
######################################################################
source /cvmfs/sw.hsf.org/spackages7/key4hep-stack/2023-04-08/x86_64-centos7-gcc11.2
↪  .0-opt/urwcv/setup.sh
echo =========
env | sort >> localEnv.log
echo babayaga:`which babayaga`
echo =========
babayaga --ecms 91.188 --nevts 1000 --outfile someprocess_lhef_12345_28.lhe --debug
↪  0 --seed 1234500000028
declare -x appstatus=$?
cp someprocess_lhef_12345_28.lhe events.lhe
exit $appstatus
```

Figure 3.4: Babayaga autogenerated bash script. According to the choice of software version, the appropriate release of the key4hep stack is sourced, ensuring the reproducibility of the environment, which in each transformation is also logged in a file. The generator's executable available from key4hep is then executed to produce the events.

```
|================================================================================|
| It      Calls  Integral[fb]  Error[fb]   Err[%]    Acc  Eff[%]   Chi2 N[It] |
|================================================================================|
   1       4998  7.9327105E-02  1.06E-03    1.33    0.94*  13.69
   2       4998  7.7751049E-02  5.93E-04    0.76    0.54*  29.31

  15       9999  7.8240593E-02  3.46E-04    0.44    0.44*  19.91
  16       9999  7.7992452E-02  3.50E-04    0.45    0.45   19.52
  17       9999  7.8675085E-02  3.55E-04    0.45    0.45   19.69
|--------------------------------------------------------------------------------|
  17      69993  7.8201031E-02  1.33E-04    0.17    0.45   19.69    0.86   7
|================================================================================|
| Time estimate for generating 10000 events: 0d:00h:00m:01s
decay_proc:
   7.8201031E-02 +- 1.33E-04 fb  (  0.17  %)
n_events = 100
```

Listing 3.1: Portion of Whizard-2 logs. The final values extracted for the metadata are in the line of the last iteration (line 17) and in the field decay_proc. The generator logs are parsed with regex until the final iteration is identified and the values extracted.

3.3 Previously generated events

Generated files clearly do not depend on the detector choices, but only on the physics of the processes. So, as long as a set of events is generated starting from the correct kind of particles and in the correct energy range, they can be recycled not only across different detector design choices in the context of the FCC studies, but also across different accelerators' feasibility studies, for example using events generated for CLIC.

In order to address the need of uploading generated events, a tool for automating the upload of events for FCC studies was created. The functioning of the script is based on the DIRAC Data Management System (DMS) and FC client. This makes possible to copy or upload files to the grid, respecting the convention of LFN decided for FCC data and updating the metadata consistently with the metadata that is usually created when running a generation application.

3.4 Fast simulations

Delphes is a software package used to perform fast simulations of particle detector responses within collider experiments [14]. These fast simulations aim to approximate the behavior of particle detectors in response to collision events, providing a rapid and efficient means of analyzing experimental data.

Delphes utilizes simplified models of particle detectors, modeled to mimic the response of actual detectors to the passage of particles, accounting for effects such as energy deposition, particle identification, and detector resolution.

The simulation includes a track propagation system embedded in a magnetic field, electromagnetic and hadron calorimeters, and PID. Physics objects that can be used for data analysis are then reconstructed from the simulated detector response. These include tracks and calorimeter deposits and high level objects such as isolated electrons, jets, taus, and missing energy.

The primary advantage of Delphes fast simulations lies in their speed and efficiency. By employing simplified detector models and algorithms optimized for performance, Delphes can rapidly process large datasets. While fast simulations do not capture all the physics of real detectors, Delphes simulations still provide a practical means of exploring experimental feasibility, evaluating analysis strategies, and facilitating the interpretation of collider data within the constraints of computational resources and time limitations.

The Delphes fast simulations happen with the Delphes executables made available by k4SimDelphes (which could also be used via Gaudi):

- `DelphesPythia8_EDM4HEP` for using Pythia8 to generate events to use in the fast simulation,
- `DelphesPythia8EvtGen_EDM4HEP_k4Interface` for using EvtGen to generate events to use in the fast simulation,
- `DelphesSTDHEP_EDM4HEP` for reading STDHEP inputs,
- `DelphesROOT_EDM4HEP` for reading ROOT files in the Delphes format.

As anticipated (refer to section 3.2 on page 21), two additional generators (Pythia8 and EvtGen) are available through the Delphes executables.

The different Delphes executables allow multiple workflows (as shown in the schema of figure 3.5), processes different formats of previously generated datasets, or generating data directly before performing the fast simulation.

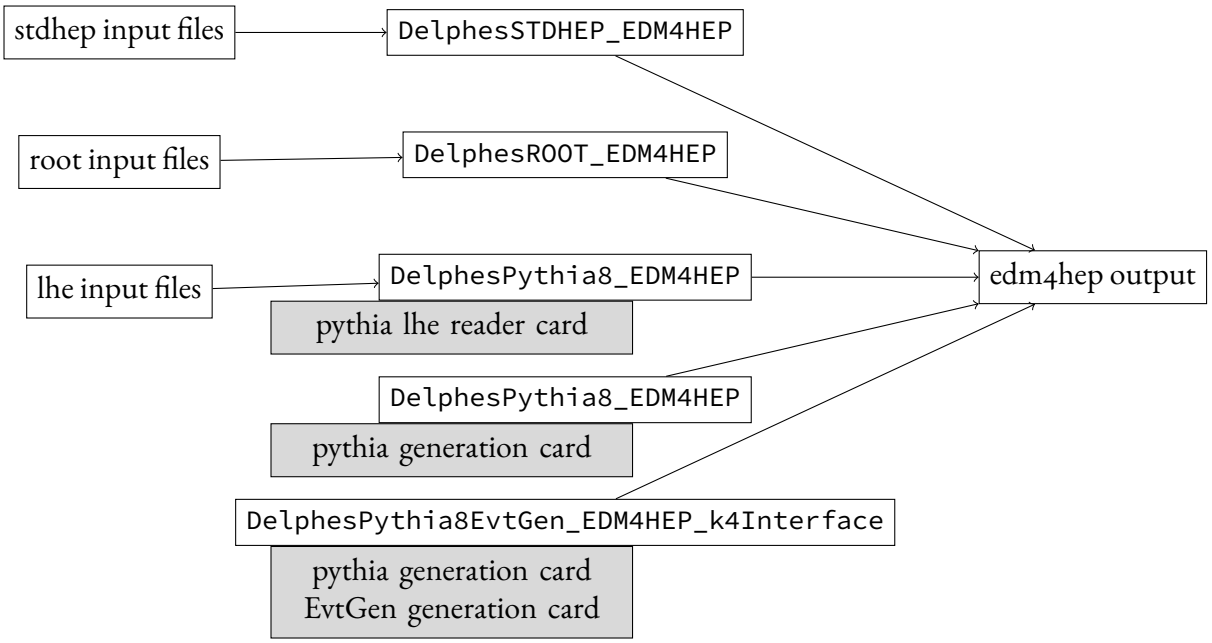

Figure 3.5: Schema of the possible Delphes workflows. Stdhep and edm4hep files can be directly processed with their own specific executable. Delphes lacks an executable for Les Houches Event (LHE) inputs, but it can exploit the Pythia LHE reader capabilities. In order to directly generate events with Pythia, a normal Pythia card is used for describing the process, and similarly happens for the use of EvtGen.

