Domain Specific Language for Magnetic Measurements at CERN

Relatore: Ch.mo Prof. Pasquale Arpaia
Candidato: Carlo Petrone
Matr. 392/69

Co-relatore: Ing. Giuseppe La Commara

ANNO ACCADEMICO 2008/2009
To my Parents
Acknowledgments

I would like to thank my supervisor Prof. Pasquale Arpaia for his enthusiasm and for having encouraged me by providing a strong motivation to do well my thesis work.

I would like to express also my gratitude to Marco Buzio, my CERN supervisor, and Louis Walckiers section leader, for their help.

I would like to express my appreciation to my tutor, Ing. Giuseppe La Commara. His guide, helpfulness, and patience were important to achieve this final result.

My gratitude also goes to Christos Zamantzas for the trust he had in me, from our first meeting at CERN.

I would like to thank Giancarlo Golluccio, Giuseppe Montenero, Lucio Fiscarelli, Nicola Cardines and Vitaliano Inglese. We worked together since I came in the TE department and we had a good time together and their works were great contributions to my final thesis for the master degree.

A very special thanks also goes to Cosimo, Marianna, Nadia, Tiziana and Vincenzo for their moral support and practical assistance for the realization of this work; I cannot do a list of all my friends equally indispensable, because I am afraid to forget someone.

I would like to say “grazie” to my father Vincenzo, my mother Antonietta, my sisters Angela and Silvia, my brother-in-law Domenico and last but not least my girlfriend Carla; they have supported me by showing their deep and unconditioned affect, the most precious contribution for my work.
6.1.3 Defining the Grammar ................................................................. 92
6.1.4 Generating the DSL Editor ....................................................... 93
6.1.5 Running the Editor ................................................................. 94
6.2 Code Generation with XPand ...................................................... 95
6.2.1 The Grammar Language .......................................................... 96
6.3 Type Rules ................................................................................. 97
6.3.1 Assignment tokens / Properties ............................................. 97
6.3.2 Cross References ................................................................. 98
6.3.3 Metatype Inheritance ............................................................. 99
6.4 Defining the MDSL ................................................................. 100

7 EXPERIMENTAL RESULTS ............................................................. 104

7.1 System Architecture ............................................................... 105
7.2 Overview of the Test Bench at SM18 ........................................ 107
7.3 Measurement Setup ............................................................... 109
7.4 DC Measurements ................................................................. 109
7.5 Measurement Procedure ....................................................... 111
7.6 Data Analysis .......................................................................... 112
7.7 Results ..................................................................................... 113
7.8 Standard AC Measurement for Field Quality ......................... 127
7.8.1 LHC machine cycle ............................................................ 128
7.8.2 Measurement Procedure ..................................................... 130
7.8.3 Analysis decay and snapback ............................................. 131
7.8.4 Results ................................................................................ 131

CONCLUSIONS .............................................................................. 137

APPENDIX ...................................................................................... 139

BIBLIOGRAPHY ............................................................................. 152
# Index of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The accelerator chain at CERN</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Overview of the Geneva area with a drawn of the two circular accelerators</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>ATLAS: A large Toroidal LHC ApparatuS</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Alice: A Large Ion Collider Experiment at CERN LHC</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>CMS: The Compact Muon Solenoid an Experiment for the LHC at CERN</td>
<td>7</td>
</tr>
<tr>
<td>1.6</td>
<td>LHCb: Large Hadron Collider beauty experiment</td>
<td>7</td>
</tr>
<tr>
<td>1.7</td>
<td>Functional drawing of the Large Hadron Collider</td>
<td>8</td>
</tr>
<tr>
<td>1.8</td>
<td>The LHC superconducting dipole: a) Magnetic field; b) particu-lars</td>
<td>12</td>
</tr>
<tr>
<td>1.9</td>
<td>Scheme of the LHC cell</td>
<td>13</td>
</tr>
<tr>
<td>1.10</td>
<td>Cross section of a superconducting quadrupole magnet for the LHC project</td>
<td>16</td>
</tr>
<tr>
<td>1.11</td>
<td>Current distributions to generate a quadrupole induction field</td>
<td>16</td>
</tr>
<tr>
<td>1.12</td>
<td>Cross section of an LHC normal quadrupole</td>
<td>17</td>
</tr>
<tr>
<td>2.1</td>
<td>Magnetics flux through a cylindrical surface</td>
<td>24</td>
</tr>
<tr>
<td>2.2</td>
<td>The rotating coils shaft</td>
<td>28</td>
</tr>
<tr>
<td>2.3</td>
<td>Cross section of the shaft for rotating coil measurements</td>
<td>29</td>
</tr>
<tr>
<td>2.4</td>
<td>The dipole field $B_1$ and the normal sextupole component $b_3$</td>
<td>30</td>
</tr>
<tr>
<td>2.5</td>
<td>Examples are shown for a sextupole ring with 3</td>
<td>31</td>
</tr>
<tr>
<td>2.6</td>
<td>The TRU unit</td>
<td>35</td>
</tr>
<tr>
<td>2.7</td>
<td>Motor for rotating coil in a long dipole magnet in the SM18 laboratory</td>
<td>36</td>
</tr>
<tr>
<td>2.8</td>
<td>The MRU unit (a) is attached directly to the magnet anticryostat(b)</td>
<td>37</td>
</tr>
<tr>
<td>2.9</td>
<td>Connection scheme for absolute $U_A$ and compensated signals $U_A - U_B$ (Left)</td>
<td>37</td>
</tr>
<tr>
<td>2.10</td>
<td>Principle of PDI based on Voltage to Frequency conversion</td>
<td>38</td>
</tr>
<tr>
<td>2.11</td>
<td>Principle of FDI (Fast Digital Integrator)</td>
<td>40</td>
</tr>
<tr>
<td>3.1</td>
<td>The FFMM Architecture</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Simplified cost prediction for DSL-based methodologies [Devanbu, 1998]</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Domain analysis, taken from [Prieto-Diaz, 1990]</td>
<td>61</td>
</tr>
<tr>
<td>4.3</td>
<td>History of software engineering</td>
<td>65</td>
</tr>
<tr>
<td>4.4</td>
<td>Semantic and Syntax mapping</td>
<td>67</td>
</tr>
<tr>
<td>4.5</td>
<td>Metamodel.</td>
<td>69</td>
</tr>
<tr>
<td>5.1</td>
<td>The multi-layered FFMM architecture</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>A typical FFMM configuration</td>
<td>76</td>
</tr>
<tr>
<td>5.3</td>
<td>Test Engineer and Developer Application User roles in measurement software DSL</td>
<td>78</td>
</tr>
<tr>
<td>5.4</td>
<td>MDSL transformation in code</td>
<td>80</td>
</tr>
<tr>
<td>5.5</td>
<td>List of events</td>
<td>81</td>
</tr>
</tbody>
</table>
Index of figures

Figure 5.6: Proposed architecture 83
Figure 6.1: Wizard to start new Xtext project 91
Figure 6.2: DSL grammar 93
Figure 6.3: Generate Xtext artifacts 94
Figure 6.4: Deployment of the DSL plug-ins 95
Figure 6.5: Xpand template 96
Figure 6.6: Example of assignment operators in our project 98
Figure 6.7: Entity 98
Figure 6.8: Abstract type rule 99
Figure 6.9: Token rule expressed 100
Figure 6.10: Comments 100
Figure 6.11: DSL test engineer steps 101
Figure 6.12: Assistance to the measurement procedure 101
Figure 6.13: The part of the Script in C++ 103
Figure 6.14: The part of the Script in DSL 103
Figure 7.1: Architecture of the new measurement system 107
Figure 7.2: Test bench F1 at SM18: main bending dipole (left) and six clusters at SM18 (right) 107
Figure 7.3: Portable Power Supply Heinzinger PTN 135-20 at SM18 108
Figure 7.4: Digital Multimeter KEITHLEY 2000 108
Figure 7.5: Main field component of LHC 114
Figure 7.6: Standard deviation of the Bz mean versus angular speed 115
Figure 7.7: \(\sigma(B_z)\) as a function of angular speed \((N\text{ variable})\) and time interval \((N=127)\) 116
Figure 7.8: \(\bar{\sigma}(B_z(k))\) as a function of the angular speed over the same measurement time 116
Figure 7.9: Sextupole component of LHC dipole versus angular speed at fixed FDIs gain 117
Figure 7.10: Decapole component of LHC dipole versus angular speed at fixed gain 117
Figure 7.11: Standard deviation of the b3 mean versus angular speed 118
Figure 7.12: Standard deviation of the b5 mean versus angular speed 118
Figure 7.13: 8 11th component of LHC dipole measured 119
Figure 7.14: Normal components of the magnetic field 120
Figure 7.15: Skew components of the magnetic field 120
Figure 7.16: Main field component of LHC dipole 121
Figure 7.17: Main field component of LHC dipole measured versus several angular speed 122
Figure 7.18: Standard deviation of the b3 mean versus angular speed 122
Figure 7.19: Sextupole component of LHC dipole measured 123
Figure 7.20: \(\sigma(b_3)\) as a function of angular speed and measurement condition 123
Index of figures

Figure 7.21: Decapole component of LHC dipole measured 124
Figure 7.22: 11th harmonic versus Angular Speed for different FDIs gain 124
Figure 7.23: Normal components of the magnetic field in the second aperture 125
Figure 7.24: Skew components of the magnetic field in the second aperture 126
Figure 7.25: Mean value of B₁ over 30 measurements, a ± 3 σ bar is displayed 126
Figure 7.26: Mean values of the harmonic coefficients from b₂ to b₁₁ 127
Figure 7.27: Mean values of the harmonic coefficients from a₂ to a₁₁ 127
Figure 7.28: The standard reference machine cycl 129
Figure 7.29: Main dipole field in the second aperture of the MBBR 2427 132
Figure 7.30: Normal sextupole as a function of the supply current 133
Figure 7.31: Normal decapole as a function of the supply current 134
Figure 7.32: b₃ base line (blue) and polynomial fitting curve (red) 134
Figure 7.33: b₃ Normal Sextupole (blue) superposed to the fitted b₃ base line 135
Figure 7.34: b₃ snapback obtained by taking off the fitted base line curve 135
Figure 7.35: b₃ snapback measured in the second aperture of the MBBR 2427 136
Figure 7.36: logarithm scale plot of the b₃ snapback (blue) and the exponential fitting (red) 136
INTRODUCTION

CERN, the European Organization for Nuclear Research, is one of the world’s largest and most respected centres for scientific research. Founded in 1954, the CERN Laboratory sits astride the Franco–Swiss border near Geneva. It was one of Europe’s first joint ventures and now has 20 Member States.

Its main purpose is fundamental research in particle physics, namely investigating what the Universe is made of and how it works. At CERN, the design and realization of the new particle accelerator, the Large Hadron Collider (LHC), has required a remarkable technological effort in many areas of engineering. In particular, the tests of LHC superconducting magnets disclosed new horizons to magnetic measurements.

At CERN, the objectively large R&D effort of the Technology Department/Magnets, Superconductors and Cryostats (TE/MSC) group identified areas where further work is required in order to assist the LHC commissioning and start-up, to provide continuity in the instrumentation for the LHC magnets maintenance, and to achieve more accurate magnet models for the LHC exploitation.

In view of future projects, a wide range of software requirements has been recently satisfied by the Flexible Framework for Magnetic Measurements (FFMM), designed also for integrating more performing flexible hardware. FFMM software applications control several devices, such as encoder boards, digital integrators, motor controllers, transducers. In addition, they synchronize and coordinate different measurement tasks and actions.

FFMM has been developed with the aim of helping the user to write high quality code, in terms of flexibility, reusability, portability and efficiency. The test engineer needs to provide a formal description of the measurement procedure (script), in order to automatically generate executable measurement applications.
FFMM needs a formal description of the measurement procedure to be provided in C++, and therefore requires knowledge of this programming language and its rules. In this thesis, the proposed idea is the development of a new easy Measurement Domain Specific Language (MDSL). Such a language models the domain of interest and provides the user with easy programming tools capable of describing the measurement application.

In this way, concise and bug free specific applications can be generated by test engineers who do not have to be skilled programmers. At the SM18 CERN magnet test facility the field experience with the current FFMM release 3.0, highlights that a significant part of the ongoing operation costs is related to the development and maintenance of test applications.

In contrast to a general-purpose programming language (GPL), a domain-specific language (DSL) is designed to allow specific complete applications to be built efficiently and quickly, yielding to programs easy to write, understand, reuse, and maintain. These advantages are making DSLs very popular and their design and implementation are becoming increasingly an intensive area of research. Programming with a DSL also contributes to safety and reduces software errors. Additionally, in practice, high-level constructs translate into the reuse of validated components.

A Measurement Domain Specific Language (MDSL) for the definition of test procedures, the synchronization of the measurement tasks and the configuration of instruments is proposed. The design and the development were carried out in the framework of cooperation between the TE/MSC department of CERN and the Department of Engineering of the University of Sannio. In this thesis, the design, implementation and experimental verification of the domain specific language are presented.

In particular, in chapter 1÷4, the magnetic measurements and test domain of the FFMM at CERN are highlighted. In chapter 5, the approach and the main components of the proposed DSL are illustrated. In chapter 6 and 7, the MDSL implementation for FFMM and experimental results are respectively described.
PART I - STATE OF THE ART
Chapter 1

1 Superconducting Magnets for Accelerators at CERN

In this chapter, after an overview of the main research projects of the European Organization for nuclear Research (CERN), the basic concepts of linear and circular accelerators are described by highlighting the trade-off among geometrical dimension, magnetic field intensity, and electrical field. Then, the rationale for main LHC design choices is explained, by giving details on the superconducting magnets.
1.1 **CERN Accellerators**

The main issues of High Energy Particle (HEP) accelerators are:

- to explore matter at small scale, by means of radiations of wavelength smaller than the dimension to be resolved;
- to produce new, massive particles in high-energy collisions, thanks to the mass-energy equivalence postulated by Einstein;
- to reproduce locally the very high temperatures occurring in stars or in the early universe, and investigate nuclear matter in these extreme conditions, by imparting energy to particles and nuclei;
- to exploit the electromagnetic radiation they emit when accelerated, particularly when the beam trajectory is curved by a magnetic field (centripetal acceleration).

CERN, one of the most important HEP laboratories, is located at Geneva in Switzerland, and it was founded in 1953, following a recommendation of the United Nation Educational, Scientific and Cultural Organization (UNESCO) Meeting in Florence 1950, with the motivation of providing a deeper understanding of the matter and its contents.

After the early stage of the Proton Synchrotron (PS), more advanced accelerator have been developed (Fig. 1.1). The Super Proton Synchrotron (SPS) machine provided the energy to discover the weak force particles $W^+$, $W^-$, and $Z^0$ earning the Nobel prize in 1984 to Carlo Rubbia and Simon Van de Meer [Rubbia, 1985], [Van Der Meer, 1985]. On the way to higher precision, the Large Electron Positron (LEP) collider was built, by providing high accuracy feature values for the aforementioned particles already during start up. In Fig. 1.1, further experiment area, such as the neutrino beam to Gran Sasso (CNGS)\(^1\) and the Antiprotron Decelerator (AD) [Mauri,1997], the first stage on the way to antihydrogen, are also depicted.

\(^{1}\) http://proj-cngs.web.cern.ch/proj-cngs.
The last CERN project is the Large Hadron Collider (LHC): a circular accelerator that will collide proton beams, but also heavier ions up to lead. It is installed in a 27 km long underground tunnel (Fig. 1.2), that already housed the previous accelerator, Large Electron-Positron Collider (LEP) [Fartoukh, 2001]. Four experiments (ATLAS, ALICE, CMS and LHCb) are currently being built, and will be running on the collider; each of them will study particle
collisions under a different point of view, and with different technologies. The
experimental detectors *ATLAS* (A Toroidal LHC Apparatus) Fig. 1.3, *ALICE* (A
Large Ion Collider Experiment) Fig. 1.4, *CMS* (Compact Muon Solenoid) Fig.
1.5 and *LHCb* (Large Hadron Collider beauty) Fig. 1.6.

Figure 1.3: *ATLAS: A large Toroidal LHC Apparatus*

Figure 1.4: *Alice: A Large Ion Collider Experiment at CERN LHC*
A structural drawing is shown in Fig. 1.7. Particles will collide in four points on the ring, corresponding to the so-called insertion points (IP) 1, 2, 5 and 8 in the picture. The injection systems are located at the insertion points 2 and 8.

The radio frequency cavities (RF) can be found at insertion point 4, and the beam dump is installed at insertion point 6. The insertions 3 and 7 house facilities to clean the beam, namely its momentum and orbit. The arcs house the superconducting magnets used to bend and focus the beam.
In a circular accelerator, high kinetic energies are imparted to particle beams by applying electromagnetic fields. A particle of charge $q$ moving through an electromagnetic field is submitted to the Coulomb and Lorentz’s forces expressed by:

$$\vec{F} = \frac{d\vec{p}}{dt} = q \left( \vec{E} + \vec{v} \times \vec{B} \right) \quad (1.1)$$

where $\vec{F}$ is the electromagnetic force exerted by the electric field $\vec{E}$ and the induction field $\vec{B}$ on the particle with velocity $\vec{v}$. Both the electric field and the magnetic field affect the trajectory and the energy of the particle. Therefore, the
The main elements of a particle accelerator are the Radio Frequency (RF) cavities accelerating the particles, the dipole magnets bending them to follow the circular orbit, and the quadrupole magnets focusing them to maintain a proper intensity and size.

The LHC contains 1232 dipole magnets, 360 quadrupole magnets, with two magnetic apertures integrated into a common yoke, and 4 RF cavity modules per beam. Although the LHC circumference is the same of the LEP, it will collide two proton beams at nominal center of mass energy of 14 TeV, i.e. nearly two orders of magnitude higher than in LEP. The use of superconducting magnets and RF cavities permit higher electric and magnetic fields to be achieved, by increasing the maximum beam energy:

\[ E_{\text{beam}} = k \cdot |B| \cdot r \]  

where \( E_{\text{beam}} \) is the beam energy in GeV, \( B \) the magnetic induction field in T, \( r \) the radius of curvature of the machine in m, and \( k \) adimensional constant. The LHC beam energy is \( 10^8 \) times the Lawrence’s first cyclotron one, but with a diameter only \( 10^5 \) times larger.

Superconductivity is a powerful mean to achieve high-energy particle beams and keep compact the design of the machine. Making a machine compact means not only saving capital cost, but also limiting the stored beam energy. According to the equation 1.3

\[ U = 3.34 \cdot E_{\text{beam}} \cdot I_{\text{beam}} \cdot C \]  

where \( U \) is the stored energy per beam in kJ, \( I_{\text{beam}} \) is the current beam in A, and \( C \) is the machine circumference in km, with a particle energy of 7 TeV, a beam current of 0.58 A and a circumference of 26.7 km, the LHC will have an energy
of 362 MJ stored in the beam. This is enough to melt half a ton of copper and thus requires an elaborate and very reliable machine protection and beam dump system [Schmidt, 2004]. In a larger machine, this problem would become even more acute.

Besides capital cost and compactness advantages, superconductivity reduces electrical power consumption. High-energy, high-intensity machines produce beams with MW power, so that conversion efficiency from the grid to the beam must be maximized, by reducing ohmic losses in RF cavities and in electromagnets [Gareyte, 1996]. In d.c. electromagnets, superconductivity suppresses all ohmic losses, thus the only power consumption is related to the associated cryogenic refrigeration.

The rationale is similar for RF cavities, where superconductivity reduces wall resistance and thus increases the $Q$ factor of the resonator, i.e. the ratio between the stored energy $U$ and the power dissipated by the cavity $P_d$ in one cycle at the resonant angular frequency $\omega_0$ [Gareyte, 1996]. However, the wall resistance of superconducting cavities subject to varying fields does not drop to zero, but varies exponentially with the ratio of operating to critical temperature $T_c$ [Gareyte, 1996]. This imposes to operate at a temperature well below $T_c$, in practice as the result of a trade-off between residual dissipation and thermodynamic cost of refrigeration.

Cryogenics plays another fundamental role in nuclear accelerators. In the LHC, the first conducting wall seen by the circulating beams, i.e. the beam screen is coated with 50 $\mu$m of copper and must operate below 20 K, by achieving a resistivity value capable of reducing the beam transverse impedance $Z_T$, directly linked to the rise time of the beam instability [Padamsee, 2004]. Another direct application of cryogenics in accelerators is distributed cryopumping. The saturated vapour pressures of all gases, except helium, vanish at low temperatures, so that the wall of a cold vacuum chamber can act as an efficient cryopump. In fact, it traps gases and vapours by condensing them on a cold surface. Therefore, cryogenics is required for this application independently
of the use of superconductivity.

1.2 LHC superconducting Magnets

The coils of the LHC superconducting magnets are wound with NbTi cables (7000 km in total), working in superfluid helium either at 1.9 K or at 4.5 K. A vertical dipole field $B$ of 8.33 T is required to bend the proton beams, whereas the LHC quadrupole magnets are designed for a gradient of 223 $T m^{-1}$ and a peak field of about 7 T.

1.2.1 LHC Dipole Magnets

The LHC dipole is like a split pair of circular coils, stretched along the particle trajectory in such a way that the dipole field is generated only along the beam pipe, as shown in Fig. 1.8a. The LHC dipoles are based on a compact and cost-saving two-in-one design, where two beam channels with separate coil systems are incorporated within the same magnet [Rossi, 2004]. The main parts of an LHC dipole are depicted in Fig. 1.8b. The superconducting cables of the coils for the LHC magnets are made of NbTi hard superconductor multi-wires, embedded in a copper stabilizer. Such wires are wrapped together to form the so-called Rutherford type cable. The coils are surrounded by the collars which limit the conductor movements [Rossi, 2003].
The iron yoke shields the field so that no magnetic field leaves the magnet. The so-called cold-mass is immersed in a bath of superfluid liquid helium acting as a heat sink. The helium is at atmospheric pressure and is cooled to $1.9 \, \text{K}$ by means of a heat exchanger tube. The cold mass is delimited by the inner wall of the beam pipes on the beam side and by a cylinder on the outside. The iron yoke, the collars, and the cylinder compress the coil by withstanding the Lorentz forces during excitation. The cylinder case improves the structural rigidity and longitudinal support and contains the superfluid helium.
Stability requirements for the beam motion impose stringent constraints to the quality of the magnetic field in the LHC magnets. Owing to the magnets non-ideality, the magnetic field presents multipoles that require corrections to achieve the required beam performance. The major tolerances are specified in [Fartoukh, 2001].

Figure 1.9: Scheme of the LHC cell with main bending dipoles, main focusing quadrupoles, and a full correction scheme.

The LHC arc includes main bending dipoles, main focusing quadrupoles, and a full correction scheme, featuring sextupoles, octupoles and decapoles (Fig. 1.9). Each cell of the LHC arcs has two different types of correction circuits to deal with the sextupole and decapole field errors:

- spool piece corrector magnets, built-in with the main dipole cold masses;
- lattice corrector magnets, mounted in the main arc quadrupole magnets as part of the Short Straight Section (SSS) assembly [Fartoukh, 2001].

Its structure is based on a cost-saving ‘two-in-one’ design, where two beam channels with separate coil systems are incorporated within the same magnet structure. The two coils (physical length of 14.6 m) are fixed by a support structure of laminated collars, which define the exact geometry and provide mechanical stability. The collared coils are integrated into an iron yoke, which serves to increase the central field by about 19 %, and to shield the magnetic field, thus no magnetic field leaves the magnet. Bus bars accommodate the cables to power the magnets of the arcs. They are located in grooves in the iron yoke. The so-called ‘cold mass’ is immersed in a bath of superfluid helium at atmospheric pressure and cooled to 1.9 K by means of a heat exchanger tube, in
which two-phase low-pressure helium is circulated and acts as a heat sink. The cold mass is delimited by the inner wall of the beam pipes on the beam side and by a cylinder on the outside.

In storage rings like the LHC, stable beams have to run as long as possible on the circular orbit (for several hundreds of millions of turns), in order to increase the number of collisions between the counter rotating beams. This imposes strong constrains on the tolerable field perturbations along the trajectory. Deviations from the dipole and quadrupole fields, even if short in both space and time, can induce instabilities reducing the beam life-time. Higher-order multipoles correctors are required to compensate the unavoidable imperfections of dipole and quadrupole magnets. Ideally, a pure n-pole field could be produced by a current flowing along an infinitely thin cylindrical shell, with a cosine like distribution:

\[ I(\theta) = I_0 \cos(n\theta) \] (1.4)

where \( \theta \) is the azimuthal angle.

The LHC dipoles are 15-meters long with a beam aperture of 50 mm in diameter, giving the possibility to consider the coils as infinitely long, and to evaluate the magnetic field in the x-y complex plane by neglecting the z component. In the central part of the dipole taking into account the properties of the analytical functions, it can be postulated that the magnetic field generated \( B \) can be expanded in the complex plane in a power series [Arpaia, 2006].

\[
\mathcal{B}(z) = B_x(x,y) + jB_y(x,y) = \sum_{n=1}^{\infty} \left[ b_n + j a_n \right] \left( \frac{x+jy}{R_{\text{ref}}} \right)^{n-1} = 10^{-4} B_0 \sum_{n=1}^{\infty} \left[ b_n + j a_n \right] \left( \frac{x+jy}{R_{\text{ref}}} \right)^{n-1} \] (1.5)

The reference radius \( R_{\text{ref}} \) is defined to be 17 mm, i.e. approximately two thirds of the magnet aperture radius of 28 mm. \( B_n \) and \( A_n \) are the so called normal and
skew multipole coefficient, also referred to as field components or field harmonics

\[
B_n = C_n \cos(n\alpha_n) \\
A_n = -C_n \sin(n\alpha_n)
\]  

(1.6)

where \(\alpha_n\) is the angle between \(A_n\) and \(B_n\), \(C_n\) is referred as the amplitude of the component of the total field. \(b_n\) and \(a_n\) are their normalized values, expressed standard ‘UNITS’. If \(B_1\) is the main field, \(b_n = B_n/(B_110^{-4})\) and \(a_n = A_n/ (B_110^{-4})\), where the factor \(10^{-4}\) is used for scaling the existence of non-zero \(b_n\) and/or \(a_n\) coefficients reflects the fact that the magnetic field generated by the superconducting coil in a dipole is not a pure dipole and is affected by higher order of multipoles (quadrupole, sextupole, etc.). The multipole components are generated by the difference between the ideal current distribution of Equation 1.4 and the actual current distribution in the coil. Because of the approximation, the field distribution inside de magnet bore is not a pure dipole field, higher components are present (Eq.1.5). All undesired multipole components other than the main field are referred to as “field errors”.

### 1.2.2 LHC Quadrupole Magnets

These magnets are used to focus the beam by squeezing it into a smaller cross-section, a similar effect to a lens focusing light. However, each magnet only focuses the beam in one direction so alternating magnet arrangements are required to produce a fully focused beam [Nogiec, 2006].
Fig. 1.10 shows the cross-section of short straight section with quadrupole cold mass inside cryostat. Two cylinders of elliptical cross section carrying equal and opposite current densities are made to intersect at right angle to each other, $I=I_0\cos(2\theta)$. The overlap region carries no current, and can be treated as the aperture of the magnet. Any point inside this aperture is also inside both the cylinders. Then a pure normal quadripole field presents, according to in equations 1.5 and 1.6, only the component $B_2$, to produce a skew-quadrupole field, the shown arrangement of Fig. 1.11 should be turned by 45°.
The total complex field at any point \((x,y)\) is given by:

\[
\mathbf{B}_{\text{quadrupole}}(x,y) = B_2 \left(\frac{x + jy}{R_{\text{ref}}}\right) = G(x + jy) = B_y(x) + jB_x(y)
\]  

(1.7)

where \(G\) is referred as the field gradient (expressed in \(T/m\)). LHC superconducting quadrupole uses the same cables as the LHC dipole. As the current distribution is only an approximation of the ideal case, also the quadrupole is affected by field errors. Fig. 1.12 shows the cross section of a normal LHC quadrupole, and the field distribution at the current 1185 \(A\).
Chapter 2

2 Magnetic Measurements at CERN

Accelerator magnets are designed and built with stringent specifications on strength, orientation, homogeneity, and position of the null point for the gradient of the magnetic fields. In spite of the great advances in computational techniques for the optimization and performance analysis of a magnet, and given the unavoidable manufacturing and assembly tolerances in the construction process, the above target remains very demanding. Hence, the production of magnets with high field quality has been invariably assisted by a spectrum of various measurements, based on different methods depending on the goal and the accuracy of the desired analysis. At CERN, the Research and Development (R&D) program is based on the upgrade of the measurement techniques in order to analyze dynamic features of the magnets and achieve more accurate magnet
models for the exploitation of the LHC. Considered that the flux induction measurement methods require the integration of the incoming signal, the development of a new digital integrator was launched as a key factor of the R&D program.

In this Chapter, at first an overview of the main methods for magnetic measurements is given by pointing out the instrumentation and the required accuracy.

### 2.1 Methods for magnetic measurements

The most commonly used methods for magnetic measurement in beam-guidance magnets for particle accelerators are:

1. **Fluxmeter method.**
2. **Hall generator.**
3. **Magnetic resonance technique.**

The choice of a measurement method depends on several factors. The field strength, homogeneity and variation in time, as well as the required accuracy, all need to be considered.

**Fluxmeter method**

The *fluxmeter method*, based on the induction law, is the oldest of the currently used methods for magnetic measurements, but it can be very precise. It is also the most accurate method for measuring the direction the magnetic flux lines; this being of particular importance in accelerator magnets. Measurements are performed either by using fixed coils in a dynamic magnet field, or by moving the coils in a static field. Very accuracy can be reached in differential fluxmeter measurements by using a pair of search coils connected in opposition, with one coil moving and the other fixed, thus compensating fluctuations in the magnet excitation current and providing a much higher measurements, but with both coils moving.
The coil method is particularly suited for measurements with long coils in beam-guidance magnets, where the accurate measurement of the field integral along the particle trajectory is the main concern. With the advent of modern digital integrators and angular encoders, harmonic coil measurements have improved considerably and are now considered as the best choice for most types of accelerator magnets. The method provides the additional advantage of simultaneous measurement of strength, quality, and geometry. A compensating coil, connected to in series and rotated with the main coil, may be used to suppress the main field component and thus increase the sensitivity of the system for measurements of field quality.

Dynamic fields are measured with static coil linking to selected harmonics. Another induction measurement consists of moving a stretched wire in the magnetic field, thus integrating the flux cut by wire. It also possible to measure the flux change while varying the field and keeping the wire in a fixed position. Tungsten is often selected, if the wire cannot be placed in a vertical position. The accuracy is determined by mechanical positioning of the wire. Sensitivity is limited, but can be improved by using a multi-wire array. This method is well suited to geometry measurements, to absolute calibration of quadrupole fields and in particular to measurements in strong magnets with very small aperture.

The choice of geometry and methods depends on the useful aperture magnet. The sensitivity of the fluxmeter method depends on the coil surface and the quality of integrator. The coil integrator assembly can be calibrated to an accuracy of a few tens of ppm in a homogeneous magnetic field by reference to a nuclear magnetic resonant probe, but care must be taken not to introduce thermal voltages. The main advantage of search coil techniques is the possibility of a very flexible design of coil. The high stability of the effective coil surface is another asset. The linearity and the wide dynamic range also play an important role. The technique can be easily adapted to measurements at cryogenic temperatures. After calibration of the coils at liquid nitrogen temperature, only a minor correction has to be applied for use at lower temperatures. One the other
hand, the need for relatively large induction coils and their related mechanical apparatus which is often complex, may be a disadvantage. Furthermore, the measurements in static fields are relatively slow.

**Hall generator method**

The *Hall generator method* is based on the Hall’s effect. A metal strip immersed in a transverse magnetic field and carrying a current developed a voltage mutually at right angles to the current and field that opposed the Lorentz force on the electrons. The Hall-generator provides an instant measurement, uses very simple electronic measurement equipment and offers a compact probe, suitable for point measurements. The probes can be mounted on relatively light positioning gear. Considerable measurement time may be gained by mounting Hall generators in modular multi-probe arrays and applying multiplexed voltage measurement. The wide dynamic range and the possibility of static cooperation are other attractive features. However several factors set limits on the obtainable accuracy. The most serious is the temperature coefficient of the Hall voltage.

Temperature stabilization is usually employed in order to overcome this problem, but increase the size of probe assembly. The temperature coefficient may also be taken into account in the probe calibration by monitoring the temperature during measurements. Last but not least is the problem of the non-linearity of the calibration curve, since the Hall coefficient is a function of the field level. The measurement of the Hall voltage sets a limit about 20 $\mu T$ on the sensitivity and resolution of the measurement, if conventional direct current excitation is applied to the probe. The sensitivity can be improved considerably by application of *ac* excitation. In the following, the main two measurement techniques currently employed at CERN for field harmonic analysis, based on rotating coils and Hall plate probes, are described.
Magnetic resonance techniques

The nuclear magnetic resonance technique is considered as the primary standard for calibration. It is frequently used, not only for calibration purposes, but also for high accuracy field mapping. The method was first used in 1938 for measurements of the nuclear magnetic moment in molecular beams [Kusch, 1939].

A few years later, the phenomenon was observed in solids by two independent research teams [Purcell, 1946], [Bloch, 1946]. Based on an easy and accurate frequency measurement, it is independent of temperature variations. Commercially-available instruments measure fields in the range from 0.011 T up to 13 T with accuracy better than ±10 ppm.

In practice, a sample of water is placed inside an excitation coil, powered from a radiofrequency oscillator. The precession frequency of the nuclei in the sample is measured either as nuclear induction (coupling into a detecting coil) or as resonance absorption [Bloembergen, 1948]. The measured frequency is directly proportional to the strength of the magnetic field with coefficients of $42.57640 \text{ MHz/T}$ for protons and $6.53569 \text{ MHz/T}$ for deuterons. The advantages of the method are its very high accuracy, its linearity, and the static operation of the system. The main disadvantage is the need for a rather homogeneous field in order to obtain a sufficiently coherent signal.

Pulsed NMR measurements have been practiced for various purposes even at cryogenic temperatures [Putlitz, 1996]. Electron paramagnetic resonance (EPR) and electron spin resonance (ESR) can be viewed as two alternative names in a family of electron magnetic resonance (EMR) techniques. ESR is a related and accurate method for measuring weak fields [Kernevez, 1992]. It is now commercially available in the range from 0.55 mT to 3.2 mT, with a reproducibility of ±1 ppm and is a promising tool in geology applications.
2.1.1  Rotating coils

The principle of the measurement is based on Lenz’s law: when a conductor loop moves with respect to a magnetic field, a flux variation occurs and a voltage is induced proportional to the time variation of the flux. In Fig. 2.1 a cylindrical surface parallel to the axis of the magnet $z_m$ and uniform in the axial direction is considered. $\Gamma$ designates the arc at the intersection between $\Sigma$ and the $xy$ plane $z_1$ and $z_2$ determine the positions of the ends in the complex plane. The magnetic flux $\Phi$ through this surface is defined by:

$$\phi = \iint_{\Sigma} \vec{B} d\vec{\sigma}$$  \hspace{1cm} (2.1)

with $d\vec{\sigma}$ the surface element vector.

Since the surface is parallel to the axis of the magnet, and since $\vec{B}$ and $\Sigma$ are uniform along the magnet’s axis:

$$\phi = L \int_{\Gamma} \vec{B}(\vec{z}_m \times d\vec{\gamma})$$  \hspace{1cm} (2.2)

with $L$ the length of the surface along the $z_m$ axis and $d\vec{\gamma}$ the arc element vector.
Figure 2.1: Magnetics flux through a cylindrical surface

Now the coordinates of $d\gamma$ are set to $(dx, dy, 0)$. The coordinates of $(\bar{z}_m \times d\gamma)$ are $(-dy, dx, 0)$. Using the two dimensional expression of the $\bar{B}$ the flux is given by:

$$\phi = l \text{Re} \left[ \int_{\Gamma} B_y dx - B_z dy \right]$$  \hspace{1cm} (2.3)

In complex notation it is expressed by:

$$\phi = L \text{Re} \left[ \int_{z_1}^{z_2} B(z) dz \right]$$  \hspace{1cm} (2.4)

Introducing the definition of the complex potential [Devred, 1998] and using the multipoles expansion series, the flux is expressed in terms of field harmonics as:
\[ \phi = L \text{Re} \left[ \sum_{n=1}^{\infty} \frac{1}{n} C_n \frac{z_2^n - z_1^n}{R_{\text{ref}}^{n-1}} \right] \] (2.5)

Now it is assumed that the surface \( \Sigma \) represents the surface for all turns of a pick up coil rotating around the axis \( z_m \) (i.e. the windings are infinitely thin). The angle \( \theta' \) describes a rotation of the surface around the axis \( z_m \), \( z_2 \) and \( z_1 \) are the positions of the extremities of the arc \( \Gamma \) at \( \theta' = 0 \). So for any angle \( \theta' \) the location of the ends \( z_{1\theta} \) and \( z_{2\theta} \) is described by:

\[ z_{1\theta} = z_1 \exp(i\theta') \quad \text{and} \quad z_{2\theta} = z_2 \exp(i\theta') \] (2.6)

Using the equation (2.5) and (2.6) the flux \( \Phi \) seen by a rotating coil is:

\[ \phi(\theta') = \text{Re} \left[ \sum_{n=1}^{\infty} K_n C_n \exp(in\theta') \right] \] (2.7)

with \( K_n \) the coil’s sensitivity to the \( n \)th multipole:

\[ K_n = \left( \frac{NwL R_{\text{ref}}}{n} \right) \left[ \left( \frac{z_2}{R_{\text{ref}}} \right)^n - \left( \frac{z_1}{R_{\text{ref}}} \right)^n \right] \] (2.8)

In the equation (2.8) \( Nw \) represents the number of the coil turns, so that \( K_n \) only depends on the coil geometry.

The voltage induced by a flux change is given by Faraday’s law:

\[ V = \frac{d\phi}{dt}, \] (2.9)
A change of flux inside the coil is achieved either by varying the magnetic field (i.e. varying the magnet current) or by rotating the coil inside the magnetic field. Here the second method, called rotating coil method, is described. The angular dependence of the flux on the angular position of the coil is shown in equation (2.7). In the following the magnetic field is considered to be independent from time, so that the field harmonics $C_n$ are assumed constant. Faraday’s law gives the voltage versus time. To calculate the multipoles $C_n$ the flux versus angle is needed. Therefore the measurement is performed in the following way:

- the coil is turned by a motor;
- the voltage induced in the coil is fed to an integrator;
- the integrator is read out by a controller;
- an angular encoder triggers this readout to ensure equidistant readouts.

This is needed by the *standard analysis* which is based on a Fourier transform. In the following this procedure is described mathematically. It is assumed that the $Nw$ turn pick up coil is rotating around the $z$-axis with angular velocity $\dot{\theta}(t)$. Then the angle $\theta'$ at a given time $t$ equals $\theta(t)$ and the angular speed equals its first derivative:

$$\theta' = \theta(t) \quad \text{and} \quad \frac{d\theta(t)}{dt} = \dot{\theta}(t) \quad (2.10 \ a)$$

In the ideal case

$$\theta' = \omega \cdot t \quad \text{and} \quad \frac{d\theta(t)}{dt} = \omega \quad (2.10 \ b)$$

with $\omega$ the ideal (i.e. constant) angular velocity. Faraday’s law (2.9) applied to equation (2.7) gives:

$$V(t) = -\dot{\theta}(t) \Re \left[ \sum_{n=1}^{\infty} nKnC_n \exp(i\theta(t)) \right] \quad (2.11)$$

The voltage is then integrated using an integrator:
\[ \phi(t) = - \int_0^t V(t') dt', \quad (2.12) \]

assuming that the integration starts at \( t = 0 \).

The angular encoder triggers the readout of the integrator to ensure equally spaced angular steps. Since \( \theta(t) \) gives the position of the coil versus time, its inverse function \( t = \theta^{-1}(\theta') \) describes the time at which an angle was reached. Thus the flux \( \Phi_i \) given by the integrator for an angular interval \( \theta_i - \theta'_i \) as:

\[ \phi_i = - \int_{\theta_i}^{\theta'_i} V(t') dt \quad (2.13) \]

\( \theta_0 \) is the angle at which the integration started and \( \theta'_i \)

\[ \theta_i = \frac{2\pi i}{P}, \quad i = 1, \ldots, P \quad (2.14) \]

with \( P \) the number of readings per revolution. The flux \( \Phi_i \) can be further written as

\[ \phi_i = - \int_{\theta_0}^{\theta_i} V(t) dt = \int_{\theta_0}^{\theta_i} \frac{1}{\dot{\theta}} V(\theta) d\theta \quad (2.15) \]

\( \Phi_i \) corresponds to the value of the integral at \( t_i \). Comparing the last term of the above statement to Equation (2.12) it is evident that \( \Phi_i \) is speed independent. A discrete Fourier transform is applied to the total readout \( \Phi = \{ \Phi_i | i = 1 \ldots P \} \) of the integrator with \( \psi \) the spectrum of the flux and DFT the discrete Fourier transform. It can be showed [Bottura, 1997] that the multipoles \( C_n \) are given by:
\[ C_n = \frac{1}{Kn \psi n} \]  \hspace{1cm} (2.16)

### 2.1.2 Hall probes

A Hall probe is a semiconductor-based detector which uses the Hall effect to allow the strength of a magnetic field to be measured. The Hall Effect is seen when a conductor is passed through a uniform magnetic field. The natural electron drift of the charge carriers causes the magnetic field to apply a Lorentz force (the force exerted on a charged particle in an electromagnetic field) to these charge carriers. The result is what is seen as a charge separation, with a build up of either positive or negative charges on the bottom or on the top of the plate.

![Figure 2.2: The rotating coils shaft](image)

Figure 2.2: The rotating coils shaft
Figure 2.3: Cross section of the shaft for rotating coil measurements with pick-up coil in ‘tangential’ configuration.

Hall Plane Probes at CERN

During long periods of constant current supplying, all components of the magnetic field show decay behaviour. The decay is especially pronounced at the low level of the magnetic field during injection (about 0.54 T), where the persistent current magnetization is relatively large and has a significant impact on the field. The field components return to the original hysteresis curve as soon as the current ramp restarts, i.e. they ‘snapback’. An example for $b_3$ during decay and snapback for a dipole is shown in Fig. 2.4 a) and b), as a function of both the time and the main field, respectively. The snapback during the acceleration ramp after the end of injection only lasts a few of seconds, and rotating coils used so far for measurements do not have the time resolution to accurately measure its time dependence. For this main reason, a system based on the Hall plates with a higher acquisition rate was developed [Bottura, 2000], [Berkes, 1998].
Figure 2.4: The dipole field $B_1$ and the normal sextupole component $b_3$ are shown as a function of time. The injection field is reached at a time $t = 0$. The sextupole component decays during injection. After about 1000 s the magnet is ramped again. The snapback is clearly visible a). The same measurement of $b_3$ is shown as a function of the dipole field along the up-ramp branch of the hysteresis curve. The decay and snapback are indicated b).

An arrangement of $m$ Hall plates, equally spaced on the circumference of a ring and radially oriented, allows all the field components with an order lower than $m$ to be suppressed. The measured signal for the field component of order $m$ can be maximized if all the Hall plates are placed in the poles of the $2m$-pole field. Fig. 2.4 a) shows an arrangement of 3 Hall plates in a dipolar and a sextupolar field. The projections of the field onto the normal vectors of the three plates are measured. The sum signal of the three plates is compensated for the dipole field and proportional to the sextupole component for the sextupole field.