```
~$ DelphesPythia8_EDM4HEP --help
Usage: DelphesPythia8config_file output_config_file pythia_card output_file
config_file - configuration file in Tcl format,
output_config_file - configuration file steering the content of the edm4hep output
↪  in Tcl format,
pythia_card - Pythia8 configuration file,
output_file - output file in ROOT format.
```

Figure 3.6: Usage of DelphesPythia8_EDM4HEP with its inputs.

In iLCDirac the workflows for all the Delphes executables are implemented in a single application module, which, depending on the chosen executable, performs different required operations during the setup of the application.

3.4.1 Fast simulation and Pythia 8

In order to generate events to directly process with Delphes, Pythia can be used through `DelphesPythia8_EDM4HEP`. This executable works using the arguments shown in figure 3.6.

Out of the four inputs, the `output_file` is the easiest to deal with, since it is simply the desired name to be given to the output of the fast simulation. As mentioned in section 3.1 on page 19, the names of the outputs are not decided during the setup of the application, on the client side. In fact, they will be different for each job of the large scale Monte Carlo production, and therefore automatically generated based on the IDs of the job running on each specific machine.

The `output_config_file` and the `config_file` are two cards that in general do not change between different productions; they can be updated, but rarely. For this reason, they are kept in a shared area on CernVM File System (CVMFS), with versioning determined by `configVersion` and `configPackage`. In particular, `config_file` is the configuration card of the detector, containing the description of the detector design.

On the contrary, the `pythia_card` needs to change often: it can differ every time, depending on energy and process chosen for the generation of the events. A complete list of the Pythia cards used is in a shared directory on EOS Open Storage (EOS), but it can not be accessed from the machines of the grid. In general, it can however be accessed from the client side, on the machine of the user submitting the transformation on iLCDirac. The system works by retrieving the cards locally from EOS, and then including them in the XML of the DIRAC workflow (of which an example is shown in figure 3.21 on page 39) as a string, in the same way as it happens

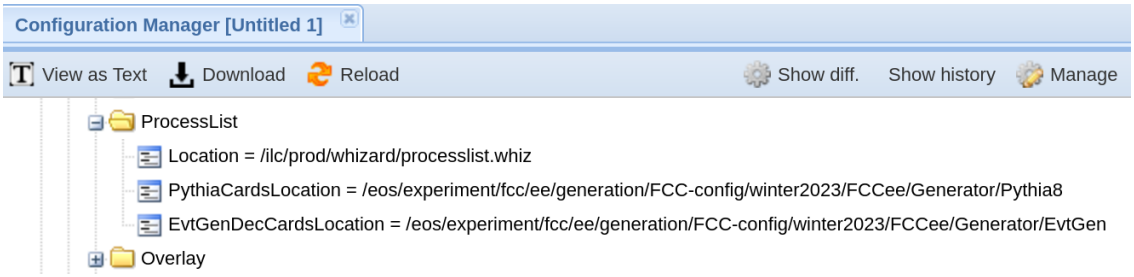

Figure 3.7: DIRAC Configuration Manager. It is shown how the paths to the Pythia cards and EvtGen decay cards are saved. Similarly, are saved also the locations of the detector cards used for full simulations.

for the values of other parameters appearing in the configuration file of the transformations.

The location of the Pythia cards on EOS is not guaranteed to stay the same indefinitely. For this reason, instead of accessing EOS directly, iLCDirac retrieves the path to the EOS Pythia cards directory from the DIRAC Configuration Manager (figure 3.7), that can be easily modified whenever necessary without the need of modifying the codebase.

The other problem arising from the Pythia cards is the consistency with the other parameters used by the transformation, which are also the ones that will be stored in the metadata of the transformation. While setting up the application, the Pythia card is parsed and compared to the transformation parameters, aborting the operation in case of inconsistencies. Parameters like number of events or random seed, that are not related to the physics of the process, will be overwritten just before running the fast simulation, on the grid.

As can be observed in figure 3.8, the structure of the autogenerated bash script is the same as the one of the generation process (section 3.2 on page 21), with the only difference being the Delphes executable replacing the generator.

```
#!/bin/bash
source /cvmfs/sw.hsf.org/spackages7/key4hep-stack/2023-04-08/x86_64-centos7-gcc11.2
↪  .0-opt/urwcv/setup.sh
echo =========
env | sort >> localEnv.log
echo DelphesApp:`which DelphesPythia8_EDM4HEP`
echo =========
DelphesPythia8_EDM4HEP card_IDEA.tcl edm4hep_IDEA.tcl pythia8card.cmd
↪  qqbar_delphes_12345_28.root
declare -x appstatus=$?
exit $appstatus
```

Figure 3.8: DelphesPythia8_EDM4HEP autogenerated bash script

```
~$ DelphesPythia8EvtGen_EDM4HEP_k4Interface --help
Usage: DelphesPythia8EvtGenconfig_file output_config_file pythia_card output_file
↪  DECAY.DEC evt.pdl user.dec
config_file - configuration file in Tcl format,
output_config_file - configuration file steering the content of the edm4hep output
↪  in Tcl format,
pythia_card - Pythia8 configuration file,
output_file - output file in ROOT format,
DECAY.DEC - EvtGen full decay file,
evt.pdl - EvtGen particle list,
user.dec - EvtGen user decay file.
```

Figure 3.9: Usage of DelphesPythia8EvtGen_EDM4HEP_k4Interface with its inputs.

3.4.2 Fast simulation and EvtGen

In order to generate events to directly process with Delphes, Pythia together with EvtGen can be used through `DelphesPythia8EvtGen_EDM4HEP_k4Interface`. This executable works using the arguments shown in figure 3.9.

Out of the seven required inputs, four are the same as of `DelphesPythia8_EDM4HEP` (figure 3.6 on page 27): `output_config_file`, `config_file`, `output_file`, and `pythia_card`); they are treated exactly in the same way. In addition, three other inputs are required.

The EvtGen full decay file and the EvtGen particle list have the same characteristics of `output_config_file` and `config_file`, and are in fact also treated in the same way: kept in shared area in CVMFS and retrieved on the grid just before running the executable. The EvtGen user decay file instead has necessities similar to the Pythia 8 card. The main differences are that no internal controls on the content of the card are performed, and that the card is used to generate other parameters used for the event generation with EvtGen (like particle IDs). The complete command that is executed can be seen in figure 3.10 on the next page, where particle ID, BSignal and numeric code are: `511 Bd_SIGNAL 1`.

3.4.3 Fast simulation and external generator

If the generation process has already happened with a standalone generator, the Delphes fast simulation usually happens by using either of the two executables `DelphesSTDHEP_EDM4HEP` and `DelphesROOT_EDM4HEP`, depending on the data format of the input data.

The usage of the executable is simple (refer to figure 3.2 on the following page): beside the output of the generator, it requires in input a detector card and an output card.

```
#!/bin/bash
source /cvmfs/sw.hsf.org/spackages7/key4hep-stack/2023-04-08/x86_64-centos7-gcc11.2
↪  .0-opt/urwcv/setup.sh
echo =========
env | sort >> localEnv.log
echo DelphesApp:`which DelphesPythia8EvtGen_EDM4HEP_k4Interface`
echo =========
DelphesPythia8EvtGen_EDM4HEP_k4Interface card_IDEA.tcl edm4hep_IDEA.tcl
↪  pythia8card.cmd Zbb_delphes_12345_12345.root DECAY.DEC evt.pdl evtGenCard.dec
↪  511 Bd_SIGNAL 1
declare -x appstatus=$?
exit $appstatus
```

Figure 3.10: DelphesPythia8EvtGen_EDM4HEP_k4Interface autogenerated bash script

```
~$ DelphesSTDHEP_EDM4HEP --help
Usage: DelphesHepMC config_file output_config_file output_file [input_file(s)]
config_file - configuration file in Tcl format,
output_config_file - configuration file steering the content of the edm4hep output
↪  in Tcl format,
output_file - output file in ROOT format,
input_file(s) - input file(s) in STDHEP format,
```

Listing 3.2: Usage of `DelphesSTDHEP_EDM4HEP` standalone executable

The `output_config_file`, `config_file` and `output_file`, being the same as of `DelphesROOT_EDM4HEP` (figure 3.6 on page 27), are treated in the usual way. The only difference is the presence of input files. The bash script generated will be the one in figure 3.11.

```
#!/bin/bash
source /cvmfs/sw.hsf.org/spackages7/key4hep-stack/2023-04-08/x86_64-centos7-gcc11.2.
↪  .0-opt/urwcv/setup.sh
echo =========
env | sort >> localEnv.log
echo DelphesApp:`which DelphesSTDHEP_EDM4HEP`
echo =========
DelphesSTDHEP_EDM4HEP card_IDEA.tcl edm4hep_IDEA.tcl ZHH_delphes_12345_12345.root
↪  /home/lvalenti/Work/configFileProductions/delphesstdhep/events.stdhep
declare -x appstatus=$?
exit $appstatus
```

Figure 3.11: `DelphesSTDHEP_EDM4HEP` autogenerated bash script

In the case of generators producing outputs in LHE format, there are no Delphes executables that can directly process them. In this case, the Pythia 8 capability of being an LHE reader is exploited. The script to be executed will be the same as for the event generation with Pythia 8 (figure 3.11), but using a Pythia card containing instructions for reading an LHE file, instead of the instructions for event generation.

3.5 Full simulations

The goal of a full single particle simulation is to replicate the entire experimental process as closely as possible, from the initial interaction of the particle with the detector to the final reconstruction of its properties.

In this simulation, a detailed model of the particle detector is used to replicate the detector's behavior when interacting with the particle. The simulation takes into account factors such as energy deposition, particle trajectories, and detector resolution, aiming to replicate the real-world response of the detector as accurately as possible.

Following the simulation stage, the reconstructed data obtained from the simulated detector response is analyzed to extract information about the properties of the particle. This reconstruction process involves algorithms that interpret the signals recorded by the detector components, identify particle tracks, measure energy depositions, and infer particle properties such as momentum, energy, and particle type.

3.5.1 Simulation: DDSim

Geant4 is a toolkit to create simulations of the passage of particles or radiation through matter. Applications built on Geant4 can simulate any setup or detector and radiation source, and record chosen output of physical quantities due to source particles and secondaries interacting with the material of the setup.

Geant4 provides comprehensive functionality for simulating particle transport. It enables users to create geometry models with shapes and materials, navigate particle tracks, simulate physics interactions, generate secondary particles, record data, visualize setups and particle tracks, and interact with the simulation through a terminal or graphical user interface.

It includes a complete set of physics processes for electromagnetic, strong and weak interactions of particles in matter over an energy range that starts from milli-eV (for thermal neutrons), eV (electrons) or typically KeV (hadrons), up to hundreds of GeV (or even in part up to 100 TeV). For each type of interaction, a complete set of physics model implementations is provided. Some choices of modeling approaches are available and ready to be used as coherent configurations (named physics lists).

DD4hep is a software framework for providing a complete solution for full detector description (geometry, materials, visualization, readout, alignment, calibration, etc.) for the full experiment life cycle (detector concept development, detector optimization, construction, operation). It offers a consistent description through a single source of detector information for simulation, reconstruction, analysis, etc.

DD4hep is generally the detector description toolkit for high energy physics, used for simulations of detectors using the Geant4 particles through matter simulation code. In addition to providing a detector geometry description layer on top of geant4, DD4hep provides alignment, CAD, digitization, reconstruction, and other modules. Although some projects use custom infrastructure to steer simulations, DD4hep comes standard with DDSim: a python-based utility to run simulations on the command line. DDSim is currently focused on running Geant4 simulations.

After setting up the DDSim application (refer to figure 3.19 on page 37) similarly to what is done for generator applications (figure 3.3 on page 23), iLCDirac proceeds with the generation of the bash script to be executed. Its execution produces an output file in edm4hep data format with Geant4 simulation hits that can then be fed to the reconstruction application.

As shown in the generated bash script in figure 3.12 on the facing page, DDSim requires in input a path to input files, the number of events to simulate, a random seed, and the path for