An expression for the sum signal $S$ of a group of $m$ plates with equal sensitivities in a magnetic field with normal and skew multipole components $B_k$ and $A_k$, respectively, is given by [Breschi, 2000]:

\[
S = \sum_{k=1}^{\infty} \frac{R}{R_{\text{ref}}}^{(2k-1)m-1} (-1)^{\frac{(2k-1)m-1}{2}} mB_{m(2k-1)} + \sum_{k=1}^{\infty} (-1)^{mk} \left( \frac{R}{R_{\text{ref}}} \right)^{2km} mA_{2km} \tag{2.17}
\]

$R$ is the radial distance of the active area in the Hall plates from the center of the ring.
In an ideal case, where all the Hall plates are well aligned and have equal sensitivities, the only multi-poles contributing to the total signal $S$ are the normal odd and the skew even multiples of order $m$.

In an arrangement of three plates, the sum signal $S$ is compensated for the dipole, and only normal multipoles of order $3(2k-1)$ (i.e. $B_3$, $B_9$, $B_{15}$,…) and skew harmonics of order $6k$ (i.e. $A_6$, $A_{12}$, $A_{18}$,…) contribute according to:

$$S = \sum_{k=1}^{\infty} 3 \left( \frac{R}{R_{\text{ref}}} \right)^{3(2k-1)-1} B_{3(2k-1)} + \sum_{k=1}^{\infty} (-1)^k 3 \left( \frac{R}{R_{\text{ref}}} \right)^{6k-1} A_{6k}$$

(2.18)

In the case of a pure sextupole field, this yields:

$$S = 3 \left( \frac{R}{R_{\text{ref}}} \right)^2 B_3$$

(2.19)

The sum signal of a decapole arrangement with five plates ($m = 5$) is:
\[ S = \sum_{k=1}^{\infty} 5 \left( \frac{R}{R_{\text{ref}}} \right)^{5(2k-1)-1} (-1)^{\frac{(2k-1)5-1}{2}} B_{5(2k-1)} + \sum_{k=1}^{\infty} (-1)^{k} 5 \left( \frac{R}{R_{\text{ref}}} \right)^{10k} A_{10k} \] (2.20)

In the case of a pure decapole field, this yields:

\[ S = \sum_{k=1}^{\infty} 5 \left( \frac{R}{R_{\text{ref}}} \right)^{4} B_{5} \] (2.21)

Examples for decapole rings are sketched in Fig. 2.5.

### 2.1.3 Stretched wire

The stretched-wire technique is also based on the induction method [DiMarco, 1996], [DiMarco, 2000]. A thin wire, with a diameter of 0.1 mm, is stretched in the magnet bore between two precision stages. A motion results in a voltage at the two ends of the wire, whose integral is the magnetic flux through the area scanned by the motion. The method, a robust null technique with very high resolution, provides a measurement of the integral field, of the field direction, and of the magnetic axis. The uncertainty depends on the accuracy of the precision stages driving the wire motion (±1 μm), on the effectiveness of the sag correction, and on the alignment errors during installation. The overall uncertainty on the integrated strength and on the angle measurement was estimated at ±5 units and ±0.3 mrad, respectively [DiMarco, 2000]. The wire used is thin and its handling is quite difficult. Further on, the wire must be free of dirt because it often has magnetic properties, and the magnetic field acting on it will deviate the wire from its ideal position by generating a fake result. In spite of the practical difficulties, this is a very powerful technique.
2.2 Instrumentation for Magnetic Measurements

In the following sections, we give an overview about the principal devices used for magnetic measurement at CERN.

2.2.1 Rotating Coil system at CERN

Devised since 1954 [Elmore, 1954], [Dayton, 1954], the rotating coil method is now widely used for magnets with cylindrical bore owing to its capability at measuring all properties of the magnetic field (field strength, multipoles, angle, direction) integrated over the coil length. An induction coil is placed on a circular support and is rotated in the field to be mapped [Bottura, 1998]. The coil angular position is measured by an angular encoder, rigidly connected to the rotating support. The coil rotating in the field cuts the flux lines and a voltage is induced at the terminals. The voltage is integrated between predefined angles obtaining the flux change as a function of angular position. If the measured field is 2-D in the cross section of the magnet, with negligible variation along the magnet length, it can be shown [Jain, 1998] that a Fourier analysis of the angular dependence of the measured flux leads naturally to coefficients directly proportional to the so-called multipole coefficients of the field [Beth, 1966]. In turn, the multipole coefficients of the field can be related directly to linear and non-linear accelerator beam properties, thus explaining the wide acceptance of the rotating coil method for mapping accelerator magnets.

This method eliminates the time dependence [Bottura, 2004], and, in particular, the influence of variations of the rotation speed, greatly relaxing requirements for uniform rotation. Differential measurements are also beneficial to increase the resolution of high-order multipoles, several orders of magnitude smaller than the main field. This is realized by using a set of compensation coils mounted on the rotation support [Bidon, 1995]. The signal from the compensation coils is used to suppress analogically the strong contribution from
the main field. The compensated signal is analyzed in Fourier series together with the absolute signal of the outermost rotating coil in order to obtain the main field, as well as the higher order multipoles. The overall uncertainty on the integral field strength and on the harmonics depends on the shaft type so far used at CERN, and is not greater than few units [Pérez, 2006], [Delsolaro, 2001], [Billan, 2000]. The Twin Rotating Unit (TRU) and the new Micro Rotating Unit Rotating coils ($\mu$RU) system have been developed continuously at CERN. In the following, a description of the latest development, the Micro Rotating Unit ($\mu$RU), compared to the system used for the series measurements of the LHC magnets, the Twin Rotating Unit (TRU), is given. The rotating coil system utilized at CERN for the dipoles is based on a Twin Rotating Unit (TRU) [Billan, 2000].

For the usual measurements on constant current dipoles and quadrupoles this time duration is considered acceptable. However, to fully analyze fast field transients [Bottura, 2000], a new Micro Rotating Unit ($\mu$RU) was designed to turn faster and provide harmonic measurements at rates in the range from 1 to 10 Hz. Such a system was developed in the framework of the project Fast Magnetic measurement Equipment (FAME). Fast measurements require that the coils rotate continuously in one direction and at higher speeds [Brooks, 2007].

**TRU**

The current rotating coil system utilized at CERN is based on a Twin Rotating Unit (TRU). This system consists of a motor unit that rotates a 16 meter long shaft composed of 13 coil-carrying hollow ceramic segments connected in series using flexible titanium bellows. For measurements of dipole magnets, each ceramic segment has 3 separate coils of wire mounted within it, 1 central coil and 2 tangential coils. The central coil is located along the central axis of the segment, while the tangential coils lie directly opposite of one another on the circumference of the segment. These coils cover the length of the segment and
lie parallel to one another. The nominal rotation speed is 1Hz with variations smaller than 3%. The acquisition software remotely controls the operation of the unit. An angular encoder gives the angular position of the shaft with 4096 counts per revolution plus a “zero” pulse on a separate channel. The encoder housing is rigidly connected to an electronic inclinometer, giving an absolute reference for the orientation of the encoder “zero”. Furthermore the TRU side of the shaft is provided with a reference surface, aligned with the reference surface on the coil shaft. Each measurement cycle consists of three turns in alternating direction. The first turn is for accelerating the shaft in order to get the right constant rotation speed. The read-out is executed during the second turn with constant rotation speed. The last turn is for decelerating the shaft so as to change the rotation direction. This mode is called washing machine mode Fig. 2.6. The final measurement results are obtained from the average of the forward and backward revolutions.

Figure 2.6: The TRU unit

μRU

The μRU-system Fig. 2.7, based on a modified version of the long ceramic coil shafts with 12 dipole-compensated coil sectors (1/4 of the turns of a standard system), better mass balancing, and sturdier connectors, is capable to turn
continuously in one direction up to 8 Hz thanks to 54-channel slip rings. The \( \mu RU \) attaches directly to the anticryostat and replaces the previous bulky TRU (Fig. 2.8). The available coils are connected in series arbitrarily by means of a patch panel. This permits changes in the compensation schemes or combination of several coils in virtual supersectors, used to measure the integral field.

Figure 2.7: Motor for rotating coil in a long dipole magnet in the SM18 laboratory
The signals induced into the rotating coil are split in an “absolute” and a “compensated” signal. The dipole field is derived from the absolute voltage signal $U_A$ of the coil $A$ only. In order to measure higher multipole field components and to compensate the signal for the disturbing contribution of the dipole field, the two pick-up coils (A and B) are electrically connected with opposite polarities (array of two coil).

In both, radial and tangential arrangements, the pick-up coils A and B are parallel and, thus, always have the same angle with respect to the dipole field. For this reason, the contribution of the dipolar field component $B_1$ to the compensated signal $U_{\text{comp}} = U_A - U_B$ vanishes, and only field components of order $n > 1$ contribute to the signal.
CHAPTER 2

Voltage signals from the rotating coils are first pre-amplified and then read-out simultaneously by a set of digital integrators. A schematic drawing of the circuit is shown in Fig.2.9. An angular encoder is connected to the shaft. Since the time integration is triggered by pulses from the angular decoder, the signals are after all sampled as a function of the rotation angle $\theta_k$ in a discrete series of $k$ points for a total of $M$ points uniformly distributed over a full revolution. A software on a workstation controls the integrators, the motor rotating the shaft and the magnet power supply. For every angle $\theta_k$, the magnetic flux $\Phi_k$ through the pick-up coils is obtained as a cumulative sum over the flux increments $\Delta \Phi_i$

$$\Phi_k = \sum_{i=1}^{k} \Delta \Phi_i.$$

### 2.2.2 Digital Integrators

A magnetic flux measurement by means of the rotating coils technique requires the integration of the voltage induced on the coil; therefore digital integrators are an important part of the instrumentation for magnetic measurements. Digital integrators currently used in the most important research centers are:

**PDI (Portable Digital Integrator)**

![PDI diagram](image)

Figure 2.10: Principle of PDI based on Voltage to Frequency conversion

Digital integrators have been the basic electronic tool for magnetic measurements at CERN since the 80’s. The CERN Portable Digital Integrator
(PDI) has been in use for over 20 years [Elmore, 1954]. In this integrator, the voltage from the induction coil $V_{in}$ is sent, after conditioning and amplification, to a Voltage-to-Frequency Converter (VFC), whose output is a square signal with frequency $f$ proportional to the VFC input voltage Fig. 2.10. This signal is then entered in a counter that accumulates the number $n$ of square pulses during a measurement period $dt$ starting at tstart and ending at tend. The frequency $f$ of the square signal is equal to the time derivative of the number of pulses $(dn/dt)$ and the output of the counter is, apart for the amplifier gain $g$ and a proportionality constant $KVFC$, a digital measurement of the integral of the input voltage. The digital integrator achieves high accuracy owing to the conversion to frequency domain. The limiting elements in this concept are the stability and linearity of the VFC, and the resolution of counting operation that depends on the maximum operation frequency of the VFC. Hybrid technology VFC’s have linearity and stability better than a few ppm over the whole range of input voltage. The typical maximum frequency of operation is 1 MHz. In order to make the circuit practical, some additional features are added to the basic scheme described above: 1. Commercial VFC circuits work only with single polarity voltage, e.g. 0 to 10 V, while the signal from an induction coil can have both polarities. The dual polarity capability is restored by shifting the input voltage by a precise and stable reference $V_{ref}$ whose effect is to place the input zero exactly in the middle of the VFC range. This offset is then eliminated after counting, subtracting the counts from a reference source $f_{ref}$ oscillating at exactly half of the maximum frequency of the VFC.
**FDI (Fast Digital Integrator)**

![Figure 2.11: Principle of FDI (Fast Digital Integrator)](image)

The Fast Digital Integrator (FDI) was developed to overcome the limitations of the PDI, providing a more advanced and performing solution with respect to the other integrators previously described; it represents the new state-of-art solution. The block diagram of the FDI is shown in Fig. 2.11 [Dayton, 1954]. The basic principle consists in the immediate integration of the input signal $V_{in}$ in the digital domain, without previous analog processing, in order to reduce the impact of analog uncertainty sources.

The input stage is represented by a gain programmable amplifier (PGA), with automatic gain and offset calibration and adjustment. The gain and the voltage offset are controlled by a Field Programmable Gate Array (FPGA) performing the calibration, storing the calibration coefficients and applying them in measurements. The input signal is digitized by an Analog-to-Digital Converter (ADC), with $N_{\text{resolution}}$ numbers of bit and a sampling rate equal to $f_{\text{sampling}}$.

The signal just acquired and converted becomes the input of a Digital Signal Processor (DSP) performing numerical integration when triggered from an external digital signal (e.g. pulses coming from an angular encoder). The DSP manages the analog and digital I/O of the instrument through the FPGA which
plays as an I/O processor. At last the result of the integration F is made available on a digital communication bus in order to be sent to an external device as a PC.
Chapter 3

3 Flexible Framework For Magnetic Measurements – FFMM

In this chapter FFMM basic principles are discussed with the architectural solution and design choices made in order to achieve above mentioned goals. The development of a software easily adaptable or extendable to include new applications, and satisfying a wide range of measurement requirements, was the aim of the design and implementation of the Flexible Framework For Magnetic Measurements platform (FFMM), a new version of the CERN of acquisition and control software [Arpaia, 2006].
3.1 **FFMM concepts**

The FFMM is a software framework for magnetic measurement applications based on Object Oriented Programming (*OOP*), and Aspect-Oriented Programming (*AOP*) [Lieberher, 1989]. In particular, FFMM aims at supporting the user in developing software maximizing quality in terms of flexibility, reusability, maintainability, and portability, without neglecting efficiency, vital in test applications. Moreover, the requirements for a wide range of magnetic measurement applications, as required for the test of superconductive magnets for particle accelerators, have to be satisfied.

FFMM can be regarded as a set of rules allowing the user to easily create high-quality software in the field of magnet testing. On the other hand, the produced measurement software is not flexible, since it can be only reconfigured within the boundaries of a specific measurement application. The user defines to which extent the measurement software has to be reconfigurable. The realization of the framework goals is based on the following basic ideas:

1. A group of interfaces and abstract classes represents a white-box layer defining the high-level structure of FFMM used to generate new parts of the framework. This allows potentiality and flexibility of FFMM to be extended. The flexibility is achieved by means of reusability of the code: rapid variations of measurement requirements due to the frequent occurrence of different small batches of tests are satisfied by redesigning software by reusing modules.

2. A group of modules, already available to the test engineer (end user), represents a black-box layer, allowing both module reusability and use easiness to be achieved, even by test engineers without deep knowledge of internal FFMM mechanisms. Reusability is achieved by object-oriented approach and modularity: a suitable design of the code allows modules to be reused.
3. Aspect-Oriented Programming (AOP) improves the reusability and the maintainability of FFMM: in large projects, several concepts are transversal to many modules (cross-cutting concerns). They are extrapolated from the native units and implemented in separated modules (aspects), in order to improve the system modularity (maintainability enhancement). Incremental building of module libraries: once modules can be reused, a finite application domain will be saturated in a finite time.

4. A suitable definition of the code structure (normalization of structures and software modules) gives rise to standard modules, representing the basic library for the realization of new components and the extension of already existing ones. Standardization of software structure and modules: a definition of code structure and patterns gives rise to the production of standard modules to be reused easily.

5. A library of reusable modules is built incrementally during the start-up of the framework up to a “saturation” condition inside an application domain, allowing further requirements in the same domain to be satisfied by a limited effort. Predefinition of a software structure of the test program, organized in standard modules: such an organization provides the user about templates to be filled for generating new codes.

3.2 FFMM Architecture

On the basis of the above ideas, in Fig. 3.1 is shown the FFMM architecture.
The test engineer (end user) produces a description of the measurement application, *User Script*, whose semantic and syntactic correctness is verified by the *Script Checker*. Then, from the *User Script*, the *Builder* assembles the *Measurement Program*, according to the architecture of the *Scheme*, by picking up suitable modules from the *Software Module Library*. If some modules are not available in the library, a template is provided to the user (administrator user) in order to implement them according to a suitable predisposed structure. Once debugged and tested, the *Measurement Program* will be stored in the *Database* in order to be reused. According to the analysis of typical use-case tests on superconductive magnets, the generic *User Script* is organized into the following phases:

- definition of the measurement components;
- specification of mechanical and electrical connections;
- definition of dynamic parameters, i.e. configurable during run-time of the *Measurement Program*;
- component checking;
- configuration of measurement devices;
- description of the measurement procedure;
• preliminary data analysis;
• data saving.

The TestManager organizes the test by knowing the Unit Under Test, the Quantity to be measured, the measurement configuration, and the measurement procedure. TestManager has an association with the Devices (software representation of the measurement devices). Among Devices, Virtual Devices can be controlled remotely by PC through a Communication Bus [Arpaia, 2006].

The Synchronizer and the FaultDetector are units managing critical topics in a measurement application. The Synchronizer manages the software temporization in the measurement procedure, while the FaultDetector intercepts malfunctions and errors. The Synchronizer manages the software temporization in the measurement procedure, while the FaultDetector intercepts malfunctions and errors. The Synchronizer and the FaultDetector can be considered cross-cutting concerns, because they are transversal to many software modules. As a matter of fact, the synchronization policy involves all the measurement devices and all the test procedures. Furthermore, the fault detection is a fundamental part of all the devices, as well as of the measurement system as a whole. Then, the Synchronizer and the FaultDetector are encapsulated in Aspects according to AOP approach. Therefore, the policy for managing synchronization actions and faults can be extrapolated from the single modules and handled separately. In this way, further modifications will affect only those two components, without any need for code changes in all the modules related to the fault detection or to the synchronization. The Logger class handles the stock up of configuration and measurement data, as well as system warnings and exceptions (Appendix D).
CHAPTER 3

3.3  **FFMM Design**

There are some key requirements and system constraints that have a significant bearing on the architecture. The following kinds of constraints are identified:

- Portability
- Distribution
- Reuse
- Use of off-the-shelf products

### 3.3.1  Portability:

A key requirement for the FFMM system was the portability across the following platform:

- GNU/Linux Kernel 2.4 and 2.6
- Microsoft Windows Win32 Platform

The FFMM was hence designed from start with portability in mind; for this reason an isolation layer abstracting the OS platform detail for basic services has been defined and implemented with either ad-hoc solutions or third part cross-platform libraries.

FFMM components need to access different communication media on all platforms; in particular:

- RS232 and GPIB
- Ethernet 10/100 Mbit
- PLX
- WorldFIP
- FFMM needs multithreading support on all platforms.
- Platform-specific I/O functionalities should be used for each platform to improve performances.
CHAPTER 3

To decouple communication needs of FFMM components a forwarding/receiving design pattern has been implemented. In particular objects of CommunicationBus hierarchy act as forwarder/receiver and can be encapsulated by those components that need to communicate across process boundary.

3.3.2 Distribution:

The FFMM framework has been designed to build measurement application that are local to a measurement node connected with all virtualized measurement devices that are involved in a test session. The measurement node, hosting the FFMM application instance, is then responsible for the execution of the measurement script and the coordination of all distributed hardware devices directly or connected to the node itself. From this point of view, the FFMM itself is not distributed on several nodes: it acts as a coordinator for the hardware devices that are needed to carry on specified measurement sessions.

3.3.3 Reuse

In order to maximize reuse, a white-box layer that lets users and developers to extend framework by means of inheritance. The user of framework can re-define/extend behavior by adding new classes that inherits from the abstract ones of FFMM. FFMM also provide a black-box layer (made by using the white-box one) that can be used directly without any internal knowledge of the framework.

3.3.4 External libraries and Off-the-shelf products

The following libraries were used to design and implement the FFMM system:

- WxCTB 0.9
- GPIB Drivers (for both Linux and Windows)
- PLX API and Drivers (for both Linux and Windows)
- NIDAQmx
- Poco (thread, logging and event infrastructure)

All these components are available for multiple platforms and in particular for Win32 and Linux (on both 2.4 and 2.6 kernels).

PLX and GPIB drivers are only available for Win32 and Linux platforms: this puts a constraint on portability on other platforms other than those two.

### 3.4 Components

#### 3.4.1 Logger

In order to be effective, loggers need to be simple for programmers to use. Programmers aren't going to frequently use something that is inconvenient. The user should be able to emit a log message with something no complicated. On the other hand the logger needs to gather all of this peripheral information together, format it into a log message, and then add it to the growing list of logged messages. Moreover another question the logger architecture must answer is mainly: where should the logged messages be stored? Data could be stored in a text or binary file or in a database table. The possibility to accumulate it in RAM and that is constraints are satisfied might even take into account. The choices are endless. However, the final destination of the logged messages has to be kept decoupled with the format of the messages themselves. There are indeed two different responsibilities: logged message formatting, and logged message recording. These are both in the flow of logging a message, but both can vary independently of each other. The formatter does not care where the message is recorded, and the recorder does not care about the format of the message. Whenever there are two connected but independent algorithms and the Strategy pattern can be used to connect them.
3.4.2 Virtual device

Virtual Devices are software components modeling in FFMM the concrete devices that can be orchestrated during measurement processes. While the VirtualDevice interface defines a role, in FFMM a hierarchy of device has been defined and evolved during the development iteration through feedbacks gathered during several meetings with the measurement team.

Virtual Devices implementations are designed as singletons with a strict control on the number of instances. A single device registry is kept in the MeasurementDevice abstract class in order to provide access to devices in every context of the user script by using symbolic identifiers. VirtualDevice class is involved in event handling and provides a basic interface to create/destroy devices using named identifier. This functionality is very important since let script developers to bind symbolic names to devices and, by means of such names, obtain, in every context, a reference to specific devices.

3.4.3 Event Handling

A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. Achieving consistency by making the classes tightly coupled reduces their reusability.

For example a logger can be interested to the end of an acquisition from a FDI. Both classes can be reused separately, but can work together too. The logger and FDI can depict the same data using different presentation. They don’t know about each other, but when the data are changed both reflect the changes immediately. This behavior implies that the logger and FDI are dependent on the data object and therefore should be notified of any change in its state. And there's no reason to limit the number of dependent objects to two; there may be any number of different user interfaces to the same data. The Observer pattern describes how to establish these relationships. For this reason, to increase the
system flexibility the behaviors of devices are collected in homogeneous groups representing behavioral interfaces modeling the devices functionalities. During the analysis of instrumentation, if a characteristic behavior is detected, to provide it to the user, the class instrument has to implement the relative interfaces. So in a modular way, if studying a device the necessity of include a new functionality emerges, the interfaces describing this behavior have to be implemented. The benefit is that the interface of a device is modified modular using existing interfaces and avoiding modification to the class hierarchy that could involve the complete framework structure.