```
#!/bin/bash
#########################################################################
# Dynamically generated script to run a production or analysis job. #
#########################################################################
source /cvmfs/sw.hsf.org/key4hep/releases/2024-03-10/x86_64-almalinux9-gcc11.3.1-op
↪  t/key4hep-stack/2024-03-10-gidfme/setup.sh
echo =========
env | sort >> localEnv.log
echo ddsim:`which ddsim`
echo =========
ddsim --compactFile
↪  /cvmfs/sw.hsf.org/key4hep/releases/2024-03-10/x86_64-almalinux9-gcc11.3.1-opt/k
↪  4geo/0.20-hapqru/share/k4geo/FCCee/CLD/compact/CLD_o2_v05/CLD_o2_v05.xml
↪  --steeringFile cld_steer.py  --inputFile /home/lvalenti/Work/configFileProducti
↪  ons/whizardddsimgaudi/userjob/Local_144165oh_JobDir/events.stdhep
↪  --numberOfEvents 10  --random.seed 36  --outputFile events_edm4hep.root
declare -x appstatus=$?
exit $appstatus
```

Figure 3.12: DDSim autogenerated bash script. It requires in input a path to input files, the number of events to simulate, a random seed and the path for saving the output. In addition to these, DDSim also requires in input a steering file and a detector file (–steeringFile, –compactFile).

saving the output. In general, DDSim can also digest many MC output formats like hepevt, hepmc, pairs. In addition to these, DDSim also requires in input a steering file and a detector file (`--steeringFile`, `--compactFile`), with the latter that needs to be the correct version specified in the application setup.

In particular, the detector compact file is present on CVMFS in a location that changes periodically, depending on the releases of the key4hep stack, but is always associated to some key4hep environment variables whose names are updated only rarely. The names of these environment variables are stored on the DIRAC Configuration Manager, and retrieved during the application setup, so that the location of the detector compact files can be obtained from them after sourcing key4hep.

3.5.2 Reconstruction: Gaudi

The reconstruction process follows the simulation with DDSim and makes use of the Gaudi wrappers and data format converters. Under the hood, depending on the Gaudi algorithm chosen, the reconstruction can happen in many ways, but an example used in the first iLCDirac production tests is Pandora Particle Flow.

After setting up the Gaudi application (as done in figure 3.19 on page 37) similarly to what is done for DDSim, iLCDirac proceeds with the generation of the bash script. Its execution produces an edm4hep ROOT file with a many new digitalization/reconstruction level collections.

```
#!/bin/bash
source /cvmfs/sw.hsf.org/key4hep/releases/2024-03-10/x86_64-almalinux9-gcc11.3.1-op
↪  t/key4hep-stack/2024-03-10-gidfme/setup.sh
echo =========
env | sort >> localEnv.log
echo gaudiApp:`which k4run`
echo =========
k4run CLDReconstruction.py  --inputFiles /home/lvalenti/Work/configFileProductions/
↪  whizardddsimgaudi/userjob/Local_144165oh_JobDir/events_edm4hep.root -n 10
↪  --GeoSvc.detectors
↪  /cvmfs/sw.hsf.org/key4hep/releases/2024-03-10/x86_64-almalinux9-gcc11.3.1-opt/k
↪  4geo/0.20-hapqru/share/k4geo/FCCee/CLD/compact/CLD_o2_v05/CLD_o2_v05.xml
↪  --PodioOutput.filename
declare -x appstatus=$?
exit $appstatus
```

Figure 3.13: Gaudi autogenerated bash script. It requires in input a path to the simulation output files, a random seed, the number of events to simulate and the path for saving the output. In addition to these, Gaudi also requires in input a steering file and a detector file (`CLDReconstruction.py`, `--GeoSvc.detectors`)

As shown in the generated bash script in figure 3.13, Gaudi requires in input a path to the simulation output files, a random seed, the number of events to simulate and the path for saving the output. In addition to these, Gaudi also requires in input a steering file and a detector file (`CLDReconstruction.py`, `--GeoSvc.detectors`), with the latter being the same one already retrieved in the case of DDSim.

3.6 Creating transformations

For production activities on the grid, iLCDirac relies on the DIRAC TS to submit DIRAC jobs to the grid, and generates them similarly to what was described in the previous sections. The difference is that, in order to have major flexibility, the setup parameters are given in input in a configuration file, and used for the preparation of the ProductionJob under the hood, inside iLCDirac. Every job essentially constitutes a Dirac workflow, and, through this object, users have the ability to load and execute code on the grid machines in a predefined sequence and with specified parameters.

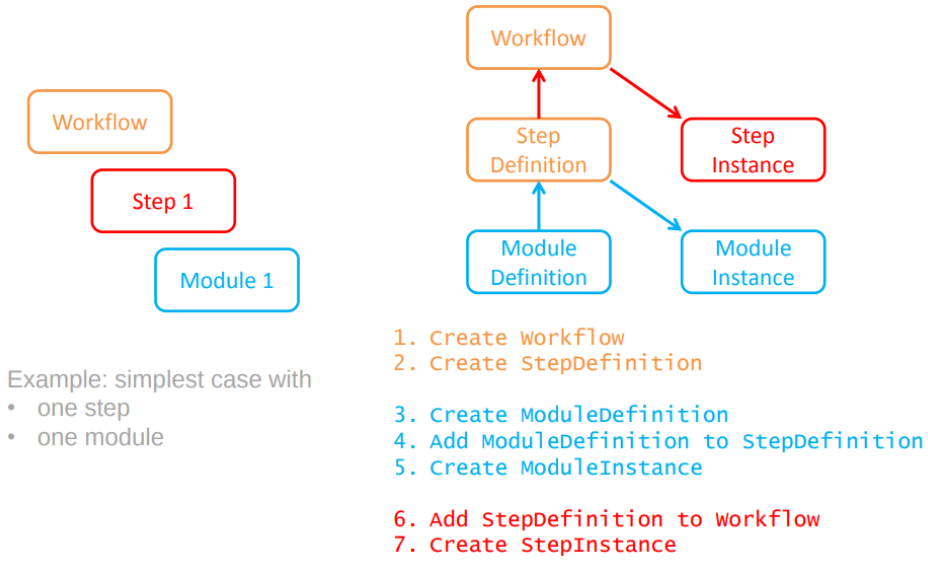

Figure 3.14: The Dirac Workflow [4] serves as a method for outlining Dirac jobs. Through Dirac workflows, users have the ability to load and execute code in a predefined sequence with specified parameters.
Indeed, a workflow comprises steps and modules, each potentially containing parameters. It serves as a container for organizing steps, where a step further acts as a container for modules.

In fact, a DIRAC workflow (whose structure is described in the schema of figure 3.14) is typically depicted in XML format (figure 3.21 on page 39), but it can also be represented in Python format. This dual representation enables DIRAC to directly access names and variables defined within the workflow from Python. On each machine running the jobs, following the DIRAC workflow, bash scripts, similar to the ones described in the previous sections, are automatically generated and executed. Their outputs are saved and stored on the specified Storage Element (SE), and the relative metadata saved to the FC.

Figure 3.17 on the next page and figure 3.20 on page 37 show a comparison between a UserJob and the relative configuration file used for the generation of the ProductionJob, in the case of some large scale tests on the WLCG that also later described in the Conclusions.

All the configuration files share the same structure. They are divided in sections: some application specific sections, containing parameters affecting only a specific application, and a general section containing parameters that affect the transformation chain as a whole. The application parameters are always treated in the same way as in a UserJob (so, as they are treated in the python scripts of figure 3.16 on the next page and figure 3.19 on page 37).

The block `campaign`, `energies`, `processes`, `detectorModel`, together with `machine` determines the path for saving the outputs of the transformation. The affected part of the

```
[delphesapp]
ExecutableName = DelphesPythia8_EDM4HEP
DetectorCard = card_IDEA.tcl
OutputCard = edm4hep_IDEA.tcl
Version = key4hep_230408

[Production Parameters]
machine = ee
prodGroup = several
softwareVersion = key4hep_240310
generatorApplication = delphesapp
generatorSteeringFile =
↪ p8_ee_ZH_ecm240.cmd,
↪ p8_ee_ZZ_ecm240.cmd,
↪ p8_ee_WW_ecm240.cmd
configVersion = key4hep-devel-2
configPackage = fccConfig
eventsPerJobs = 100
numberOfTasks = 1,1,1
campaign = test_spring2024
energies = 240,240,240
processes = ZH,ZZ,WW
detectorModel = idea
productionLogLevel = VERBOSE
outputSE = CERN-DST-EOS
finalOutputSE = CERN-SRM
MoveStatus = Stopped
MoveGroupSize = 10
ProdTypes = Gen
move = False
```

Figure 3.15

```
def run():
  """Run The Job."""
  job = UserJob()
  job.setConfigPackage("fccConfig",
  ↪ 'key4hep-devel-2')

  delphes = DelphesApp()
  delphes.setVersion('key4hep_230408')
  delphes.setExecutableName("DelphesPyt
  ↪ hia8_EDM4HEP")
  delphes.setDetectorCard('card_IDEA.tc
  ↪ l')
  delphes.setOutputCard('edm4hep_IDEA.t
  ↪ cl')
  delphes.setPythia8Card('p8_ee_ZH_ecm2
  ↪ 40.cmd')
  delphes.setEnergy(240)
  delphes.setNumberOfEvents(100)
  delphes.setOutputFile('output.root')

  job.append(delphes)
  job.submit(DiracILC(), mode='local')
```

Figure 3.16: UserJob for, in a single step, combining the generation of events with Pythia8 and the fast simulation with Delphes.

Figure 3.17: A comparison between a UserJob and the relative configuration file used for the generation of the ProductionJob. This is the configuration file used for the fast simulations for the IDEA detector described in the Conclusions.

```
[whizard2]
Version = 2.8.3

[ddsim]
SteeringFile = cld_steer.py

[gaudiapp]
ExecutableName = k4run
SteeringFile = CLDReconstruction.py
InputFileFlag = --inputFiles
OutputFileFlag = --PodioOutput.filename

[Production Parameters]
machine = ee
prodGroup = several
softwareVersion = key4hep_240310
generatorApplication = whizard2
simulationApplication = ddsim
reconstructionApplication = gaudiapp
generatorSteeringFile =
    ↪ wzp6_ee_nunuH_Htautau_ecm240.sin,
    ↪ wzp6_ee_nunuH_Hgg_ecm240.sin,
    ↪ wzp6_ee_nunuH_Hdd_ecm240.sin,
    ↪ wzp6_ee_nunuH_Huu_ecm240.sin,
    ↪ wzp6_ee_nunuH_Hss_ecm240.sin,
    ↪ wzp6_ee_nunuH_Hcc_ecm240.sin,
    ↪ wzp6_ee_nunuH_Hbb_ecm240.sin
configVersion = key4hep-devel-2
configPackage = fccConfig
eventsPerJobs = 1000, 1000, 1000, 1000,
    ↪ 1000, 1000, 1000
numberOfTasks = 1, 1, 1, 1, 1, 1, 1
campaign = test_spring2024
energies = 240, 240, 240, 240, 240,
    ↪ 240, 240
processes = Htautau, Hgg, Hdd, Huu,
    ↪ Hss, Hcc, Hbb
detectorModel = CLD_o2_v05
productionLogLevel = VERBOSE
outputSE = CERN-DST-EOS
finalOutputSE = CERN-SRM
MoveStatus = Stopped
MoveGroupSize = 10
ProdTypes = Gen, Sim, Rec
```

Figure 3.18

```
def run():
  job = UserJob()
  job.setConfigPackage("fccConfig",
  ↪ 'key4hep-devel-2')

  whiz = Whizard2()
  whiz.setVersion('2.8.3')
  whiz.setNumberOfEvents(10)
  whiz.setSinFile('wzp6_ee_nunuH_Hgg_ec
  ↪ m240.sin')
  whiz.setEnergy(240)
  whiz.setEvtType('Hgg')
  whiz.setOutputFile('events.stdhep')
  job.append(whiz)

  ddsim = DDSim()
  ddsim.setVersion('key4hep_240310')
  ddsim.setSteeringFile('cld_steer.py')
  ddsim.setDetectorModel('CLD_o2_v05')
  ddsim.getInputFromApp(whiz)
  ddsim.setOutputFile('events_edm4hep.r
  ↪ oot')
  job.append(ddsim)

  ga = GaudiApp()
  ga.setVersion('key4hep_240310')
  ga.setExecutableName('k4run')
  ga.setSteeringFile('CLDReconstruction
  ↪ .py')
  ga.setDetectorModel('CLD_o2_v05')
  ga.getInputFromApp(ddsim)
  ga.setInputFileFlag('--inputFiles')
  ga.setOutputFileFlag('--PodioOutput.f
  ↪ ilename')
  ga.setOutputFile('gaudioutput.root')
  job.append(ga)