### 3.4.4 Fault Detector

The AOP-based architecture for fault self-detection in measurement systems is based on:

- a fault detection subsystem, designed for:
  - monitoring the ‘health’ state of the measurement system's component devices;
  - catching software faults such as stack overflow, live-lock, deadlock, and application-defined faults as they occur.

- a fault notification subsystem, responsible for:
  - constantly receiving the sequence of faults occurring from all the system components;
  - storing the diagnostic history and providing means to other components or to external humans to access it and adequately react to faulty events.

In the architecture, several kinds of classes of faults relevant in automatic measurement systems are identified: faults in virtual device, faults in the measurement environment and faults in software components.

The analysis of several state-of-the-art measurement systems highlighted that fault detection is usually scattered all over different hierarchies, mainly with
reference to devices hierarchy. This means that concrete virtual devices classes contains code for fault detection resulting in code duplication that will be difficult to comprehend and maintain [Arpaia, 2007].

3.4.5 **Synchronizer**

Tasks are synchronized by means of a Petri Net modeling an execution graph, where each node represents a task and the arrows among nodes imply that an arriving node can be executed after the starting node. This allows synchronization to be abstracted above the code-level so that the Test Engineer can work at a more intuitive level.

The main basic idea is to have a software component capable of managing the execution of generic tasks by modeling sequential and parallel task executions, tracing the execution status of each task, and determining the task available for the execution, step by step.
Chapter 4

4 DSL – Domain Specific Languages

“Works of imagination should be written in very plain language; the more purely imaginative they are the more necessary it is to be plain.”

This section describes what a domain specific language is, what kind of advantages and disadvantages a DSL has and also what common DSL analysis, design and implementation patterns exist.

---

1 Samuel Taylor Coleridge
4.1 What is a domain specific language?

To understand the meaning of the term domain specific language or more precisely domain specific programming language the term programming language is defined. One possibility is given by [Raphael A., 1995]:

“A programming language or computer language is a standardized communication technique for expressing instructions to a computer. It is a set of syntactic and semantic rules used to define computer programs. A language enables a programmer to precisely specify what data a computer will act upon, how these data will be stored/transmitted, and precisely what actions to take under various circumstances.”

- However there exists no definition which all authors agree upon. Watts therefore proposes [David A., 1990] some criteria which have to be fulfilled by a programming language: Must be universal (every problem must have a solution that can be programmed in the language, if that problem can be solved at all by computer).
- Must be implementable on a computer.
- Should also be reasonably natural for solving problems, at least problems within its intended application area.

Programming languages in general can be grouped or classified by different criteria. Possible criteria are the purpose (for example FORTRAN for scientific programming versus C [Brian W. K, 1988] for system programming), the paradigm (LISP as a functional language or small talk as a object oriented language), the generation (1GL up to 5GL), whether it is imperative or declarative and domain specific or general purpose. General purpose languages (GPLs) are less specialized and are suited for a wide area of applications from business processing up to scientific computing. Java\(^1\) is a prominent representative.

\(^1\) http://java.sun.com
The term domain specific means that the language is explicitly tailored to a
target domain. Complex constructs and abstraction of the domain are offered
within the language increasing its expressiveness in comparison to GPLs. It is
possible to express solutions for domain problems with a lesser effort. The
higher abstraction and the compactness and therefore better readability and
writability enables a larger group of people with less programming knowledge to
be productive using the DSL. This leads to productivity gains in general and also
to decrease maintenance costs.

Often a DSL does not fulfill all criteria given by Watts. Nevertheless, many
DSLs are regarded as special programming languages. Today there are many
well known DSLs like HTML, SQL, VHDL, make (software build process),
Latex (document preparation), BNF (context free grammars) or even Excel.

The use of DSLs is not new. These languages had been named special-
purpose languages, end-user languages or as Bentley [Bentley, 1989] called
them “little languages” before the term domain specific language was coined.
Already in 1957 APT [Brown, 1963], a language for numeric controlled
machines was developed at the MIT, which can be considered as one of the first
available DSLs. The boarder between a DSL and a GPL is fuzzy, for example
COBOL was considered a GPL but also a DSL for business applications.
Another example is Prolog which can be understood as a DSL for applications
specified by the predicate calculus. One attempt to classify a language has been
done by Jones [Greenfield, 2003]. A higher level stands for more domains
specific whereas a lower level means more generality (table IV.1). As stated by
Mernik [Mernik, 2005] the domain-specificity of a language is a matter of
degree. In this thesis a definition by the former will serve as guidance:

“DSLs are languages tailored to a specific application domain. They offer
substantial gains in expressiveness and ease of use compared with GPLs in their
domain of application”
### 4.1.1 Advantages

A DSL offers different advantages. Productivity and maintainability [Van Deursen, 1997] are increased due to an appropriated domain specific notation. DSLs are more suitable for end-user programming. Domain experts are able to understand, validate, modify and develop within the language (better readability, writability and high abstraction). The gains can be measured quantitatively and qualitatively. Most qualitative reasoning is backed up by practical observations. According to [Mernik, 2005] the quantitative validation of DSL advantages is an ongoing field of research, yet supporting results is reported. Fig. 4.1 shows the advantage of DSLs regarding to long term cost. Because of the concise nature and the domain fitting notation DSLs are up to a certain degree self-documenting. This also facilitates the embodying of domain knowledge which eases reuse [Duggan, 2000] and conservation.

Another advantage is the possibility to validate at domain level [Consel, 2002]. While normal GPL compilers do not know about any domain concept beyond the general language constructs, a DSL can be checked for any domain specific constraint. An example may be real time properties: as long as for every language construct a certain execution time is ensured, it is possible to automatically proof the whole program. Just as verification, optimization can be done more effectively at the domain level [Basu, 1997].

<table>
<thead>
<tr>
<th>DSL</th>
<th>domain</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>GPL</td>
<td>5</td>
</tr>
<tr>
<td>VHDL</td>
<td>hardware design</td>
<td>17</td>
</tr>
<tr>
<td>HTML</td>
<td>web pages markup</td>
<td>22</td>
</tr>
<tr>
<td>SQL</td>
<td>database queries</td>
<td>25</td>
</tr>
<tr>
<td>Excel</td>
<td>spreadsheets</td>
<td>57</td>
</tr>
</tbody>
</table>

Table IV.1: *Well known DSLs* [Capers, 2007].
4.1.2 Disadvantages

A DSL has not only advantages, but also potential shortcomings. One drawback is the high development effort which is needed for a new language. The language developer needs at least experience in language design and knowledge about the target domain. He has to find fitting abstractions, the right scope and balance between GPL and DSL constructs. Furthermore the language must be implemented and maintained.

Figure 4.1: Simplified cost prediction for DSL-based methodologies [Devanbu, 1998]

Other problems are tooling, user training costs and performance. While general purpose languages such as Java or C#\(^2\) have a strong tool support, corresponding tools for a new DSL have to be created. IDEs like Eclipse or Visual Studio offer deep integration with these languages like powerful editors with syntax highlighting and checking, integrated compilers and advanced debuggers.

Creating a tool ecosystem for a DSL is a time consuming process which adds to the total costs caused by language design and implementation.

Without a development methodology and suitable tools the risk is high that the DSL development costs surpass the estimated saving by using a DSL. The mentioned training costs originate from the fact that possible DSL users have by definition never used the language before, however this is mitigated as in most cases the new language should match the domain expert’s expectations.

Often a DSL will suffer from a lower performance than a hand written software. As long as performance is not critical the other DSL benefits will make this a minor problem. Nevertheless are some cases performance can be equal or faster because optimization is possible on a high abstraction level but in most cases the potential is limited.

4.1.3 Development phases and patterns

The development of a DSL can be divided into different phases. The design and the implementation phase. A finer grained phase subdivision is possible. Five stages can be distinguished: decision, analysis, design, implementation and deployment. The development process of a DSL has not to follow these phases sequentially. Different authors [Thibault, 1999] have identified numerous patterns which are reoccurring in DSL development and can serve as guidance for a developer without prior expertise in this field. Each pattern can be assigned to one of the five phases. The patterns are divided into decision patterns, analysis patterns, design patterns and implementation patterns each capturing common approaches. In the following section phases and patterns will be described according to the extensive analysis by Menrik et al. [Mernik, 2005].

4.1.4 Decision phase

Before the development of a new DSL can begin, a decision has to be made. Is it feasible or not? Economic considerations have to be taken into account. Do the accumulated development, deployment and maintenance costs justify a new DSL in comparison with other conventional approaches? Is there already a
suited existing DSL? If so are documentation and maintenance good enough? If not, is the risk developing a new DSL acceptable?

The following decision patterns have been identified. Most of them based on the same general concerns such as allowing domain experts with less programming experience [Thibault, 1999] to develop software or improving software economics.

- **Notation** An improved new or existing domain specific notation can be a definitive factor. Two common subpatterns are the transformation of a visual to a textual notion and the creation of a user friendly notation for an existing API. The first pattern for example enables easier composition for large artifacts.

- **AVOPT** Domain-specific Analysis, Verification, Optimization, Parallelization and Transformation for applications developed in a GPL are in general time consuming and hard to automate due to for example source code complexity. With a well defined DSL AVOPT is more feasible.

- **Task automation** In some cases GPL programming suffers from repetitive programming tasks. Automatic code generation driven by an appropriated DSL can ease this [Smith, 2006].

- **Product line** Some software products do not exist as a single standalone application but are part of a product line or software family, sharing common parts. A DSL can facilitate the specification and support automated assembly [Weiss, 1999].

- **Data structure representation** Representing structured data in an easy to read, write and maintainable form assists in making complex structures accessible. An appropriated DSL can help achieving these goals. YAML [Ben-Kiki, 2004] and JSON [Crockford, 2006] are examples.
• **Data structure traversal** Like representation, traversal of data structures can often be expressed more effective with a fitting DSL (for example SQL [Groff, 1999]).

• **System front-end** DSL based configuration and adaption for system front-ends.

• **Interaction** Text, menu, dialog or voice based applications which interact with the user can benefit from a DSL which specifies input and reaction in a high level representation.

• **GUI construction** Often GUI design is done by using a DSL. For example XUL and XAML are XML based DSL for GUI description [Bishop, 2006].

### 4.1.5 Analysis phase

After the decision in favor for a (new) DSL is made, the specific domain has to be analyzed with the goal of gathering as much domain knowledge as possible. It is important to ensure a high quality of the gathered material and to have access to domain experts. The term domain analysis was introduced by Neighbors [Neighbors, 1980] and defined as identifying similar objects and operations in a particular domain. Different sources of information can be examined for example already existing technical documents, APIs and GPL code or knowledge from domain experts.

After gathering the knowledge must be clustered to find meaningful abstractions and must be consolidated. In most cases the results of the analysis are a domain definition, the domain specific terminology and concepts, a domain model, the domain scope and a description of the (operational) semantics. Fig. 4.2 summarizes different sources and possible results. Yet there is no widely adopted notation to capture the results of the analysis phase.

Three different domain analysis patterns can be identified: informal, formal and extraction from code.
Informal pattern

The *informal pattern* means that the domain analysis is done informally and therefore no formal process is used. Most DSLs are developed without an analysis methodology [Mernik, 2005]. This often leads to incomplete requirements and can complicate the development process. While it is possible to get first results earlier the quality is not as high as with formal patterns. For simple domains an informal process is often enough.

Formal pattern

Domain analysis can also be done using a defined process/methodology. Those which use a methodology can be counted to those that follow the formal pattern. Using a formal pattern helps to avoid missing important parts of the
domain and can lead to more appropriate requirements. A large number of methodologies used, come from another field of research: domain engineering. Domain engineering is derived from the area of software reuse and refers to the systematic modeling of a target domain. This is strongly related to the notation of program families [Van Der Linden, 1998] and software product lines [Sugumaran, 2006].

While domain engineering and analysis techniques focus mainly on commonalities, family and product line analysis examine the variations inside a domain. Several methodologies exist today: FAST (Family-Oriented Abstractions, Specification and Translation) [Weiss, 1999], Sherlock [Valerio, 1997], DSSA (Domain-Specific Software Architectures) [165], DARE (Domain Analysis and Reuse Environment) [Frakes, 1998], FODA (Feature-Oriented Domain Analysis) [Kang, 1990], PROTEUS [CAP, 1994], ODE (Ontology-based Domain Engineering) [De Almeida Falbo, 2002] or ODM (Organization Domain Modeling) [Simos, 1998]. This list consists of the most well know methods but is by no means complete.

An example where FODA and FAST are applied can be found in [Mernik, 2005]. While most methodologies have a graphical feature diagram or domain model as result, Deursen and Klint propose a formalized textual representation which can be used to generate UML diagrams or other types of documentation even code.

Semi formal

A specific semi formal approach (domain driven design) covering analysis is proposed in [Evans, 2003]. The creation of a fitting domain model is most important in domain driven design. At first domain experts and software architects try to find a domain model which serves as a base for a common communication language (Ubiquitous Language). This language will be used

---

3 The Feature Description Language (FDL), which is a separate DSL again.
later on in all aspects of the development process. It is advised that the notation for the domain model is UML. Not only one large diagram, but several small diagrams each describing a certain aspect or part should be used. The reason behind this is avoiding cluttering and reducing complexity. The UML artifacts should be accompanied by documents that contain information not captured by UML like the meaning of concepts or what certain objects are supposed to do. In comparison to other methodologies Evans gives extensive information how to continue after the domain model is established or the feature analysis is done.

*Extraction from code pattern*

The last identified pattern extraction from code derives a DSL directly from an existing implementation. In most cases this implementation is done in a GPL though it is also possible to derive from another DSL.

### 4.1.6 Design phase

The design of a DSL and therefore the development of the language itself is based on the results of the earlier phases. Two questions have to be answered approaching the design:

1. How is the DSL related to existing languages and what kind of formal description for the language is chosen? With each question different possible design patterns are associated helping to find an appropriated answer.

2. Creating a language based on an existing one can have different advantages. Some users may be familiar with the base language resulting in reduced training cost. Common operations such as arithmetic’s for the family of C languages are well known to many developers. Furthermore an existing implementation and/or eco system can be leveraged. Three different approaches reusing existing language can be distinguished. The fourth approach is the entirely new development of a language.
**Piggyback**

The new language can piggyback domain specifics feature on part of the existing language. Examples are Hancock [Cortes, 2000], lava [Sirer, 1999] or Facile [Schnarr, 2001]. Hancock is a DSL for high performance signature processing and it piggybacks on C by modifying language parts and adding processing related constructs. From this DSL, C code is generated again. Similar to that lava, a production grammar DSL to describe and generate test cases for a JVM, piggybacks on the textual Java byte code representation. The byte code is generated from the DSL. The Facile language helps developing high performance processor simulation, also by augmenting C.

**Extension**

A related pattern is extension. The base language is extended by features corresponding to domain concept. In comparison to piggybacking the base language is not modified or replaced. A problem of this approach is the seamlessly integration of new features with existing ones. A DSL which follows the extension pattern is SWUL [Bravenboer, 2004], SWUL supports the development of Java SWING GUIs and is embedded into Java.

**Specialization**

Developing a new DSL does not always mean to create something new. A more uncommon pattern is specialization (not to confuse with specialization in UML). An existing language is reduced to fit the needs of a special domain. Examples are RPython [Rigo, 2006] or OWL-Lite [Van Harmelen, 2002]. RPython is a subset of the Python language used inside the PyPy project [Rigo, 2006]. The complexity of Python is reduced in order to make C code generation from RPython easier.
4.2 Model Driven Engineering (MDE)

Model Driven Engineering (MDE) is the new trend in software engineering. MDE is the collection of all approaches that use models as a core principle for software engineering. The Model Driven Architecture (MDA) is the proposed approach for the MDE given by the Object Management Group (OMG). The aim of the MDA is to reach an abstraction level that is more focused on defining the structure and behavior of the system disregarding the underlying implementation technology.

The core element of the MDA is the Model Object Facility (MOF), which aim to enable the development and interoperability of model and metadata driven systems, such as modeling and development tools, data warehouse systems and metadata repositories. For realizing this, MOF provides a metadata management framework, and a set of metadata services.

If we look to the history of software engineering (Fig. 4.3), we can detect that we are continuously searching for a technique that provides a better and more natural approach for defining a system.

![Figure 4.3: History of software engineering](image)
4.3 Basic Concepts

4.3.1 Model

The term model is applicable in a broad area, which leads to many definitions. For example, a definition of model according to Benyon is [Benyon, 1997], “A model is a representation of something, constructed and used for a particular purpose.” The model is always the representation of something.

A model on its own has no meaning. The meaning of the model is related to the situation and context wherein the model is used. Like information and data [Harel, 004], the data is the syntactic representation of information. Data on its own has no meaning, but in combination with an interpretation, the information behind it can be extracted and understood.

4.3.2 Meaning of a model

The modeler as constructor of the model will define together with constructing the model the meaning of the model. The modeler will construct the model in such way that based on the representation the meaning can be extracted. Therefore, the modeler is using already commonly understood concepts. The role of the interpreter is to extract the meaning from the model. The interpreter is only capable of extracting the correct meaning if the interpreter has the same common understanding of the concepts used for the model. The exchange of a model between a modeler and an interpreter is called communication.

We can assign communication with a degree of meaning. The degree of meaning can be fuzzy, but we can at least define a minimum and maximum degree of meaning. The minimum degree of meaning is called meaningless, and maximum degree is called meaningful. If the modeler communicates with the interpreter, the modeler has a purpose for communicating. The communication
between the modeler and interpreter is meaningful if the purpose is obtained, if the purpose is not obtained the communication is meaningless.

To enlarge the chance that the interpreter can understand the model, the modeler can refer to a description of the notation of the model. This can be useful if the notation of a truth table is new for the interpreter. Therefore, the interpreter should be capable of interpreting the description of the truth table; otherwise we need again a description of the description of a truth table.

### 4.3.3 Language

For structurally describing something, we use a *language*. A language can be compared with the common understanding as described in previous section. The language is used for communication, and will at least need the following concepts. A language needs a concrete notation, which can be stored or transported. Furthermore, an interpretation is needed that will explain the meaning of the language constructs. These definitions are the fundamental concepts of a language, and are described as *syntax* and *semantics* [Harel, 2004]. The syntax of the language defines the notation, and the semantics describes the meaning of the notation.

Both syntax and semantics can be divided into aspects that are more specific. For the syntax those aspects are *concrete syntax*, *syntax mapping*, and *abstract syntax*, and for the semantics those aspects are *semantic mapping* and *semantic domain*. Those aspects are related to each other in some way. The Fig. 4.4 shows an overview of those aspects and the relation with each other.

![Figure 4.4: Semantic and Syntax mapping](image)

The syntax of the language is divided into *concrete syntax* and *abstract syntax*. Where the concrete syntax defines the physical notation, the abstract syntax
defines the structure of the notation. The structure of the notation is defined independently of the physical notation. Both syntaxes are mapped to each other by means of the syntax mapping, which provides the ability for defining a program using the physical notation according the abstract syntax.

For describing the meaning of the language the semantics are used, which describes the meaning in terms of the concepts that are already well-defined and well-understood. The well-defined and well-understood concepts are covered in the semantic domain, which is part of the semantics. For the semantic domain, we can use a variety of notations, like natural language or mathematical definitions. The abstract syntax is mapped to the semantic domain. This provides the abstract syntax with a well-defined and well-understood meaning.

As for everything we would like to describe, we need a language for describing it. In the case of the defined language aspects, it is not necessary that the same language is capable of describing each aspect. The language used for describing models is called a modeling language. The relationship between the modeling language and the model is that the model is expressed by using the modeling language. The modeler and interpreter need an understanding of the modeling language. The modeler can construct the model, based on this understanding. For the interpreter, the understanding will provide the ability to extract the correct meaning of the model.

4.3.4 Metamodel

A model that represents a modeling language is called a metamodel [Seidewitz, 2003]. Meta is Greek and is used for describing something. In the case of a metamodel it describes the possible models that can be expressed using the language, as shown in Fig. 4.5. The model is an instantiation based on the metamodel. The relationship between a model and metamodel is called an instance of relationship.
4.3.5 **Meta Metamodel**

A meta metamodel is a specialized metamodel that describes other metamodels. The position in the modeling hierarchy defines if a metamodel is a meta metamodel.

4.4 **MDSD Model driven software development**

The application of models to software development is a long-standing tradition, and has become even more popular since the development of the Unified Modeling Language (UML).

Yet we are faced with ‘mere’ documentation, because the relationship between model and software implementation is only intentional but not formal. We call this flavor of model usage model-based when it is part of a development process. However, it poses two serious disadvantages: on one hand, software systems are not static and are liable to significant changes, particularly during the first phases of their lifecycle. The documentation therefore needs to be meticulously adapted, which can be a complex task – depending on how detailed it is – or it will become inconsistent. On the other hand, such models only indirectly foster progress, since it is the software developer’s interpretation that eventually leads to implemented code. These are the reasons why many programmers consider models to be an overhead and see them as intermediate results at best.

Model-Driven Software Development has an entirely different approach: Models do not constitute documentation, but are considered equal to code, as
their implementation is automated. MDSD [Stahl, 2006] therefore aims to find
domain-specific abstractions and make them accessible through formal
modeling. This procedure creates a great potential for automation of software
production, which in turn leads to increased productivity. Moreover, both the
quality and maintainability of software systems increase. Models can also be
understood by domain experts. This evolutionary step is comparable to the
introduction of the first high-level languages in the era of Assembler
programming. The adjective ‘driven’ in ‘Model-Driven Software Development’
— in contrast to ‘based’ — emphasizes that this paradigm assigns models a central
and active role: they are at least as important as source code.

To successfully apply the ‘domain-specific model concept, three requirements
must be met:

- Domain-specific languages are required to allow the actual
  formulating of models.
- Languages that can express the necessary model-to-code
  transformations are needed.
- Compilers, generators or transformers are required that can run the
  transformations to generate code executable on available platforms.

MDSD may sound a lot like MDA. This is correct to a certain extent. In
principle, MDA has a similar approach, but its details differ, partly due to
different motivations. MDA tends to be more restrictive, focusing on UML-
based modeling languages. In general, MDSD does not have these restrictions.
The primary goal of MDA is interoperability between tools and the long-term
standardization of models for popular application domains. In contrast, MDSD
aims at the provision of modules for software development processes that are
applicable in practice, and which can be used in the context of model-driven
approaches, independently of the selected tool or the OMG MDA standard’s
maturity.

Basically Model-Driven Software Development consists of two major
aspects. The first one is processing models, i.e. checking their validity,
transforming them into other models as well as generating code (and other textual artifacts) from models. The other aspect addresses the creation of models. Traditionally, the processing of models has received more attention from the MDSD community. In particular, in Eclipse (open source) community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, the Graphical Modeling Framework is a tool that allows developers to easily define graphical editors for EMF-based meta models. Graphical editors are not enough, though. Many problems are better described with textual concrete syntaxes. As part of the Eclipse Modeling Project, there’s a placeholder project called TMF (for Textual Modeling Framework) which will address exactly this challenge – defining “nice” textual syntaxes for EMF-based meta models.
PART II: PROPOSAL
Chapter 5

5 Domain Specific Language for Magnetic Measurements

A DSL can be regarded as a programming or specification language dedicated to a particular domain or problem. The advantage of a domain-specific language in contrast to a general purpose language is that the DSL provides appropriate built-in abstractions and notations. In particular, DSL uses terms derived from a model created for a particular problem domain and used for defining components or complete solutions to be used in that domain. A domain can be seen as a specific setting with an implicit set of artifacts, actors and processes [Object Management Group, 2003].
CHAPTER 5

5.1 Magnetic Test Domain and FFMM Architecture

At CERN, measurement systems were developed under different conditions and with variable requirements for the series tests, of the LHC superconducting magnets. The result is a number of systems whose software has scarce reusability, without the necessary separation between the generic and the specific code, the main design criterion to ensure a good maintainability.

Although a good base to develop a new control and/or measurement application is provided, a strict collaboration between developers is still required in order to fully integrate new applications.

The first step was to realize a new framework (FFMM presented in Chapter 3) was based on the following basic ideas [Arpaia, 2007]:

(i) The flexibility is achieved by means of the code reusability: rapid variations of measurement requirements due to the frequent occurrence of different small batches of tests are satisfied redesigning software by reusing modules.

(ii) Reusability is achieved by object-oriented approach and modularity: a suitable design of the code allows modules to be reused.

(iii) Incremental building of module libraries: once modules can be reused, a finite application domain will be saturated in a finite time.

(iv) Standardization of software structure and modules: a definition of code structure and patterns gives rise to the production of standard modules to be reused easily.

(v) Predefinition of a software structure of the test program, organized in standard modules: such an organization provides the user with templates to be filled for generating new code.
Correspondingly, the fundamental principle underlying the FFMM architecture is the decoupling of software components through three main layers (Fig. 5.1):

- **Base service layer** - Communication and service packages: This layer implements the necessary foundations for communications, utilities (like useful algorithms and class libraries), and an OS service abstraction package.

- **Core service layer** – Virtual Devices and Event-handling, Logging/Fault Detection: Virtual Devices are software components modeling in FFMM the concrete devices to be orchestrated during measurement processes. Event handling was implemented to let Virtual Device and other software components obtain the needed information about the state of components of their interest. Logging/Fault Detection are responsible for monitoring the state of the component devices and catching software faults such as stack overflow, live-lock, deadlock, and application-defined faults as they occur.

- **Measurement service layer** – Test management and acquisition synchronization are able to create groups of tasks to be synchronized to well defined events (e.g. start and stop or device events) as needed.
In Fig. 5.2, a typical FFMM configuration is shown.

![Figure 5.2: A typical FFMM configuration](image)

### 5.2 The proposed approach

After developing FFMM (Chapter 3), it was necessary to provide the test engineer with an easy and fast way to write a measurement script. To achieve this goal, MDSL has been developed.

The final use of the Domain-Specific Language is in its domain. For our purposes, a language is a set of terms and expressions which are bounded by a set of syntax and semantic rules and used for communication within a domain.

Some features common to all languages should be understood in order to develop a generic approach to language definition:

- **Concrete Syntax**: all languages provide a notation fostering the presentation and construction of models and programs in the language. This notation is known as its concrete syntax. There are two main types of concrete syntax: textual and visual. A textual syntax enables models and programs to be described in a structured textual form. A visual syntax presents a model or program in a diagrammatical form. The advantage of a textual syntax is that it is aimed at representing details, while a visual syntax at communicating structure.
Abstract Syntax: the abstract syntax of a language describes the vocabulary of concepts provided by the language and how they may be combined to create models or programs. It consists of a definition of the concepts, the relationships that exist between concepts and may also include rules stating how the concepts may be legally combined. It is important to emphasize that a language’s abstract syntax is independent of its concrete syntax and semantics. Abstract syntax deals solely with the form and structure of concepts in a language without any consideration given to their presentation or meaning.

Semantic: the semantics of a language describes what models or programs in the language actually mean and do. In the context of programming languages, execution semantics is essential in order to run programs written in the language. Semantics are also important in the context of modeling languages.

External and internal textual DSLs can be defined. An External DSL is a domain specific language represented in a separate language to the main programming language it's working with. This language may be a custom syntax, or it may follow the syntax of another representation (like XML). An Internal (or Embedded) DSL is DSL expressed within the syntax of a general purpose language. It's a stylized use of that language for a domain specific purpose.

5.3 DSL Requirements

Test engineers are not skilled programmers and have to produce concise and bug-free FFMM specific applications (Fig.5.3).
Thus, a new Measurements Domain Specific Language (MDSL) with specialized constructs was designed in order to:

1. define logical, numeric, and temporal conditions;
2. perform conditional branching, immediate verification of conditions, verification of conditions within a time period, and continuous verification of conditions;
3. be able to define events based on measurement value and attribute changes, time changes, external event notifications, and user inputs;
4. subscribe and unsubscribe to events, and respond to them with behaviors that include sending text messages to users or commands and generate measurements;
5. enable, configure and disable framework service;
6. be able to interact with the user through a command prompt;
7. compare measurement data against specified criteria within a specified time period, and compute results that are numeric and Boolean functions.
To meet these requirements has been developed a domain-specific language called MDSL. Before giving details on the architecture, the concept of Semantic Model has to be introduced.

5.4 The architecture

5.4.1 Semantic model

The Semantic Model of a DSL is a subset of the overall Domain Model for an application. In the context of a DSL, a semantic model is an in-memory representation, an object model, of the same subject that the DSL describes. While the DSL describes a state machine, the Semantic Model is an object model with classes for state, event, etc.

The semantic model was separated from the DSL in order to:

1. think about the semantics of this domain without getting tangled up in the DSL syntax or parse
2. be able to test the semantic model by creating objects in the model and manipulating them directly;
3. have an incremental approach, starting with simple internal DSL and after add an external DSL; this is possible because having an explicit semantic model we can support multiple DSLs, since both DSLs can parse easily into the same Semantic Model;
4. be able to evolve the model and language separately; if the model is to be changed, this can be explored without changing the DSL, by adding the necessary constructs to the DSL; or new syntaxes for the DSL can be experimented by just verifying the creation of the same objects in the model; two syntaxes can be evaluated by comparing how they populate the semantic model.

This separation of semantic model and DSL syntax mirrors the separation of domain model and presentation suggested in a DSL can be thought as another form of user interface [Bosch, 1996].
The proposed MDSL is based on a Semantic Model, seen as a part of the FFMM domain model. It captures the Measurement Test Procedure core structure and behavior. Semantic Model is part of the difference between working with DSLs and with general purpose languages. In Fig. 5.4, the proposed approach for the transformation of the Measurement Domain-Specific Description (MDSD) into the final code is shown.

The external DSL, written by the Test Engineer, is parsed to create an internal file treated by the semantic model (Fig.5.4). The external DSL, the DSL scripts i.e. the MDSD, the parser and the Semantic Model is very clearly separated. The MDSL scripts are written in a clearly separate language; the parser then reads these scripts and populates the Semantic Model. Direct writing in the internal DSL risks to mix up difficulties. An explicit layer of Expression Builders providing the necessary fluent interfaces to act as the language were conceived. MDSL scripts run by invoking methods on an Expression Builder which then populates the Semantic Model. Thus, in an internal DSL, parsing the DSL scripts are done by a combination of the host language parser and the Expression Builders.

Once a Semantic Model is defined, it is passed to Builder for code generation, i.e. the code is separately compiled and run.

![Figure 5.4: MDSL transformation in code](image)
The code generator is decoupled from the parser: a code generator can be written without having to understand anything about the parsing process, as well as tested independently too. For our project, the code generated is a script in C++ language to be compiled to obtain the executable code for the measurement.

5.4.2 Parser

Parsing is a strongly hierarchical operation. When a text is parsed, the chunks are arranged into a tree structure. Let's consider the simple structure of a list of events shown in Fig. 5.5.

```plaintext
BEGIN_SCRIPT
  scriptName=ID "t:"
  scriptDeclarations=Declarations
  scriptAssignments=ScriptAssignments
  scriptDeviceDefinitions=DeviceDefinition
  scriptDeviceConfigurations=ConfigurationStatement
  deviceConfigurations=DeviceConfigurations
  [taskState=TaskState]
  [taskExecutionStatement=TaskExecutionStatement]
  //int=FunctionSequence()
END_SCRIPT;

ScriptAssignment:
  [as=AssignStatement | cpp=CppCode];

DeviceSetting:
  cmd=CommandStatement | sets=SettingStatement | gets=GettingStatement | uses=Use_Statement |
  cmd=CommandStatement | sets=SettingStatement |
  cmd=CcppCode | as=AssignStatement;

NTask:
BEGIN_NTASK
  [taskName=TD (taskClass=STRING) "t:"
  taskDeclarations=Declarations
  taskActions=GenericStatement]
END_NTASK;
```

Figure 5.5: List of events

In this composite structure (Fig. 5.5), a list contains events, each one with a name and a code. There is no explicit notion of an overall list, but each event is still a hierarchy of events each containing a name symbol and a code string.

The proposed MDSD can be represented as a hierarchy: in this way, such a hierarchy is called a syntax tree (or parse tree). A syntax tree is a much more useful representation of the MDSD than the words; it can be manipulated in many ways by walking up and down in the tree. Basically, the parser reads the
textual MDSD, builds syntax trees and translates them into the Semantic Model. The syntax tree was built by means of a specific grammar, i.e. a set of rules describing how a stream of text is turned into a syntax tree. Grammars consist of a list of production rules, where each rule has term and a statement of how it gets broken down.

5.4.3 Builder

Code generators have been around for decades. They can trace their roots back to the origin of compilers. One of recent developments in code generation is Model-Driven Architecture (MDA) [Object Management Group, 2003]. It uses basic models and domains represent specific situations and then create code from that. A tool that implements the MDA concept allows developers to:

1. Produce models of the application and business logic.
2. Generate code for a target platform by means of transformations.

The major benefit of this approach is that it raises the level of abstraction in software development.

Model-Driven Architecture (MDA) is an approach to software development produced and maintained by the Object Management Group (OMG)\(^1\). MDA is not to be confused with Model-Driven Development (MDrD), also known as Model-Driven Software Development. MDrD is an approach to software development where extensive models are created before source code is written or generated. MDA is the OMG implementation of MDrD. The MDA concept is implemented by a set of tools and standards that can be used within an MDrD approach to software development.

The basis for automatic code generation is to read in project artifacts, such as class diagrams, activity diagrams, and requirements documents and turn them into meaningful and correct source code. The implementation of automatic code generators relies on the fact that most artifacts are created in the early stages if

\(^1\) http://www.omg.org
software development arises from UML notations and diagrams. UML (Unified Modeling Language) is a standard in which object-oriented design patterns can be easily recognized. Since these artifacts are repetitive and have design patterns they can be automated. Most simple implementations of automatic code generators use only the class diagram to create source code. Class diagrams have been the easiest to implement because of the inherited design pattern to object-oriented languages such as Java and C++.

5.5 The proposed architecture

The proposed architecture, shown in Fig. 5.6, is organized through a 2-way decomposition separating the developer view (FFMM Core) from test engineer view (DSL script):

1. **FFMM Core**, the involved data structures and classes of the framework
2. **DSL Script**, Domain Specific Language (DSL) code.

![Figure 5.6: Proposed architecture](image)

Figure 5.6 shows two possible views of FFMM for two different classes of user. On FFMM block, or on FFMM Classes, there are two way: the developer can
operate with code C++ at any level in the system and can define a measurement through a script in C++ code. On the other hand the test engineer, with limited effort and programmation skills, can operate at script level by means of the DSL, defining a procedure that the xPand / Builder will translate into C++. The interaction between xPand / Builder and FFMM Classes allows the execution of the measurement application described by the test engineer in the DSL script. The link between DSL script and xPand / Builder is bidirectional because there is a mechanism providing the programmer with suggestions about the code by giving insight into the FFMM core class structure. The next chapter will show in detail the new MDSL project.
PART III: IMPLEMENTATION
6 MDSL Implementation

The first part of this chapter will show how to create a new project in Eclipse by using openArchitectureWare (oAW) plug-in in order to define a new DSL. The second part, instead, will show how the new language, MDSL, was implemented.
6.1 **Eclipse platform**

The Eclipse platform\(^1\) was used to develop the proposed DSL. Eclipse is a multi-language software development platform comprising an IDE and a plug-in system to extend it. It is written primarily in Java and is used to develop applications in this language and, by means of the various plug-ins, in other languages as well C, C++, COBOL, Python, Perl, PHP and more. The initial codebase originated from VisualAge. In its default form it is meant for Java developers, consisting of the Java Development Tools (*JDT*). Users can extend its capabilities by installing plug-ins written for the Eclipse software framework, such as development toolkits for other programming languages, and can write and contribute their own plug-in modules. Language packs provide translations into over a dozen natural languages. Released under the terms of the Eclipse Public License; Eclipse is free and open source software.

Eclipse employs plug-ins in order to provide all of its functionality on top of (and including) the runtime system, in contrast to some other applications where functionality is typically hard coded. The runtime system of Eclipse is based on Equinox, an OSGi standard compliant implementation. This plug-in mechanism is a lightweight software componentry framework. In addition to allowing Eclipse to be extended using other programming languages such as C and Python, the plug-in framework allows Eclipse to work with typesetting languages like LaTeX, networking applications such as telnet, and database management systems. The plug-in architecture supports writing any desired extension to the environment, such as for configuration management. Java and CVS support is provided in the Eclipse SDK, with Subversion support provided by third-party plugins.

The key to the seamless integration (but not of seamless interoperability) of tools with Eclipse is the plug-in. With the exception of a small run-time kernel,

---

\(^1\) [http://en.wikipedia.org/wiki/Eclipse_%28software%29](http://en.wikipedia.org/wiki/Eclipse_%28software%29)
everything in Eclipse is a plugin. This means that every plugin developed integrates with Eclipse in exactly the same way as other plugins; in this respect, all features are created equal. Eclipse provides plugins for a wide variety of features, some of which are through third parties using both free and commercial models. Examples of plugins include UML plug-in for Sequence and other UML diagrams, plug-in for Database explorer, and many others. The Eclipse SDK includes the Eclipse Java Development Tools, offering an IDE with a built-in incremental Java compiler and a full model of the Java source files. This allows for advanced refactoring techniques and code analysis. The IDE also makes use of a workspace, in this case a set of metadata over a flat file space allowing external file modifications as long as the corresponding workspace "resource" is refreshed afterwards. The Visual Editor project (discontinued since June 30, 2006) allows interfaces to be created interactively, thus allowing Eclipse to be used as a RAD tool. Eclipse's widgets are implemented by a widget toolkit for Java called SWT, unlike most Java applications, which use the Java standard Abstract Window Toolkit (AWT) or Swing. Eclipse's user interface also uses an intermediate GUI layer called JFace, which simplifies the construction of applications based on SWT.

### 6.1.1 oAW openArchitectureWare

When starting a new project we must first create xText project in order to define a new language. Our Xtext projects are based on the Eclipse plug-in architecture (oAW). The purpose of this section is to illustrate the definition of external DSLs using tools form the Eclipse Modeling Project (EMP).

OpenArchitectureWare [oAW] is nowadays one of the most used MDDS frameworks. Much of this success results from its flexibility: rather than providing pre-made generator templates, oAW serves as a generator toolkit and enables users to easily create tailored generator solutions that really fit their needs. Besides this flexibility, openArchitectureWare users benefit from the
tight integration with Eclipse: not only does oAW come with an array of editors that make writing templates and workflows an easy task. oAW also delivers refactoring support, easy navigation, an incremental project builder and a debugger. It supports parsing of arbitrary models, and a language family to check and transform models as well as generate code based on them. Supporting editors are based on the Eclipse platform. oAW has strong support for EMF (Eclipse Modelling Framework) based models but can work with other models, too (e.g. UML2, XML or simple JavaBeans). At the core there is a workflow engine allowing the definition of generator/ transformation workflows. A number of prebuilt workflow components can be used for reading and instantiating models, checking them for constraint violations, transforming them into other models and then finally, for generating code. In other words oAW helps with meta modeling, constraint checking, code generation and model transformation.

More recently a framework has been developed that supports the creation of textual domain-specific languages (DSL): xText

The main focus is on the xText framework. We will start by defining our own DSL in an xText grammar. Then we will use the xText framework to generate a parser, an Ecore-based metamodel and a textual editor for Eclipse. Afterwards we will see how to refine the DSL and its editor by means of xTend extensions. Finally, we will learn how one can generate code out of textual models using the template language xPand. The actual content of this example is rather trivial our DSL will describe entities with properties and references between them from which we generate Java classes according to the JavaBean conventions a rather typical data model. In a real setting, we might also generate persistence mappings, etc. from the same models.
6.1.2 \textit{xText project}

\textit{xText} is part of the openArchitectureWare project (which is in turn part of Eclipse GMT). Based on an EBNF like notation, \textit{xText} generates the following artifacts:

- A set of AST (Abstract Syntax Tree) classes represented as an EMF-based metamodel.
- A parser that can read the textual syntax and returns an EMF-based AST (model).
- A number of helper artifacts to embed the parser in an oAW workflow.
- An Eclipse editor that provides syntax highlighting, code completion, code folding, a configurable outline view and static error checking for the given syntax.

\textit{xText} starts from a description of a textual syntax (the grammar) and derives an AST class model (the metamodel) from that concrete syntax definition. The linking of cross references within the same model or through different models can be done separately from the textual syntax description. Linking can be a quite complicate process if you consider scopes, namespaces and visibility of elements we think that it is crucial for a textual language framework to allow the separation of parsing and linking.

The separation of these two concerns (parsing and linking) helps to implement more sophisticated linking logic independent of the concrete syntax. Additionally we can check the AST before doing additional linking and transformations. In some cases you even don't want to link references up-front, but want them to be looked up dynamically.

Linking in \textit{xText} can be done in several ways. The easiest way is to make use of so called extensions. Extensions are operations that can be annotated to existing meta classes. Another solution is to transform the AST to a “real” meta model. This has the additional advantage that the concrete syntax can be changed, or one can have several different concrete syntaxes for the same metamodel. The
necessary transformation is relatively straightforward to define, because it is basically a one to one mapping with some additional linking logic.

To create a new textual DSL with xText, we need up to three files that depend on each other (Appendix), according to the following steps:

- Start up Eclipse with oAW installed in a fresh workspace
- Select File > New... > Project... > openArchitectureWare > Xtext Project
- Specify the project settings in the wizard dialog.
- Click Finish (Fig 6.1)

![Figure 6.1: Wizard to start new Xtext project](image)

The wizard creates three files, my.dsl, my.dsl.editor, and my.dsl.generator (Appendix):
• *my.dsl* is the language project, in which we will define the grammar for our DSL. After running the Xtext generator, this model also contains a parser for the DSL and a metamodel representing the language.

• *my.dsl.editor* will contain the DSL editor.

• *my.dsl.generator* contains an openArchitectureWare code generator skeleton.

### 6.1.3 Defining the Grammar

An *xText* grammar consists of a number of rules (Model, Message, Field and Type). A rule is described using sequences of tokens. A token is either a reference to another rule or one of the built-in tokens (STRING, ID, LINE, INT). *xText* automatically derives the meta model from the grammar, instead, the meta model is basically a data structure whose instances represent the structure of sentences in the language.

A rule results in a meta type, the tokens used in the rule are mapped to properties of that type (comments, name, fields). Different assignment operators are been used. The equals sign (‘=’) just assigns the value returned from the token to the respective property (the property will have the type of the token) and ‘+’ adds the value to the property.

So after creating our new *xText* project, we can define the grammar for our MDSL (an example is show in Fig. 6.2).
The grammar specifies the metamodel and the concrete syntax for our Measurements Domain Specific Language (MDSL).

### 6.1.4 Generating the DSL Editor

We will use the grammar language provided by xText. The following screen shot shows how the syntax is described for the FFMM-DSL. In fact language and tooling used for describing DSL syntax is bootstrapped, i.e. it is implemented using the xText framework itself. Bootstrapping is a common technique in the field of language and compiler development. If you can bootstrap your language and tools, this proves a certain level of maturity of the tools.

Having specified the grammar, we can now generate the DSL editor:

- Right-click inside the xText grammar editor to open the context menu.
- Select *Generate xText Artifacts* to generate the DSL parser, the corresponding metamodel and, last but not least, the DSL editor (Fig. 6.3).
6.1.5 Running the Editor

To see the generated editor in action, we must run the plug-ins in an Eclipse installation. The most convenient way to do this is to start a new Eclipse application from within the running Eclipse:

- Select the editor plug-in and choose Run As > Eclipse Application from its context menu.

The generated editor can also be deployed into an existing Eclipse installation. Note that you have to redeploy the editor on every change you apply to the plug-ins. To install the editor into the Eclipse we are currently running, perform the following steps:

- Choose Export... > Deployable plug-ins and fragments...
- The Export dialog appears. Select the three DSL plugins.
CHAPTER 6

- Enter the path to your Eclipse installation. Make sure the selected directory contains the Eclipse executable and a folder named plugins. Usually, the directory is called eclipse.
- Choose Finish (Fig. 6.4).
- Restart Eclipse.