  job.submit(DiracILC(), mode='local')


if __name__ =="__main__":
  run()
```

Figure 3.19: UserJob for generation using Whizard, simulation with DDSim and reconstruction with Gaudi

Figure 3.20: A comparison between a UserJob and the relative configuration file used for the generation of the ProductionJob. This is the configuration file used for the full simulations for the CLD detector described in the conclusions.

output path will be:
`.../fcc/ee/campaign/energy/process/detectorModel/datatype/DiracProdID`,
where:

- `datatype` is determined automatically based on application and workflow of the transformation chain,
- `DiracProdID` are the job IDs, assigned to the transformations during the submission on DIRAC,
- `detectorModel` will be included only in the output path of transformations that involve the interaction with a detector (as opposed to event generations),

resulting in something like:
`/fcc/ee/spring2024/240gev/Htautau/CLD_o2_v05/sim/00012345`.

The parameters `softwareVersion` and `configPackage` work exactly as in the UserJobs, but applied to all the transformations generated with the configuration file.

All the parameters just described, as also happens for the application specific parameters, are saved (either directly in the file paths or in the transformation metadata) so that all the data produced is tied to both the physical parameters of the process and to the versions of the software used. This, together with the random seed used in the generation process, ensures the complete reproducibility of all the jobs submitted to the grid.

The physical location of the replicas of the outputs is decided by `outputSE`: in these cases it is EOS.

Figure 3.15 on page 36 shows the use of `DelphesPythia8_EDM4HEP` executable for, in a single step, combining the generation of events with Pythia8 and the fast simulation with Delphes (section 3.4 on page 25). The choice of using Delphes with Pythia8 as a generator is done with `generatorApplication = delphesapp`, and, due to the setting `ProdTypes = Gen` the combination of the two operations of generation and fast simulation in a single transformation is treated as a generation for all the purposes of data storage and logging.

The parameters `eventsPerJobs` and `numberOfTasks` decide respectively the number of events per job and the total number of jobs to submit in the transformation.

The Delphes applications gets in input `ExecutableName`, `DetectorCard`, `OutputCard` and, in case, `Version`. It also receives in input a `generatorSteeringFile`, that, in the case of a `DelphesPythia8_EDM4HEP` executable is treated by default as a Pythia card.

As shown in the figure, some fields actually receive in input a list instead of a single value. This happens because the submission script accepts configuration files describing multiple (three in this case) transformation chains to submit to the grid.

```
<name>whizard2_step_1</name>
<type>whizard2_step_1</type>
<Parameter name="applicationName" type="string" linked_module=""
↪  linked_parameter="" in="False" out="False" description="Application
↪  Name"><value><![CDATA[whizard2]]></value></Parameter>
<Parameter name="applicationVersion" type="string" linked_module=""
↪  linked_parameter="" in="False" out="False" description="Application
↪  Version"><value><![CDATA[2.8.3]]></value></Parameter>
<Parameter name="applicationLog" type="string" linked_module="" linked_parameter=""
↪  in="False" out="False" description="Log
↪  File"><value><![CDATA[whizard2_2.8.3_@{STEP_ID}.log]]></value></Parameter>
<Parameter name="OutputFile" type="string" linked_module="" linked_parameter=""
↪  in="False" out="False" description="Output
↪  File"><value><![CDATA[Hdd_stdhep.stdhep]]></value></Parameter>
<Parameter name="OutputPath" type="string" linked_module="" linked_parameter=""
↪  in="True" out="False" description="Output File path on the grid"><value><![CDAT⌋
↪  A[/fcc/ee/test_spring2024/240gev/Hdd/stdhep]]></value></Parameter>
```

Figure 3.21: Snippet from the XML workflow generated for Whizard-2 with the configuration file of figure 3.18 on page 37.

Figure 3.18 on page 37 shows a full simulation (section 3.5 on page 31): generation using Whizard, simulation with DDSim and reconstruction with Gaudi (with Pandora Particle Flow).

Every setting works following the same logic of the previous case of Delphes, with the only exception of the Whizard `Version`, that, by design, is set independently, and does not depend on the key4hep release. The algorithms of Gaudi and DDSim are contained in the `SteeringFile` parameter.

In total, this configuration card will generate 21 transformations: seven transformation chains (as many as the Whizard Sindarin files given input to the `generatorSteeringFile`) of three transformations each, as set in `ProdTypes = Gen, Sim, Rec`.

3.7 Development of an agent for additional metadata

As anticipated in section 3.1 on page 19, requirements of additional metadata for FCC analysis led to the creation of a new iLCDirac agent: `TransformationFileMetadataAgent`. The design of the agent follows the standard design of DIRAC agents, described in section 2.3 on page 10.

In particular, it was requested to create a JSON, to be kept stored on EOS. The JSON contains, for every FCC transformation submitted on the grid, a dictionary with some metadata (as in figure 3.22). Part of the metadata stored in the dictionary (`cross-section`, `cross-section-error`, `efficiency`) is the metadata collected from the logs of the generators, or set when uploading previously generated data, as is described in section 3.2 on page 21 and in section 3.3 on page 25, while the values of the remaining fields are obtained from the DIRAC services.