![Deployment of the DSL plug-ins](image)

Figure 6.4: Deployment of the DSL plug-ins

6.2 Code generation with xPand

The xText wizard already created a generator project for us. In this part we must connect the FFMMs class with our new language DSL. Part of xPand implemented is shown in Fig. 6.5.
6.2.1 The Grammar Language

At the heart of xText lies its grammar language. It is a lot like an extended Backus-Naur-Form (BNF), but it doesn’t describe only the concrete syntax, but can be also used to describe the abstract syntax (metamodel).

As stated before, the grammar is not only used as input for the parser generator, but it is also used to compute a metamodel for your DSL.

The analysis of text is divided in two separate tasks: the lexing and the parsing.

The lexer is responsible of creating a sequence of tokens from a character stream. Such tokens are identifiers, keywords, whitespace, comments, operators,
etc. XText comes with a set of built-in lexer rules which can be extended or overwritten if necessary.

The parser gets the stream of tokens and creates a parse tree out of them.

6.3 Type Rules

The name of the rule is used as name for the metatype generated by Xtext.

6.3.1 Assignment tokens / Properties

Each assignment token within an XText grammar is not only used to create a corresponding assignment action in the parser but also to compute the properties of the current metatype. Properties can refer to the simple types such as String, Boolean or Integer as well as to other complex metatypes. It depends on the assignment operator and the type of the token on the right, what the type actually is. There are three different assignment operators:

- Standard assignment '=' : The type will be computed from the token on the right.
- Boolean assignment '?=' : The type will be Boolean.
- Add assignment '+=' : The type will be List. The inner type of the list depends on the type returned by the token on the right.

An example in our project of these assignment operators is show in Fig. 6.6.
6.3.2 Cross References

Parsers construct parse trees not graphs. In order to implement crosslinks in the model, one usually has to add third task: the linking. However, xText supports specifying the linking information in the grammar, so that the metamodel contains cross references and the generated linker links the model elements automatically. Linking semantic can be arbitrary complex. xText generates a default semantic which can be selectively overwritten.

```
DeviceSetting:
  cmdID=CmdID | sets=SettingStatement | gets=GettingStatement | uses=Use_Statement |
  cppCode=CppCode | set=AssignStatement;

#Task:
"EO2D_TASK" meansName=ID (intoExp=-STRING) ; ":" |
(cashDeclarations=Declarations)* |
| cashActions=GenericStatement; * |
END TASK;

GenericStatement:
  cmdID=CmdID | sets=SettingStatement | gets=GettingStatement | uses=Use_Statement |
  cppCode=CppCode | set=AssignStatement |
  lst=ForStatement | ust=WhileStatement | lst=IfStatement | ust=WhileStatement;

Util_Statement:
  print=Print_ | delay=Delay_ | trigEvent=TrigEvent_; |
```

Figure 6.6: Example of assignment operators in our project

```
Entity : |
  *=entity* name=ID ("extends" superType={Entity})? |
  *(features=Feature)* |
*;
```

Figure 6.7: Entity

Have a look at the optional extends clause. The rule name Entity on the right is surrounded by squared parenthesis (Fig. 6.6). By default, the parser expects an ID to point to the referred element.
6.3.3 Metatype Inheritance

After to have define metatypes and its features we to have also define type hierarchies using the grammar language of \textit{xText}. We need to have more different kinds of “Feature” (Fig 6.7) we did create it with an abstract type rule like shown in Fig. 6.8.

![Abstract type rule](Image)

The transformation creating the metamodel automatically normalizes the type hierarchy. This means that properties defined in all subtypes will automatically be moved to the common supertype.
The ID Token

We also have seen the identifier token (ID). This is the token rule expressed in AntLR grammar syntax how shown in Fig. 6.9.

```
//LVPowerSupply definition
def LVPowerSupply:
  "LV_POWER_SUPPLY:" name=ID "\$L\$"("add=\$Form\$,"type_num=\$Form\$,"ser_num=\$Form\$,"pow_client=\$Form\$")?:
  //static LVPowerSupply createDevice( std::string name):
  //static LVPowerSupply createDevice( std::string name):

//LVPowerSupply Using
def LVPowerSupply:
  "LV_POWER_SUPPLY:" name=ID:

//LVPowerSupply Configuration
Cfg LVPowerSupply:
  "LV_POWER_SUPPLY:" name=ID "\$L\$"("inteface="face","devser=\$face","timeout=\$face");
  // void configure( int interfaceName, int devOrder, int timeout ];
```

Figure 6.9: Token rule expressed

The return value of the ID token is a String. So, we use the usual assignment operator "+", the value is assigned to will be of type String.

Comments

There are two different kinds of comments automatically available (Fig. 6.10) in any xText language.

```
// single-line comments and
/
 */
 multi-line comments
 */
```

Figure 6.10: Comments

Note that those comments are ignored by the language parser by default.

6.4 Defining the MDSL

Our goal was to create a simple scripting language for the test engineer; this problem has been addressed through the definition of a DSL. The test engineer has to follow the steps shown in Fig. 6.11 to define, set and execute a measurement task.
To be more precise the test engineer should first define the object (or device) that intends to use, than configure its setting and use it through appropriate commands, defined in device interfaces, which should be known by the test engineer. To make this task easier MDSL project provides one of the most useful things: the assistance to the measurement procedure definition.

While he writes the script, the test engineer, can click on $CTRL+SPACE$ to see the menu where all the possibilities are shown in Fig. 6.12.

![Figure 6.11: DSL test engineer steps](image)

![Figure 6.12: Assistance to the measurement procedure](image)
It is possible to appreciate the ease of writing and the flexibility of software. In appendix A all the scripts for magnetic dipole measurement are shown. In the following, for the sake of comparison two script fragments are shown. The Fig. 6.23 refers to a C++ script for permeability measurements (Appendix B). The Fig. 6.14 shows the same procedure written in DSL. The improvements in clarity and conciseness are evident.

```cpp
#include <ffmm.h>
#include <sstream>
#include <math.h>
using namespace ffmm::core::events;
using namespace ffmm::core::devices;
using namespace std;
_FFM.Initialization
#include "core/devices/FdiCluster.h"
#include "core/devices/EncoderBoard.h"
#include "core/devices/Keithley2k.h"
#include "core/devices/DAQmx.h"
#include "core/utils/FdiClusterDataConversion_byN2Ascii.h"
#include "core/events/FdiClusterListener.h"

std::string Cluster="FDI_Cluster_1";
std::string Encoder="Encoder_Board";
std::string Multimeter="Keithley2k";
std::string DAQM="NI_DAQ";
const int Encoder_slot=13;
const int Encoder_bus=4;
const int Encoder_Channel=1;
const int Encoder_mode=1;
const double Encoder_freq=2048;
const int Multimeter_intfNum=0;
const int Multimeter_busAddress=16;
const int Multimeter_timeout=100;
const int numberOf_FDI = 3;
const int surceStop = 1;
int Cluster_slot[numberOf_FDI]={12,11,10};
int Cluster_bus[numberOf_FDI]={4,4,4};
double Cluster_abs_gain_ = 1.0;
double Cluster_comp_gain_ = 10;
int SamplePerTurn = 1024;
int numberOfTurn = 4;
U32 AcquisitionBufferSize;
std::string Daq_channel_name = "AO_Ch";
std::string Daq_task_name = "Trap_G";
int Daq_channel = 0;
int Daq_timeOut = 200;
int Daq_generationMode = 0;
const double Daq_sample_rate = 1000;
int Daq_minVolt = -10;
int Daq_maxVolt = 10;
std::string path_name;
double epsC = 0.1;
int measurementCycle;
double plateaux[38] = {0,-0.1,0.1,-0.2,0.2,-0.3,0.3,-0.4,0.4,-0.5,0.5,-0.6,0.6,-0.7,0.7,-0.8,0.8,-0.9,0.9,-1,1,-1.2,1.2,-1.4,1.4,-1.6,1.6,-1.8,1.8,-2,2,-3,3,-5,5,-10,10,0};
BEGINSCRIPT
   NI_Daq->setTimingTrigger(Daq_sample_rate, 0, numOfSamples);
```

102
NI_Daq->startVoltage(signal, numOfSamples);
NI_Daq->waitGeneration();
NI_Daq->setTimingTrigger(Daq_sample_rate, 0, numOfSamples);
NI_Daq->startVoltage(signal, numOfSamples);
NI_Daq->waitGeneration();
Poco::DynamicAny plat;
while(!demagnetized)
{
    Poco::DynamicAny plat(plateau*4);
    environment->console->writeln(plat.convert<std::string>());
    if (plateau >= value1)
    {
        old_plateau = plateau;
        plateau = plateau/1.5;
    }
    else if (plateau >= value2)
    {
        old_plateau = plateau;
        plateau = plateau/1.2;
    }
    else
    {
        old_plateau = plateau;
        plateau = plateau/1.1;
    }
}

Figure 6.13: The part of the Script in C++

DEF ENCODER_BOARD: Enc_B WITH ["1", "2", "CERR"];
DEF ENCODER_BOARD: Enc_I WITH ["1", "2", "CERR"];
DEF ENCODER_BOARD: Enc_J WITH ["1", "2", "CERR"];
DEF ENCODER_BOARD: Enc_M WITH ["1", "2", "CERR"];
DEF ENCODER_BOARD: Enc_L WITH ["1", "2", "CERR"];
DEF ENCODER_BOARD: Enc_N WITH ["1", "2", "CERR"];

// Device Configuration
//**********************************************************************************
CFG ENCODER_BOARD: Enc_B WITH [ Encoder_bus, Encoder_slot ];
CFG ENCODER_BOARD: Enc_I WITH [ Encoder_bus, Encoder_slot ];
CFG ENCODER_BOARD: Enc_J WITH [ Encoder_bus, Encoder_slot ];
CFG ENCODER_BOARD: Enc_M WITH [ Encoder_bus, Encoder_slot ];
CFG ENCODER_BOARD: Enc_L WITH [ Encoder_bus, Encoder_slot ];
CFG ENCODER_BOARD: Enc_N WITH [ Encoder_bus, Encoder_slot ];

// Device Setting
//*******************************************************************************
CHG ENCODER_BOARD: Reset Cluster_0;
CHG ENCODER_BOARD: Reset Cluster_1;
SET ENCODER_BOARD: PErrxCluster_1 optl: SamplePerTurn, Cluster_min_gain, Cluster_compr_gain, CONT, 50000, optl: 10);
SET ENCODER_BOARD: StopSource Cluster_1, sourceStop);
SET ENCODER_BOARD: VoltageRangeOptxCluster_1 (Nano, Deq_minVolts, Deq_maxVolts);
CHG ENCODER_BOARD: Calibrate_Gain Cluster_1, 0, 1.0);
CHG ENCODER_BOARD: Calibrate_Gain Cluster_1, 1, 1.0);

Figure 6.14: The part of the Script in DSL
Chapter 7

7 Experimental results

This chapter will show how the new project reaches all the specifications required for the magnetic measurements, instead, in the following, the results of the qualification tests of the new system (MDSL and FAME) performed on an LHC main dipole at cryogenic conditions, i.e. 1.9 K, with a DC supply current (1500 A) and standard LHC cycle current. In the former case, the stability and the repeatability of the measurements are determined by evaluating synthetic parameters, i.e. such as the mean of the harmonic coefficients and its experimental standard deviation. In the latter case, the aim of field quality inspection is to understand the field changes in the main dipole in order to compensate undesired effects during LHC operation. In particular, the “snapback” phenomenon and the foremost field components will be detailed and explained, via the experimental results.
7.1 System Architecture

At CERN, the facility for testing the LHC main dipoles is installed in the SM18 hall [Mishra, 2005]. SM18 has six test clusters, each one consisting of two benches. Only one bench in a cluster can be used at a time. In the following, the main components of the bench for the tests of MDSL prototype are detailed, for both warm and cold conditions.

**Warm Conditions:**

- Portable Power Supply 20 $A$;
- Portable DCCT 60 $A$.

**Cold conditions:**

- Main Power Converter (14 $kA$, 15 $V$):
  - Voltage Source;
  - DCCT (DC Current Transducer);
  - FGC1 (Function Generator Controller first generation);
- Worldfip Gateway;
- Anti-cryostat (heated tube to give access to the magnet bore at room temperature)

The architecture of the MDSL prototype is shown in Fig. 7.1. The following components are common to both measurements condition:

- PC, with FFMM MDSL, managing the measurement station;

Fast Acquisition Measurement Equipment components follow:

- ADLINK PXIS-3320 chassis composed by:
  - ADLINK PXI-8570;
  - 6 FDI boards;
CHAPTER 7

- Encoder Board;
- Analog bucking (compensation of the dipole components);
- Micro Rotating Unit (μ RU), including:
  - HEIDENHAIN Rotary Encoder ERN420;
  - MAXON Angular Motor EC-40;
  - MAXON Motor Encoder HEDL 5540;
- New rotating coils shaft;
- MAXON Motor Controller EPOS 24/5;
- Digital Multimeter KEITHLEY 2000.

Only one of the two apertures of the magnet is under test (Fig. 7.1). The field measurement is carried out by means of a maximum of 24 FDI, integrating the signals produced by coils placed in the 12 sectors in which the shaft is divided.

Two FDI are required to acquire the absolute and compensated fluxes of each sector. At the moment, only 6 FDIs are used for the harmonic analysis of three super sectors (4 sectors connected in series). The Next step will be to add the remaining 18 FDIs, in order to acquire all the signals from the coils, and to duplicate the same architecture on the other aperture.
7.2 Overview of the test bench at SM18

The first prototype of the new platform was integrated at SM18 (Fig. 7.2). The measurements of the main bending dipole field were carried out at room temperature.
The dipole is connected directly to the Heinzinger PTN 135-20 20 A, 135 V, DC Portable Power Supply (Fig. 7.3). This configuration permits only measurement shorter than one hour, in order to avoid an excessive heating of the magnet.

![Portable Power Supply Heinzinger PTN 135-20 at SM18](image)

Figure 7.3: Portable Power Supply Heinzinger PTN 135-20 at SM18

The high-accuracy portable Direct Current-Current Transformer (DCCT) is connected directly to the Portable Power Supply and will be connected to the Multimeter (Fig. 7.4) in order to perform the desired current measurement.

For each measurement the coils shaft can turn with a fixed frequency. At the start of rotation, the first turn is dedicated to reach the desired angular velocity, then the 6 FDI's start acquiring and performing the integration of the 3 absolute and 3 compensate signals, coming from the 3 shaft sections.

![Digital Multimeter KEITHLEY 2000](image)

Figure 7.4: Digital Multimeter KEITHLEY 2000
7.3 Measurement setup

The measurement system is installed at the test facility hall for superconducting magnets at CERN (SM18). The setup architecture of the measurement station at SM18 is the same as in warm validation tests (Fig. 71). At cold conditions, the main power supply of the test facility was used providing a current up to $15 \, kA$. The LHC dipole under test was the MBBR 2427; only one aperture was considered. In the following, the DC measurements and the measurements with standard LHC cycle are reported.

7.4 DC Measurements

In order to define the repeatability purposes of the new station for magnetic measurement at CERN, using MDSL, several measurement sessions were defined, at a current plateau of $1500 \, A$ (the considered segment shaft has been the $5^{th}$). Each session is specified by changing the setting parameters, i.e. angular speed, signal gain, time measurement interval and number of samples per turn. Table VII.1 reports the settings for each measurement session.
The above table identifies three main measurement categories:

- **Scan on the angular speed**: the angular speed is increased from 6.28 rad/s (1 turn/s) up to 56.26 (8 turn/s), by fixing the FDIs gain and the number of samples per turn, in order to highlight the only effect of the rotation speed;
- **Cross-check between angular speed and gain**: gain and angular speed are adjusted to feed up the FDIs with a full scale signal. The rationale of such measurement is to investigate the performance using a trade-off
between a speed value and electronic gain, both affecting the amplitude of the FDIs input signal;

- **Repeatability on single turn acquisition**: a single turn for harmonic analysis was acquired separately from the others, such as reported in the table: the time interval of the measure is 0.125 s, 30 single turns were carried out in order to check the repeatability of the system in temporally decoupled acquisitions.

The direction of shaft rotation is the same for all the measurement sessions.

### 7.5 Measurement Procedure

The common settings of each measurement, defined via the MDSL user script is shown below:

```plaintext
//************************************
//Variable assignement
//************************************
AcquisitionBufferSize = numberOf_FDI*( SamplePerTurn/2)*4*2;

//************************************
// Device Definition
//************************************
DEF ENCODER_BOARD: Enc_B WITH ( "1", "1","CERN" );
DEF FDI_CLUSTER: Cluster_1 WITH (numberOf_FDI );
DEF KEITHLEY2K: Mult_M WITH ( "1", "2", "NI" );

//************************************
// Device Configuration
//************************************
CFG ENCODER_BOARD: Enc_B WITH ( Encoder_bus , Encoder_slot );
CFG FDI_CLUSTER: Cluster_1 WITH ( Cluster_bus , Cluster_slot );
CFG KEITHLEY2K: Mult_M WITH ( Multimeter_intfNum, .

//************************************
// Device Setting
//************************************
CMD FDI_CLUSTER: Reset ( Cluster_1, 0);
CMD FDI_CLUSTER: Reset ( Cluster_1, 1);
SET FDI_CLUSTER: Params2 ( Cluster_1, spt1, SamplePerTurn, Cluster_abs_gain_, Cluster_comp_gain_, CONT, 500000, spt2,10);
SET FDI_CLUSTER: Stop_Source ( Cluster_1, surceStop);
CMD FDI_CLUSTER: Calibrate_Gain (Cluster_1, 0, 1.0);
CMD FDI_CLUSTER: Calibrate_Gain (Cluster_1, 1, 1.0);

The complete script is in Appendix A.
```
In order to handle continuous rotating coil measurements are [Animesh, 1997], [Brooks, 2007]:

- Motor rotation speed and rotating direction.
- Time interval of the measurement.
- FDIs configuration (gain, samples to be acquired).
- Angular encoder resolution.

The raw measurement results were processed by means of a harmonic analysis, slightly differing from the standard one. The main steps are:

- Every shaft turn is considered like an elementary unit.
- The harmonic analysis is carried out on the points acquired in a elementary unit.
- The synthetic parameters, mean and standard deviation, are computed on the harmonics evaluated at each elementary unit.

In particular, the analysis results were focused on:

- Main Field normal component, $B_1$ in Tesla;
- Sextupole normal component, $b_3$ in UNITS;
- Decapole normal component, $b_5$ in UNITS;
- $11^{th}$ harmonic, $b_{11}$ in UNITS.

### 7.6 Data Analysis

The rational to take into account the aforementioned harmonics $B_1$, $b_3$, and $b_5$ is to highlight the behavior of the field components mainly affecting the LHC operation. The $b_{11}$ in a LHC dipole usually takes value about 0.6 UNITS [Sammut, 4/2006], thus this value is used as a reference to check the measurement results.

The mean is assumed to be the estimation of the harmonic:

*Normal main dipole field*
\[ B_1(k) = \frac{1}{N} \sum_{j=1}^{N} B_{j,1}(k) \quad (eq. 7.1) \]

*Normal harmonic of order \( n \)*

\[ b_n(k) = \frac{1}{N} \sum_{j=1}^{N} b_{j,n}(k) \quad (eq. 7.2) \]

where \( k = 1, \ldots, 14 \) is the measurement session defined by the row in Table VII.1, \( j = 1, \ldots, N \) is the number of turn of the coil shaft, and \( n = 1, 3, 5, 11 \) is the harmonic order taken into account. The experimental standard deviation is computed as:

*Normal main dipole field*

\[ \sigma(B_1(k)) = \sqrt{\frac{1}{N(N-1)} \sum_{j=1}^{N} \left[ B_{j,1}(k) - B_1(k) \right]^2} = \frac{\bar{\sigma}(B_1(k))}{\sqrt{N}} \quad (eq. 7.3) \]

*Normal harmonic of order \( n \)*

\[ \sigma(b_n(k)) = \sqrt{\frac{1}{N(N-1)} \sum_{j=1}^{N} \left[ b_{j,n}(k) - b_n(k) \right]^2} = \frac{\bar{\sigma}(b_n(k))}{\sqrt{N}} \quad (eq. 7.4) \]

\( \bar{\sigma} \) is the estimated standard deviation.

The repeatability of the measurement station is then assessed as 3-time the standard experimental deviation \( \sigma \) of the harmonics through the single turn.

**7.7 Results**

In this Section, the results from the three DC measurement categories defined in Table 7.1 are reported. In particular

1. Scan on the angular speed.

2. Cross-check between angular speed and gain.

3. Repeatability on single turn acquisition is detailed.
Scan on the angular speed

The aim of the measurements at several angular speeds is to verify the behaviour of the harmonics as a function of the rotation speed of the coil shaft. The number of experiments used to compute the mean and repeatability value of the harmonics depends on the angular speed. The number of turns of the shaft in the measurement time interval is equal to the number of elementary units employed in the analysis (e.g. angular speed 6.28 rad/s = 1 turn/s, measurement interval 2.7 min, number of turn = 1 turn/s*127 s = 127 turn).

In Fig. 7.5, the means of B₁ component with a ± 3σ bar versus angular speed is shown; the mean value of the main field varies between the maximum value 1.0655188 T at 6.28 rad/s and the minimum one 1.0654698 T at 43.96 rad/s. The difference between such values is about 49 μT, which is compatible with the value of the uncertainty [3*σ(B₁(k))] of the data from the overall session of measurements, i.e. about 33 μT. Therefore, any specific trend on mean values is highlighted.

![Figure 7.5: Main field component of LHC dipole measured versus several angular speed with fixed gain.](image)
Fig. 7.6 shows the standard deviation of the mean, $\sigma$, versus angular speed. Such values ensure a small dispersion of data, 2 $\mu T$, which is equivalent to a variation on the voltage signal in input to the FDIs of few tens $\mu V$. Yet, $\sigma (B_1)$ seems to improve for speed higher than 25.12 rad/s.

It is worth to note that the computed values of $\sigma (B_1)$ are normalized by the number of turns, which grows with the angular speed, because $\sigma$ is evaluated on the same time interval. Thus, a plot Fig. 7.7 and Fig. 7.8, of $\sigma (B_1)$ versus the same number of turn is needed (that means different measurement time) in order to highlight the independency of the repeatability with respect to the number of turns. Fig. 7.7a shows $\sigma (B_1)$, evaluated on the basis of the same number of turns (N=127). In this case, the measurement time decreases with the angular speed. As a further comparison, Fig. 7.8 reports the $\bar{\sigma}$ is not normalized by the number of turns (eq. 7.2) of the same measurement time.

![Figure 7.6: Standard deviation of the $B_1$ mean versus angular speed](image)

It is noted that $\bar{\sigma}$ slightly increases as a function of the angular speed. However the increment is only 9 $\mu T$.

In Fig. 7.9 and Fig. 7.10, the mean sextupole and decapole normal components, with a $\pm 3 \bar{\sigma}$ bar, versus the angular speed are shown. The first evidence from the
plots is a functional dependence of the two field components by the angular speed.

The difference between the maximum and minimum values is about 0.069 UNITS for \( b_3 \) and 0.016. UNITS for \( b_5 \). By increasing the angular speed, the amplitude of the coil signal increases. This could affect the accuracy of \( B_1, b_3, \) and \( b_5 \) measurement. However, further investigations are needed.

---

**Figure 7.7:** \( \sigma_i (B_1) \) as a function of angular speed (N variable) and time interval (N=127)

The plot shows same behavior of \( \sigma_i (B_1) \) on both cases, the differences are due to variation of the number of experiments carried out to compute the standard deviation of the mean.

**Figure 7.8:** \( \overline{\sigma} (B_1(k)) \) as a function of the angular speed over the same measurement time
Figures 7.11 and 7.12 show the standard deviation of the mean values of \( b_3 \) and \( b_5 \) turning out to be less than 0.0002 UNITS.

Figure 7.9: Sextupole component of LHC dipole versus angular speed at fixed FDI's gain

Figure 7.10: Decapole component of LHC dipole versus angular speed at fixed gain
In order to ensure proper results of the measurements, in Fig. 7.13 the behavior of the 11\textsuperscript{th} harmonic at the specified angular speed is shown. The 11\textsuperscript{th} harmonic is about 0.6 UNITS, in agreement to its typical value in an LHC dipole.

Fig. 7.14 and Fig.7.15 point out the normal and skew components of harmonics - from \( b_2 \) to \( b_{11} \) and from \( a_2 \) to \( a_{11} \) – like a function of elementary
units (acquisition on a single turn). A suitable level of stability and repeatability of the measurement system is proved.

(ii) Cross-check between angular speed and gain

The optimal operating conditions of FDI can be achieved by feeding up its Analog Digital Converter (ADC) with a fullscale signal. The parameters to be adjusted in order to ensure this condition are the angular speed of the shaft rotation and the gain of the FDIs. Only the gain of absolute signal was changed. The compensated signal amplitude at the measurement current of 1500 $A$, cannot reach the ADC full scale by applying the maximum gain and the maximum speed. Then, the gain is kept at its maximum value (100).
Figure 7.14: Normal components of the magnetic field in the second aperture of the MBBR 2427, measured with continuous acquisition at angular speed of 52.26 rad/s on the 5th segment of the new coil shaft, at 1500 A.

Figure 7.15: Skew components of the magnetic field in the second aperture of the MBBR 2427, measured with continuous acquisition at angular speed of 52.26 rad/s on the 5th segment of the new coil shaft, at +1500 A.

MDSL allows the above parameters to be easily changed in the user script. To understand the effects of the trade-off between the mechanical (effect of the speed increasing) and the electronic gain changing, four measurement tests were carried out.
The angular speeds taken into account were 6.28, 12.56, 18.84, 25.13 rad/s, and the FDI gains for the absolute signal were respectively 10, 5, 4, 4. In such conditions the signal amplitude at ADC after the PGA is of Volts’ order. Such as done for the measurements at several angular speeds, the field components, B1, b3, b5, and b11, were depicted in the plots. In Fig. 7.16, the main components of the induction field versus the angular speed are shown, with a $\pm 3\sigma$ bar.

![Figure 7.16: Main field component of LHC dipole measured versus (Angular Speed, Gain) with fixed samples per turn and supply current.](image)

The behaviour of the main field $B_1$ versus (Angular Speed, gain) show a dispersion growing slightly according to the angular speed. In Fig. 7.17, the values of $B_1$ for the two case studies at: FDI fixed and variable gain are shown. Their comparison highlights the compatibility among the two cases, and independence of the average field on the parameters. A first consideration is that the FDIs gain increasing at low speed can allow a better operation mode for the system. Less mechanical disturbs assuring a better using of the low noise FDIs amplifier. In Fig. 7.18, $\sigma(B_1)$ as a function of angular speed for fixed and variable FDIs gain is depicted.
In Fig. 7.19, the sextupole normal components of the magnetic induction field for fixed and variable FDI gain conditions are depicted. The effect of the electronic gain increasing is evident. In Fig. 7.20, the values of $\sigma(b_3)$, for fixed and variable gain measurement experiments, are depicted. Gain variations do not influence heavily the deviation of the mean, being 0.000019 UNITS the maximum difference. The standard deviations of $b_3$ are less than 0.00018 UNITS then, the average values show a good repeatability.

The decapole components (Fig. 7.21) show a similar behavior.

Figure 7.17: Main field component of LHC dipole measured versus several angular speed with fixed and variable gain.

Figure 7.18: Standard deviation of the $b_3$ mean versus angular speed
Figure 7.19: Sextupole component of LHC dipole measured for fixed gain (red) and variable gain (black), versus Angular Speed; a ± 3σ bar is displayed.

Figure 7.20: σ(b3) as a function of angular speed and measurement condition
The 11th component confirms the system measurements (Fig. 7.22). Figures 7.23 and 7.24 show the normal and skew components from 2nd to 11th order of fourteen measurement carried out at 25.12 rad/s and gain 4. The harmonics are characterised by high repeatability and stability.

(iii) Repeatability on single turn acquisition
The other measurement procedure to check the repeatability is based on the acquisition of 30 single turns at a constant speed of 50.24 rad/s. The time interval between two sequential turns is about 5 s.

In Fig 7.25, the mean of $B_1$ is plotted versus the two FDI's gain used to carry out the 30 measurements of a single turn. The values are compatible and the repeatability is hold is 2.2 $\mu$T.

In Figures 7.26 and 7.27, the normal and skew components versus the gain up to 11th are shown with a $\pm 3\sigma$ bar. Both the plots summaries that the measurements at two different gains are compatible. The overall repeatability is about 0.03 UNITS.

![Figure 7.23](image)

Figure 7.23: Normal components of the magnetic field in the second aperture of the MBBR 2427, measured with continuous acquisition at angular speed of 25.12 rad/s on the 5th segment of the new coil shaft, at 1500 A.
Figure 7.24: Skew components of the magnetic field in the second aperture of the MBBR 2427, measured with continuous acquisition at angular speed of 25.12 rad/s on the 5th segment of the new coil shaft, at +1500.

Figure 7.25: Mean value of $B_1$ over 30 measurements, a ± 3 $\bar{\sigma}$ bar is displayed.
7.8 **Standard AC measurement for field quality**

The aim of the field quality measurements on a LHC dipole is to confirm the empirical field model used in the main control system of the LHC operation (FIDEL) [Sammut, 9/2006], with the new fast acquisition equipment. Such empirical model can be validated by using the data from several measurements.
cycle, namely “Loadline” and “LHC cycle” (also known as the “standard machine cycle”).

The Loadline cycle is employed in order to compute the DC magnetization terms of the field model. The LHC cycle aims at giving comprehensive data on the long term dynamic effects in a superconducting magnet (“decay” and “snapback”). The first measurement cycle taken into account was the standard LHC cycle. In the following are presented:

1. The machine cycle.
2. A description of the measurement procedure.
3. Analysis and results of the decay and snapback.

### 7.8.1 LHC machine cycle

Fig. 7.28 shows the standard LHC machine cycle. After a suitable pre-cycle the injection phase at the current of $I_{inj} = 760 \, A$ lasts $1000 \, s$. The particles are then accelerated and the magnet is ramped up to the nominal current of $I_{nominal} = 11850 \, A$, achieving a nominal dipole field of 8.33 T. The ramp current follows a Parabolic-Exponential-Linear-Parabolic (PELP) profile.

In the standard cold test program [Sanfilippo, 2002], the above LHC cycle is always preceded by a magnet training quench (a quench occurs when a part of the magnet coil passes from the superconducting to the resistive state due to the internal field; a training is a series of controlled provoked at several value of current caused by warming the magnet) and a pre-cycle to put the magnet in a well know magnetic state (such procedure aims to erase the magnetic powering history). Table VII.2 reports the parameters of the pre-cycle applied in the qualification of the MDSL prototype, where $I_{min} = 350 \, A$ is the minimum value of the supply current during the cycle.
Figure 7.28: The standard reference machine cycle

<table>
<thead>
<tr>
<th>Parameter: ram-up</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final current</td>
<td>$I_{\text{nominal}}$</td>
<td>A</td>
</tr>
<tr>
<td>Acceleration</td>
<td>2.5</td>
<td>A/s²</td>
</tr>
<tr>
<td>Linear ramp rate</td>
<td>50</td>
<td>A/s</td>
</tr>
<tr>
<td>Deceleration</td>
<td>2.5</td>
<td>A/s²</td>
</tr>
<tr>
<td>Exponential start time</td>
<td>0</td>
<td>s</td>
</tr>
<tr>
<td>Parameters: Plateau</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Duration</td>
<td>300</td>
<td>s</td>
</tr>
<tr>
<td>Current level</td>
<td>$I_{\text{nominal}}$</td>
<td>A</td>
</tr>
<tr>
<td>Parameter: ram-down</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final current</td>
<td>$I_{\text{min}}$</td>
<td>A</td>
</tr>
<tr>
<td>Acceleration</td>
<td>2.5</td>
<td>A/s²</td>
</tr>
<tr>
<td>Linear ramp rate</td>
<td>50</td>
<td>A/s</td>
</tr>
<tr>
<td>Deceleration</td>
<td>2.5</td>
<td>A/s²</td>
</tr>
<tr>
<td>Exponential start time</td>
<td>0</td>
<td>s</td>
</tr>
</tbody>
</table>

Table VII.2: Parameter for the power supply of the pre-cycle phase

Table VII.3 reports the setting parameters for the supply current during the simulated machine cycle.
Table VII.3: Parameters for the power supply of the standard machine cycle

<table>
<thead>
<tr>
<th>Parameter: ram-up from $I_{\text{min}}$ to $I_{\text{inj}}$</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final current</td>
<td>$I_{\text{injection}}$</td>
<td>A</td>
</tr>
<tr>
<td>Acceleration</td>
<td>2</td>
<td>A/s$^2$</td>
</tr>
<tr>
<td>Linear ramp rate</td>
<td>10</td>
<td>A/s</td>
</tr>
<tr>
<td>Deceleration</td>
<td>2</td>
<td>A/s$^2$</td>
</tr>
<tr>
<td>Exponential start time</td>
<td>0</td>
<td>s</td>
</tr>
</tbody>
</table>

| Parameters: Plateau at injection |
|---|---|---|
| Duration | 1000 | s |
| Current level | $I_{\text{injection}}$ | A |

<table>
<thead>
<tr>
<th>Parameter: ram-up from $I_{\text{inj}}$ to $I_{\text{nominal}}$</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final current</td>
<td>$I_{\text{nominal}}$</td>
<td>A</td>
</tr>
<tr>
<td>Acceleration</td>
<td>$9 \times 10^{-3}$</td>
<td>A/s$^2$</td>
</tr>
<tr>
<td>Linear ramp rate</td>
<td>10</td>
<td>A/s</td>
</tr>
<tr>
<td>Deceleration</td>
<td>0.5</td>
<td>A/s$^2$</td>
</tr>
<tr>
<td>Exponential start time</td>
<td>325</td>
<td>s</td>
</tr>
</tbody>
</table>

| Parameters: Plateau at nominal |
|---|---|---|
| Duration | 300 | s |
| Current level | $I_{\text{nominal}}$ | A |

<table>
<thead>
<tr>
<th>Parameter: ram-down from $I_{\text{nominal}}$ to $I_{\text{min}}$</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final current</td>
<td>$I_{\text{min}}$</td>
<td>A</td>
</tr>
<tr>
<td>Acceleration</td>
<td>2</td>
<td>A/s$^2$</td>
</tr>
<tr>
<td>Linear ramp rate</td>
<td>10</td>
<td>A/s</td>
</tr>
<tr>
<td>Deceleration</td>
<td>2</td>
<td>A/s$^2$</td>
</tr>
<tr>
<td>Exponential start time</td>
<td>0</td>
<td>s</td>
</tr>
</tbody>
</table>

### 7.8.2 Measurement Procedure

The measurement of the magnetic flux in the magnet bore during a LHC cycle, namely from the injection plateau up to about $I_{\text{nominal}}$, allows dynamic features of the superconducting magnet, such decay and snapback to be observed. In particular, the snapback is a fast phenomenon, thus the acquisition system has to deliver data as soon as possible in order to evaluate it accurately.
With this in mind, the qualification of the platform was set as following:

- covered current cycle: from the last phase of the pre-cycle ramp down (3000 A) to about the middle of the machine cycle ramp-up (6000 A);
- 4 FDIs were employed in order to acquire the signals absolute and compensated from the coil segments 5th and 6th of the new shaft;
- a FDI was employed to measure the current;
- time interval of the measurement: 2000 s;
- the chosen speed is 50.24 rad/s (8 turn/s maxim value);
- samples per turn: 128.

It is important to remark that the above setting parameters define a new limit for the rotating coil measurement with respect of the standard one.

### 7.8.3 Analysis decay and snapback

As shown in the last years, the LHC superconducting magnets are characterized, during the phase of particle injection and subsequent ramp-up for the beam acceleration, by a drift and snapback of the sextupole (b3) and decapole (b5) components of internal field [Ambrosio, 2005].

The field model, used to describe the different contribution on the generated field, associates the decay and snapback phenomena as a AC dynamic effect. The behaviour of b3 and b5 depends on supply current, ramp rate, and powering history of the superconducting magnet. These phenomena are highlighted by emulating LHC machine cycle has to be carried out.

### 7.8.4 Results

In Fig. 7.29, the results of the harmonic analysis of the data from the MDSL qualification tests on the MBBR 2427 at SM18 are depicted: (1) main field component B1 and current versus time, (2) b3 normal sextupole component
versus current, (3) $b_3$ normal decapole component versus current. The decay and snapback are highlighted in (2) an (3).

The analysis of snapback was focused on the $b_3$ normal sextupole component. In particular, the exponential model to fit the measured values of the snapback is:

$$b_3^{\text{snapback}}(I) = \Delta b_3 e^{-\frac{1}{\Delta I}}$$

(eq. 7.5)

The $\Delta b_3$ and $\Delta I$ are the model parameters which have to be computed.

The decay and snapback were extrapolated by means of the measured harmonic, a polynomial fitting of 6th order was employed to interpolate the named base line $b_3$ ($b_3^{\text{baseline}}$), as would be measured with no plateau at injection phase. The couple of current interval used to compute the base line are: [650,750], [850,870]. In Figures 7.30, the base line extracted from the data and the interpolating polynomial function is sown; while in Fig. 7.33, the $b_3$ component superposed to the base line is depicted.

![Graph](image)

Figure 7.29: Main dipole field in the second aperture of the MBBR 2427 in the FFMM qualification test, LHC cycle (top); supply current measured by FDI (bottom).

In the current range considered, the base line fit does not show strong deviation from the real base line get by the measured data (Fig. 7.31).
As the base line is available, the decay and snapback of $b_3$ are isolated:

$$b_3^{\text{decay,snapback}} = b_3 - b_3^{\text{baseline}}$$  \hspace{1cm} (eq. 7.6)

The careful base line fitting enable to look only to the desired $b_3$ behaviours, $b_3^{\text{decay,snapback}}$, as showed in Fig. 7.33,

The $b_3^{\text{decay,snapback}}$ is used to compute the parameters for the exponential model of the snapback, by using the minimum square error method from about ($>)$ 760 A up to 870 A. Fig. 7.34 and Fig. 7.35 show respectively the snapback and the exponential fitting in decimal and logarithm scale.

In order to ensure the correctness of the analysis, the correlation $\beta_3$ factor between $\Delta b_3$ and $\Delta I$, defined as $\Delta b_3=\beta_3 \Delta I$, was computed. $\beta_3$ results to be 0.1834 $\text{UNITS}/A$, which is a value in according to the previous measurements carried out by the Hall-Plate based measurement system [Sammut, 4/2006], [Sammut, 9/2006].
Figure 7.31: Normal decapole as a function of the supply current. The decay and snapback phase are highlighted.

Figure 7.32: $b_3$ base line (blue) and polynomial fitting curve (red)
Figure 7.33: $b_3$ Normal Sextupole (blue) superposed to the fitted $b_3$ base line

Figure 7.34: $b_3^{\text{decay, samphach}}$ obtained by taking off the fitted base line curve
Figure 7.35: b3 snapback measured in the second aperture of the MBBR 2427 during a standard LHC cycle for FFMM qualification test (blue), exponential fitting model computed from data (red).

Figure 7.36: logarithm scale plot of the b3 snapback (blue) and the exponential fitting (red)
Conclusions

This thesis work has been devoted to introduce on the Measurement Domain Specific Language (MDSL) in FFMM which provide easy and flexible way to design software for magnetic measurement applications. The definition of test procedures, for the synchronization of the measurement tasks, and for the configuration of instruments is proposed.

FFMM has been developed with the aim of helping the user to write high quality code, in terms of flexibility, reusability, portability and efficiency. The test engineer needs to provide a formal description of the measurement procedure (script), in order to automatically generate executable measurement applications.

The formal description of the measurement procedure is to be provided in C++, and therefore requires knowledge of this programming language and its rules. In this thesis, a new easy Measurement Domain Specific Language (MDSL) is proposed. Such a language models the domain of interest and provides the user with easy programming tools capable of describing the measurement application, including specialized constructs concerning the automation of measurement procedures is proposed.

It provides not skilled programmers with a means for producing concise and bug free specific measurement applications.

The results have shown at the software level advantages in terms both of accuracy and dynamic performance, as well as ease of use, maintainability and reusability.

Future work will be devoted to improve the existing software tools to cover more application scenarios. Furthermore, an intensive plan of magnetic measurements is planned in order to keep exploring the superconducting magnet behaviour by means of the new platform. A new Graphical User Interface will be also developed both for the test endineer and final user. Finelly, an intensive
improvement of the existing software tools to cover more application scenarios will be carried out.
Appendix A

Script for magnetic bipole measurement in MDSL

BEGIN_SCRIPT main_Permeability_measurement:

//************************************
//Variable declaration
//************************************
DEF_VAR Encoder_slot AS int =13;
DEF_VAR Encoder_bus AS int = 4;
DEF_VAR Encoder_Channel AS int = 1;
DEF_VAR Encoder_mode AS int = 1;
DEF_VAR Encoder_freq AS float = 2048;
DEF_VAR Multimeter_intfNum AS int = 0;
DEF_VAR Multimeter_busAddress AS int = 0;
DEF_VAR Multimeter_timeout AS int = 0;
DEF_VAR numberOf_FDI AS int = 2;
DEF_VAR srcesStop AS int = 1;
DEF_VAR Cluster_abs_gain_ AS float = 1.0;
DEF_VAR Cluster_comp_gain_ AS float = 1.0;
DEF_VAR SamplePerTurn AS int = 1024;
DEF_VAR numberOfTurn AS int = 4;
DEF_VAR AcquisitionBufferSize AS int = 0;
DEF_VAR Daq_channel_name AS string = "AO_Ch";
DEF_VAR Daq_task_name AS string = "Trap_G";
DEF_VAR Daq_channel AS int = 0;
DEF_VAR Daq_timeout AS int = 200;
DEF_VAR Daq_generatioMode AS int = 0;
DEF_VAR Daq_sample_rate AS float = 1000;
DEF_VAR Daq_maxVolt AS int = 10;
DEF_VAR epsC AS float = 0.1;
DEF_VAR measurementCycle AS int = 0;
DEF_VAR spt AS int = 0;
DEF_VAR spt2 AS int = 0;
DEF_ARRAY Cluster_bus OF int [2]={4,4};
DEF_ARRAY Cluster_slot OF int [2]={11,12};
DEF_ARRAY plateaux OF float [38]={ 0, -0.1,-0.01,0.1,-0.02,0.2,-0.3,0.3,-
0.4,0.4,-0.5,0.5,-0.6,0.6,-0.7,0.7,-0.8,0.8,-0.9,0.9,-1,1,-1.2,1.2,-1.4,1.4,-
1.6,1.6,-1.8,1.8,-2,2,-3,3,-5,5,-10,10};

//************************************
//Variable assignement
//************************************
AcquisitionBufferSize = numberOf_FDI*( SamplePerTurn/2)*4*2;

//************************************
// Device Definition
//************************************
DEF ENCODER_BOARD: Enc_B WITH ( "1", "1", "CERN" ) ;
DEF FDI_CLUSTER: Cluster_1 WITH (numberOf_FDI ) ;
DEF KEITHLEY2K: Mult_M WITH ( "1", "2", "NI" ) ;
DEF DAQMX: NI_Daq WITH ( "1", "2", "NI" ) ;

//************************************
// Device Configuration
//************************************
CFG ENCODER_BOARD: Enc_B WITH ( Encoder_bus , Encoder_slot ) ;
CFG FDI_CLUSTER: Cluster_1 WITH ( Cluster_bus , Cluster_slot ) ;
CFG KEITHLEY2K: Mult_M WITH ( Multimeter_intfNum, Multimeter_busAddress, Multimeter_timeout );
CFG DAQMX: NI_Daq WITH ( Daq_channel_name, Daq_task_name, Daq_channel, Daq_timeout, Daq_generatioMode );
// Device Setting