```
"16379": {
  "Status": "Active",
  "Version": 0,
  "cross-section": null,
  "cross-section-error": null,
  "efficiency": null,
  "total-number-of-events": 9000,
  "number-of-events-per-file": 3000,
  "production-manager": "/DC=ch/DC=cern/OU=Organic
  ↪  Units/OU=Users/CN=lvalenti/CN=864060/CN=Lorenzo Valentini",
  "path": {
    "1195057": "/fcc/ee/spring2024/240gev/ZH/idea/delphes/00016379"
  }
```

Figure 3.22: Partial content of the JSON generated by the agent. It contains, for every FCC transformation submitted on the grid, a small dictionary with metadata associated to the transformation in the DIRAC TS, or associated (in the FC) to the output files of the transformation (or, in the case of simulations or reconstructions, the outputs of the generation that simulation and reconstruction are based upon).

Upon initialization, the agent establishes connections to the essential DIRAC services: Transformation Client, FC Client, Data Manager, and Configuration Manager.

During execution, the agent retrieves a compressed JSON file that was generated during its previous execution. Using predefined conditions to filter the transformations of interest for FCC, the agent also retrieves a list of currently active transformations from the Transformation Client, and checks if any of them has undergone any activity since the previous execution of the agent.

The metadata directly associated to the transformation is retrieved directly by querying the Transformation Client. On the other hand, values like `cross-section`, `cross-section-error` and `efficiency` are associated to files, since they depend on the outcome of the generation process. In that case, by calling the FC, the agent retrieves all the LFN of files with metadata that associates them to the transformation. The agent then proceeds to loop through all the files, and, taking care of error propagation, aggregate the values from individual files into single expected values for the entire transformation.

For transformations like simulation and reconstruction, however, the process just described can not work directly, since the files associated to the transformations are the outputs of simulation and reconstruction themselves, and not the outputs of the original generation. In this case, the agent uses again the FC and the metadata of the transformation in order to identify its "ancestor" transformation (the generation process the rest of the transformation chain comes from), and proceeds to loop through its outputs as just described.

The agent includes robust error handling mechanisms to address potential issues during execution. Whether it encounters failures in downloading files, retrieving transformation details, or uploading metadata, the agent is equipped to handle these scenarios, avoiding disruptions to the metadata management workflow.

Subsequently, the agent constructs a comprehensive dictionary containing relevant information about each transformation, and, after iterating over all the active transformations, updates the JSON it had initially retrieved from the grid.

Once the metadata has been updated, the agent saves and compresses the JSON file before uploading it back to the grid. This process is repeated daily, ensuring that the latest transformation metadata is readily accessible to users and systems within the DIRAC ecosystem.

4

Conclusions

ILCDirac was successfully developed to support the FCC production workflows.

At the end of the development, extensive testing took place, verifying the correct functioning of the infrastructure for the new supported applications. In addition to individual testing for each workflow, specific productions of significant interest underwent testing using large datasets. The objective was preparing to expand the scale of these productions further, following an analysis of the outputs and confirmation that they meet the expected criteria.

A first production consisted in Delphes fast simulations for the IDEA detector (figure 4.3 on page 45) after generation with Pythia, using the Delphes executable `DelphesPythia8_EDM4HEP`:

- e^+e^- annihilation to Z bosons decaying to $b\bar{b}$ quark-antiquark pairs, at a center-of-mass energy of 91 GeV. Its performance on the grid nodes is shown in figure 4.1 on the following page.
- e^+e^- annihilation to WW boson pairs at a center-of-mass energy of 240 GeV.
- e^+e^- annihilation to Z and Higgs boson pairs at a center-of-mass energy of 240 GeV.
- e^+e^- annihilation to ZZ boson pairs at a center-of-mass energy of 240 GeV.

Other productions involved full simulations for the CLD detector (figure 4.2 on the next page).

A first class of transformations was heavy flavor tagger training for Higgs bosons decays into other particles including gluons, bottom quarks, charm quarks, strange quarks, up quarks, down quarks, and tau leptons at a center-of-mass energy of 240 GeV. The transformations consisted in the full chain generation-simulation-reconstruction using Whizard, DDSim, Gaudi.

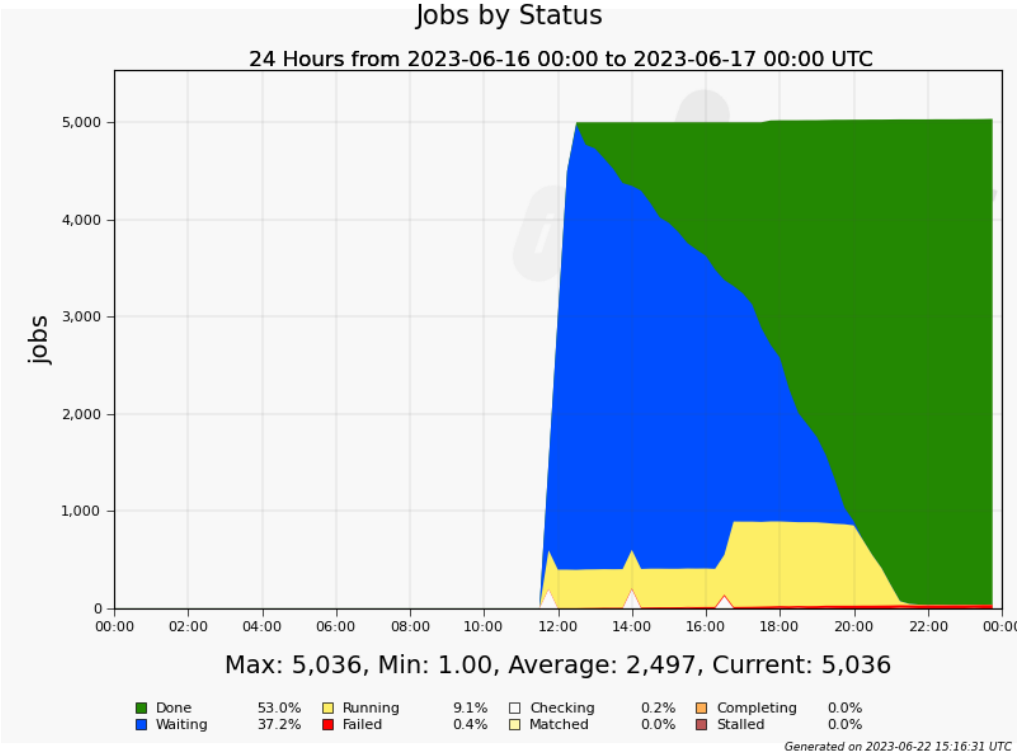

Figure 4.1: Large scale FCC production. The production was completed in about 10 hours. The failure rate was very low, with 36 failures out of 5000 jobs. The failed job were automatically replaced by 36 new jobs, until the goal of 5000 successful jobs of 100k events each was reached, for a total of 500M events.

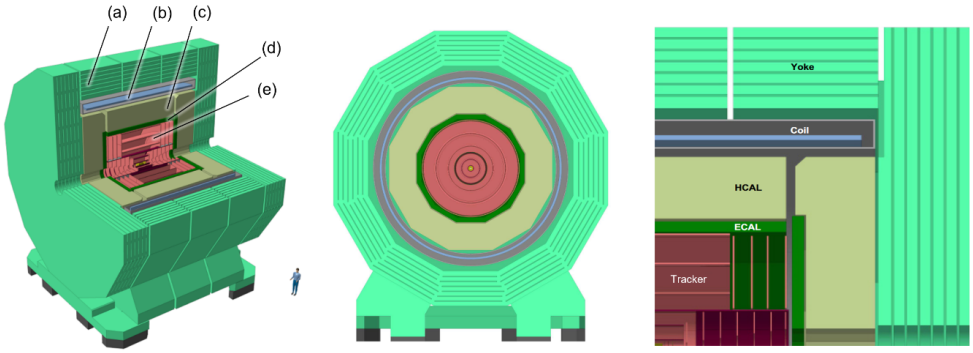

Figure 4.2: CLD detector

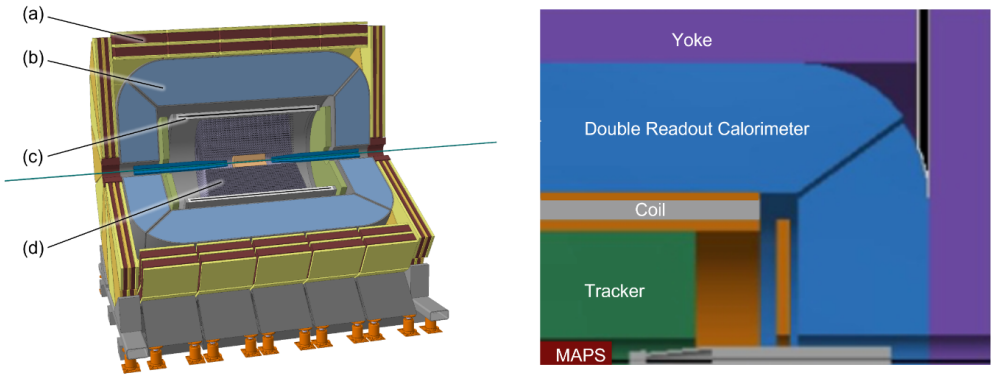

Figure 4.3: IDEA detector

A second class was the collision of e^+e^- leading to the production of Z bosons later decaying to up, down or strange quark-antiquark pairs. The transformations consisted in simulation and reconstruction of events at three distinct center-of-mass energies: 91 GeV, 100 GeV, and 200 GeV. In particular, this was a case in which previously generated datasets from CLIC studies were available, and could be uploaded used as a starting point for the simulation.

The testing was satisfactory, proving that iLCDirac is ready for supporting the future simulation campaigns.

In the short term, anticipated advancements include the testing of iLCDirac with new transformations, without substantial changes in the software architecture. In particular, two classes of productions have already been planned:

- a series of full simulations for the three FCC detector concepts (IDEA, CLD, ALLEGRO) using electrons, muons, and pions at various energy points;
- a series of full simulations of Z bosons decays to electrons and muons, to study the tracking performance.

In the long term, prospective developments may encompass the incorporation of support for additional MC generators or the gradual replacement of some existing applications as they become accessible as Gaudi algorithms.

References

[1] M. Benedikt, A. Blondel, P. Janot, M. Mangano, and F. Zimmermann, "Future circular colliders succeeding the lhc," *Nature Physics*, vol. 16, no. 4, pp. 402–407, 2020.

[2] M. Benedikt, A. Blondel, P. Janot, M. Klein, M. Mangano, M. McCullough, V. Mertens, K. Oide, W. Riegler, D. Schulte *et al.*, "Future circular colliders," *Annual Review of Nuclear and Particle Science*, vol. 69, pp. 389–415, 2019.

[3] N. Kutovskiy, V. Mitsyn, A. Moshkin, I. Pelevanyuk, D. Podgayny, O. Rogachevsky, B. Shchinov, V. Trofimov, and A. Tsaregorodtsev, "Integration of distributed heterogeneous computing resources for the mpd experiment with dirac interware," *Physics of Particles and Nuclei*, vol. 52, pp. 835–841, 07 2021.

[4] "Dirac interware: documentation," https://dirac.readthedocs.io/en/latest/index.html, accessed: 2024-03-13.

[5] "Fccsw: Software for the future circular collider." https://hep-fcc.github.io/fcc-tutorials/master/index.html, accessed: 2024-03-13.

[6] J. Shiers, "The worldwide lhc computing grid (worldwide lcg)," *Computer Physics Communications*, vol. 177, no. 1, pp. 219–223, 2007, proceedings of the Conference on Computational Physics 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S001046550700077X

[7] J. F. Molina, A. Forti, M. Girone, and A. Sciaba, "Operating the worldwide lhc computing grid: current and future challenges," *Journal of Physics: Conference Series*, vol. 513, no. 6, p. 062044, jun 2014. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/513/6/062044

[8] F. Stagni, A. Tsaregorodtsev, A. Sailer, and C. Haen, "The dirac interware: current, upcoming and planned capabilities and technologies," *EPJ Web of Conferences*, vol. 245, p. 03035, 01 2020.

[9] I. Pelevanyuk and A. Tsaregorodtsev, "Dirac interware as a service for high-thoughput computing in jinr," *Physics of Particles and Nuclei Letters*, vol. 19, pp. 580–582, 10 2022.

[10] C. Grefe, S. Poss, A. Sailer, and A. Tsaregorodtsev, "Ilcdirac, a dirac extension for the linear collider community," vol. 513, 06 2014.

[11] O. Viazlo and A. Sailer, "Efficient iterative calibration on the grid using ilcdirac," *EPJ Web of Conferences*, vol. 245, p. 03003, 01 2020.

[12] M. Clemencic, H. Degaudenzi, P. Mato, S. Binet, W. Lavrijsen, C. Leggett, and I. Belyaev, "Recent developments in the lhcb software framework gaudi," in *Journal of Physics: Conference Series*, vol. 219, no. 4. IOP Publishing, 2010, p. 042006.

[13] G. Ganis, C. Helsens, and V. Völkl, "Key4hep, a framework for future hep experiments and its use in fcc," *The European Physical Journal Plus*, vol. 137, no. 1, p. 149, 2022.

[14] P. Azzi, L. Gouskos, M. Selvaggi, and F. Simon, "Higgs and top physics reconstruction challenges and opportunities at fcc-ee," *The European Physical Journal Plus*, vol. 137, no. 1, p. 39, 2021.

Acknowledgments

Firstly, I want to thank André, an amazing supervisor, for the continuous support and patience with which he has guided me during this year. Working and learning from him has been a wonderful experience, from both the human and professional point of view.

Then, I want to thank my professor Marco Zanetti, for being my internal supervisor for this thesis, but more importantly for making the Master program of Physics of Data what it is. I am extremely grateful for the opportunities that this choice of studies is providing me.

I want to thank all my friends for the support and the pleasure of working and spending time together, at home, at the university, in Geneva.

I finally want to thank my parents and my brother for supporting me during this journey, celebrating with me the happy events and bearing with me during the difficult ones:)