://**********************************************************************//*
CMD FDI_CLUSTER: Reset ( Cluster_1, 0);
CMD FDI_CLUSTER: Reset ( Cluster_1, 1);
spt1 = (SamplePerTurn*numberOfTurn);
spt2 = SamplePerTurn/2;
SET FDI_CLUSTER: Params2 ( Cluster_1, spt1, SamplePerTurn, Cluster_abs_gain_, Cluster_comp_gain_, CONT, 500000, spt2,10);
SET FDI_CLUSTER: Stop Source ( Cluster_1, sourceStop);

:// SET ENCODER_BOARD: Synthetic_Trigger( Encoder_Channel , Encoder_mode,
Encoder_freq ) ;
SET DAQMX: VoltageRangeOutputChannel ( NI_Daq, Daq_minVolt, Daq_maxVolt);

CPP_CODE_START
" Multi_M->setMeasurementFunction(Func::DCV); 
CPP_CODE_END

CMD FDI_CLUSTER: Calibrate_Gain (Cluster_1, 0, 1.0);
CMD FDI_CLUSTER: Calibrate_Gain (Cluster_1, 1, 1.0);

://**********************************************************************//*
// Measurement Task Definition

BEGIN_MTASK test_da_cancellare:
FOR i = 1 TO 5 :
  spt = SamplePerTurn*numberOfTurn;
ENDFOR
END_MTASK

BEGIN_MTASK Demagnetization_Procedure:

// Task variable and array declaration
DEF_VAR old_plateau AS float;
DEF_VAR plateau AS float = 2.5;
DEF_VAR timePlateau AS float = 4;
DEF_VAR value1 AS float = 0.5;
DEF_VAR value2 AS float = 0.021;
DEF_VAR PS_resolution AS float;
DEF_VAR demagnetized AS int = 0;
DEF_VAR numOfSamples AS int;

// Task actions
PS_resolution = 20/65536;
USE DAQMX: NI_Daq;
BEGIN_CODE
  double* signal;
signal = NI_Daq->createPlat(0,0.01,Daq_sample_rate,&numOfSamples);
  double* plateau = 2.5;
  double* timePlateau = 4;
  double* value1 = 0.5;
  double* value2 = 0.021;
  double* PS_resolution = 20/65536;
  double* demagnetized = 0;
  double* numOfSamples = 0;
END_CODE

WHILE (demagnetized==0):
  double* signal = NI_Daq->createPlat(plateau,timePlateau,Daq_sample_rate,&numOfSamples);
  UPDATE DAQMX:
      Timing_Trigger ( NI_Daq, Daq_sample_rate, 0, numOfSamples);
      Start_Voltage ( NI_Daq, signal, numOfSamples );
      Wait_Generation ( NI_Daq );
      while (demagnetized==0):
        double* plateau = value1;
        double* timePlateau = value1/1.5;
        double* signal = NI_Daq->createPlat(plateau,timePlateau,Daq_sample_rate,&numOfSamples);
        UPDATE DAQMX:
            Timing_Trigger ( NI_Daq, Daq_sample_rate, 0, numOfSamples);
            Start_Voltage ( NI_Daq, signal, numOfSamples );
            Wait_Generation ( NI_Daq );
            while (demagnetized==0):
ELSEIF (plateau >= value2):
    old_plateau = plateau;
    plateau = plateau/1.2;
ELSE:
    old_plateau = plateau;
    plateau = plateau/1.1;
ENDIF

CPP_CODE_START " signal = NI_Daq->createRamp(old_plateau, -plateau, 1.5, Daq_sample_rate, &numOfSamples); " CPP_CODE_END
SET DAQMX: Timing_Trigger (NI_Daq,Daq_sample_rate, 0, numOfSamples);
CMD DAQMX: Start_Voltage (NI_Daq, signal, numOfSamples );
CMD DAQMX: Wait_Generation (NI_Daq);

CPP_CODE_START " signal = NI_Daq->createPlat(-plateau,timePlateau,Daq_sample_rate,&numOfSamples); " CPP_CODE_END
SET DAQMX: Timing_Trigger (NI_Daq,Daq_sample_rate, 0, numOfSamples);
CMD DAQMX: Start_Voltage (NI_Daq, signal, numOfSamples );
CMD DAQMX: Wait_Generation (NI_Daq);
IF (plateau <= 0.001):
    demagnetized = 1;
ENDIF
ENDWHILE

CPP_CODE_START " signal = NI_Daq->createPlat(0,0.01,Daq_sample_rate,&numOfSamples); " CPP_CODE_END
SET DAQMX: Timing_Trigger (NI_Daq,Daq_sample_rate, 0, numOfSamples);
CMD DAQMX: Start_Voltage (NI_Daq, signal, numOfSamples );
CMD DAQMX: Wait_Generation (NI_Daq);
PRINT "Demagnetization completed"  ;
END_MTASK

BEGIN_MTASK
END_MTASK

BEGIN_MTASK
BEGIN_MTASK
BEGIN_MTASK
BEGIN_MTASK
BEGIN_MTASK
BEGIN_MTASK
BEGIN_MTASK
BEGIN_MTASK
BEGIN_MTASK
APPENDIX

DEF_VAR numOfSamples AS int;
measurementCycle=measurementCycle+1;
CPP_CODE_START "
  double* signal;
  std::cout<<signal;
  std::cout<<measurementCycle<<std::endl;
  std::cout<<measurementCycle<<std::endl;
" CPP_CODE_END
WAIT 200 ms;
IF (measurementCycle <= 37):
  TRIG_EVENT start_ramp;
ELSE:
  USE DAQMX: NI_Daq;
  USE FDI_CLUSTER: Cluster_1;
  USE ENCODER_BOARD: Enc_B;
  CMD DAQMX: ZeroOutput(NI_Daq);
  CPP_CODE_START "signal = NI_Daq-
  >createRamp(plateaux[measurementCycle], 0, 1.5, Daq_sample_rate, &numOfSamples);" CPP_CODE_END
  SET DAQMX: Timing_Trigger (NI_Daq, Daq_sample_rate, 0, numOfSamples);
  CMD DAQMX: Start_Voltage (NI_Daq, signal, numOfSamples);
  CMD DAQMX: Wait_Generation (NI_Daq);
  CMD FDI_CLUSTER: Stop_Acquisition (Cluster_1);
  WAIT 3000 ms;
  CMD ENCODER_BOARD: Stop_Syntetic_Trigger (Enc_B, Encoder_Channel);
  PRINT "End Permeability measurement session";
ENDIF
END_MTASK

BEGIN_MTASK Conversion:

CPP_CODE_START " DataConversionByn2Ascii(path_name.c_str(),(int)SamplePerTurn/2, numberOf_FDI, 0);" CPP_CODE_END
END_MTASK
ADD_TASK Demagnetization_Procedure ;
ADD_TASK AFTER_TASK Demagnetization_Procedure Flux_Measurement ;
ADD_TASK AFTER_EVENT start_ramp Begin_Measurement_Procedure ;
ADD_TASK AFTER_TASK Begin_Measurement_Procedure Set_Next_Measurement ;
ADD_TASK AFTER_TASK Flux_Measurement Conversion ;
END_SCRIPT

Appendix B

xTxt

////////////////////////////////////////////////////////////////////
// Basic syntax (START)
////////////////////////////////////////////////////////////////////

Script:
"BEGIN_SCRIPT" scriptName=ID "::*
 (scriptIdDeclarations+=Declarations)*
 (scriptIdAssignements+=ScriptAssignment)*
 (scriptIdDeviceDefinitions+=Definition_Statement)*
 (scriptIdDeviceConfigurations+=ConfigurationStatement)*
 (scriptIdDeviceSettings+=DeviceSetting)*
 (mtasks+=MTask)*
 (taskExecutionStatements+=TaskExecutionStatement)*
 // (rts=RunTaskSequence_)
"END_SCRIPT";

ScriptAssignement:
(ast=AssignStatement | cpp=CppCode);

DeviceSetting:
  cmd=CommandStatement | sets=SettingStatement | gets=GettingStatement |
  uses=Use_Statement |
  cppCode=CppCode | ast=AssignStatement;

142
APPENDIX

MTask:
"BEGIN_MTASK" mtaskName=ID (mtaskDesc=STRING)? ":"
(taskDeclarations+=Declarations)*
(taskAction+=GenericStatement)*
"END_MTASK";

GenericStatement:
cmds=CommandStatement | sets=SettingStatement | gets=GettingStatement |
uses=Use_Statement |
cpp=CppCode | ast=AssignStatement |
fst=ForStatement | wst=WhileStatement | ist=IfStatement |
ut=Util_Statement;

Util_Statement:
print=Print_ | delay=Delay_ | trigEvent=TrigEvent_;

TrigEvent_:
"TRIG_EVENT" eventName=ID ";";

TaskExecutionStatement:
at=AddTask_ | atat=AddTaskAfterTask_ | atae=AddTaskAfterEvent_;

AddTask_:
"ADD_TASK" taskName=ID ";";

AddTaskAfterTask_:
"ADD_TASK_AFTER_TASK" task1Name=ID task2Name=ID ";";

AddTaskAfterEvent_:
"ADD_TASK_AFTER_EVENT" eventName=ID taskName=ID ";";

Print_:
"PRINT" text=STRING ";";

Delay_:
"WAIT" time=T_INT "ms" ";";

ForStatement:
"FOR" varName=ID="startValue=T_INT "TO" finalValue=T_INT ";" |
(forStatements+=GenericStatement)*
"ENDFOR";

WhileStatement:
"WHILE" cond=Expression ";" |
(whileStatements+=GenericStatement)*
"ENDWHILE";

IfStatement:
"IF" cond=Expression ";" |
(ifStatements+=GenericStatement)*
("ELSEIF" elseIfCond=Expression ":" (elseIfStatements+=GenericStatement)*)|
("ELSE:" (elseStatements+=GenericStatement)*)|
"ENDIF";

CppCode:
"CPP_CODE_START"
code=STRING
"CPP_CODE_END";

////////////////////////////////////////////////////////////
///     Assignment   ///////////////////////////////////////
////////////////////////////////////////////////////////////

AssignStatement:
varName=ID "=" (singAss=SingleAssignStatement | arrAss=ArrayAssignStatement |
funcAss=FuncAssignStatement);

SingleAssignStatement:
value=Expression ";";

ArrayAssignStatement:
"{" value1=Expression ("," value+=Expression)* "}" ";";

////////////////////////////////////////////////////////////
/// End Assignment   ///////////////////////////////////////
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
///     Declarations ///////////////////////////////////////
////////////////////////////////////////////////////////////

Declarations:
APPENDIX

```c
(vd=VarDeclarations | ad=ArrayDeclarations);
VarDeclarations:
  "DEF_VAR" name=ID "AS" type=DataType ("=" "value=Literal")? ";";
ArrayDeclarations:
  "DEF_ARRAY" name=ID "OF" type=DataType 
   "[" (size=T_INT)? "]" ("=" "[" 
   Value=Expression ("," Value2+=Expression)* "]")? ";";
/// End Declarations

Expression:
  exprval= EqualityExpr;
EqualityExpr:
  left=CondORExpr (op=EqualityOp right=CondORExpr)?;
CondORExpr:
  left=CondANDExpr (rights+=CondORRights)*;
CondORRights:
  op=OrOp right=CondANDExpr;
CondANDExpr:
  left=AtomicBoolExpr (rights+=CondANDRights)*;
CondANDRights:
  op=AndOp right=AtomicBoolExpr;
AtomicBoolExpr:
  rex=RelationalExpr;
RelationalExpr:
  left=AdditiveExpr (op=RelationalOp right=AdditiveExpr)?;
AdditiveExpr:
  left=MultiplicativeExpr (rights+=AdditiveRights)*;
AdditiveRights:
  op=AdditiveOp right=MultiplicativeExpr;
MultiplicativeExpr:
  left=AtomicExpr (rights+=MultiplicativeRights)*;
MultiplicativeRights:
  op=MultiplicativeOp right=AtomicExpr;
AtomicExpr:
  var=Variable_ | lit=Literal | parexp= ParenExpr;
Variable_:
  name=ID | arrayElement=ID("["index=T_INT"])?;
ParenExpr:
  "(" expr=EqualityExpr ")";
Param:
  var=ID | value=Literal;
Literal:
  intl=Integer_ | fltl=Float_ | strl=String_;
Native INT:
  ";"
Native T_INT:
  "('-')?('0'..'9')+('0'..'9')+
Float_:
  number=T_INT;
Integer_:
  number=T_INT;
String_:
  value=STRING;
Enum MultiplicativeOp: //Lv13
  TIMES = "*" | DIVIDE = "/" | MOD = "%";
Enum AdditiveOp: //Lv12
  PLUS = "+" | MINUS = "-";
Enum RelationalOp:
  LT = "<" | LE = "<=" | GT = ">" | GE = ">=";
Enum EqualityOp:
  EQ = "==" | NE = "!=";
Enum OrOp:
```
OR = "||";
Enum AndOp:
AND = "&&";
Enum DataType:
int="int"| short="short"| long="long"| float="float"| string="string";
="/\n
Basic syntax (END)
="/\n
Devices syntax (START)
="/\n
Definition Statement:
"DEF"
  | eb_defs = Def_EncoderBoard_
  | fdlc_defs = Def_FdiCluster_
  | key2_defs = Def_Keithley2k_
  | daq_defs = Def_DAQmx_
  | lvp_defs = Def_LVPowerSupply_
  | max_defs = Def_Maxon_Epos_
  | mmc_defs = Def_MidiMotorController_
  | omrk_defs = Def_OrientalMotorRK_
  | pcu_defs = Def_PCU2000_
  | powco_defs = Def_Power_Controller_

ConfigurationStatement:
"CFG"
  | eb_confs = Cfg_EncoderBoard_
  | fdlc_confs = Cfg_FdiCluster_
  | key2_confs = Cfg_Keithley2k_
  | daq_confs = Cfg_DAQmx_
  | lvp_confs = Cfg_LVPowerSupply_
  | max_confs = Cfg_Maxon_Epos_
  | mmc_confs = Cfg_MidiMotorController_
  | omrk_confs = Cfg_OrientalMotorRK_
  | pcu_confs = Cfg_PCU2000_
  | powco_confs = Cfg_Power_Controller_

CommandStatement:
"CMD"
  | eb_cmds=Cmd_EncoderBoard_
  | fdlc_cmds=Cmd_FdiCluster_
  | key2_cmds = Cmd_Keithley2k_
  | daq_cmds = Cmd_DAQmx_
  | lvp_cmds = Cmd_LVPowerSupply_
  | max_cmds = Cmd_Maxon_Epos_
  | mmc_cmds = Cmd_MidiMotorController_
  | omrk_cmds = Cmd_OrientalMotorRK_
  | pcu_cmds = Cmd_PCU2000_
  | powco_cmds = Cmd_Power_Controller_

SettingStatement:
"SET"
  | eb_sets = Set_EncoderBoard_
  | fdlc_sets = Set_FdiCluster_
  | key2_sets = Set_Keithley2k_
  | daq_sets = Set_DAQmx_
  | lvp_sets = Set_LVPowerSupply_
  | max_sets = Set_Maxon_Epos_
  | mmc_sets = Set_MidiMotorController_
  | omrk_sets = Set_OrientalMotorRK_
  | pcu_sets = Set_PCU2000_
  | powco_sets = Set_Power_Controller_

GettingStatement:
"GET"
  | ebGets = Get_EncoderBoard_
  | fdlcGets = Get_FdiCluster_
key2_Gets = Get_Keithley2k_
daq_Gets = Get_DAQmx_
lvp_Gets = Get_LVPowerSupply_
max_Gets = Get_Maxon_Epos_
mmc_Gets = Get_MidiMotorController_
omrk_Gets = Get_OrientalMotorRK_
pcu_Gets = Get_PCU2000_
powco_Gets = Get_Power_Controller_
);

Use_Statement:
"USE" {
    eb_uses = Use_EncoderBoard_
    fdic_uses = Use_FdiCluster_
    key2_uses = Use_Keithley2k_
    daq_uses = Use_DAQmx_
    lvp_uses = Use_LVPowerSupply_
    max_uses = Use_Maxon_Epos_
    mmc_uses = Use_MidiMotorController_
    oomrk_uses = Use_OrientalMotorRK_
    pcu_uses = Use_PCU2000_
    powco_uses = Use_Power_Controller_
}

Method_Signature_:
    cmd=ID "(" name=ID ("," params+=Param) ")";

_Exceptions_:

EncoderBoard Definition
Def_EncoderBoard_:
    "ENCODER_BOARD:" name=ID ("WITH"("mod=Param","ser_num=Param","man=Param")")?;
    static EncoderBoard* createDevice( std::string name );
    static EncoderBoard* createDevice( std::string name, std::string mod,
                                    std::string ser_num, std::string man);

Use_EncoderBoard_:
    "ENCODER_BOARD:" name=ID;
    "DAQmx* NI_Daq = DAQmx::getDeviceInstance(DAQmx);";

EncoderBoard Configuration
Cfg_EncoderBoard_:
    "ENCODER_BOARD:" name=ID ("WITH"("bus=Param","slot=Param
","sp=Param") (","(remap=Param)?(","(ac=Param))?)?)?)?";

EncoderBoard Command
Cmd_EncoderBoard_:
    "ENCODER_BOARD:" sig=Method_Signature_;
APPENDIX

Appendix C
xPand

«IMPORT mydsl»
«DEFINE main FOR Script»
«FILE scriptName + ".cpp"»
//Script name: «scriptName».cpp
#include "core/utils/DynamicParameter.h"
«FOREACH scriptDeviceDefinitions AS e-»
«EXPAND DevicesIncludeArea FOR e-»
«ENDFOREACH-»

// DECLARATION

// Script devices name
«FOREACH scriptDeviceDefinitions AS e-»
«EXPAND DevicesNameArea FOR e-»
«ENDFOREACH-»

// Script variable and array
«FOREACH scriptDeclarations AS var-»
«IF var.vd != null-»«EXPAND VarDeclarationArea FOR var.vd-»
«ELSEIF var.ad != null-»«EXPAND ArrayDeclarationArea FOR var.ad-»«ENDIF-»
«ENDFOREACH-»

BEGINSCRIPT

//************************************
//Dynamic parameters
//************************************
//Create the devices
//************************************

DEVICE_CREATION
«FOREACH scriptDeviceDefinitions AS e-»
«EXPAND DevicesCreationArea FOR e-»
«ENDFOREACH-»

END_DEVICE_CREATION

//************************************
//Configure the devices
//************************************

DEVICE_CONFIGURATION
«FOREACH scriptDeviceConfigurations AS e-»
«EXPAND DevicesConfigurationArea FOR e-»
«ENDFOREACH-»

END_DEVICE_CONFIGURATION

//************************************
//Set the devices
//************************************

SET_DEVICE
«FOREACH scriptDeviceSettings AS e-»
«EXPAND DeviceSettingArea FOR e-»
«ENDFOREACH-»

END_DEVICE_SETTING

//************************************
//Define tasks
//************************************

«FOREACH mtasks AS mtask -»
«EXPAND MTaskDefinitionArea FOR mtask -»
«ENDFOREACH-»

END_TASK_SETTING

//************************************
//Define execution graph
//************************************

//************************************
APPENDIX

```
«FOREACH taskExecutionStatements AS es -»
 «EXPAND TaskExecutionStatementArea FOR es -»
«ENDFOREACH»
RUN_TASK_SEQUENCE //Delete Devices
«FOREACH scriptDeviceDefinitions AS e-»
 «EXPAND DevicesDeleteArea FOR e-»
«ENDFOREACH»
ENDSCRIPT
ENDFILE
«ENDDEFINE»
«REM» --------------------------------------------------------  «ENDREM»
«REM» ------------------------ MTASKS ------------------------  «ENDREM»
«REM» --------------------------------------------------------  «ENDREM»
«REM» MTask Definition Area «ENDREM»
«DEFINE MTaskDefinitionArea FOR MTask»
/*«mtaskDesc»*/
BEGIN_TASK(«mtaskName»)
// Task variable and array
«FOREACH taskDeclarations AS var-»
 «IF var.vd != null -» «EXPAND VarDeclarationArea FOR var.vd-»
 «ELSEIF var.ad != null -» «EXPAND ArrayDeclarationArea FOR var.ad-» «ENDIF»
«ENDFOREACH»
// Task actions
«FOREACH taskAction AS e -»
 «EXPAND GenericStatementArea FOR e-»
«ENDFOREACH»
END_TASK
«ENDDEFINE»
«REM» Generic Statement Area «ENDREM»
«DEFINE GenericStatementArea FOR GenericStatement-»
«IF (fst != null) -» «EXPAND ForStatementArea FOR fst -»
 «ELSEIF (wst != null) -» «EXPAND WhileStatementArea FOR wst-»
 «ELSEIF (ast != null) -» «EXPAND AssignStatArea FOR ast-»
 «ELSEIF (ist != null) -» «EXPAND IfStatementArea FOR ist-»
 «ELSEIF (cmds != null)-» «EXPAND CommandStatementArea FOR cmds-»
 «ELSEIF (sets != null)-» «EXPAND GettingStatementArea FOR sets-»
 «ELSEIF (uses != null)-» «EXPAND Use_StatementArea FOR uses-»
 «ELSEIF (cpp != null)-» «EXPAND CppCodeArea FOR cpp-»
 «ELSEIF (ust != null)-» «EXPAND Util_StatementArea FOR ust-»
«ENDIF»
«ENDDEFINE»
«REM» Util Statement Area «ENDREM»
«DEFINE Util_StatementArea FOR Util_Statement-»
 «IF (print != null) -» «EXPAND Print_Area FOR print-»
 «ELSEIF (delay != null)-» «EXPAND Delay_Area FOR delay-»
 «ELSEIF (trigEvent != null)-» «EXPAND TrigEvent_Area FOR trigEvent-»
«ENDIF»
«ENDDEFINE»
«REM» Print Area «ENDREM»
«DEFINE Print_Area FOR Print_-»
environment->console->writeln("«text»");
«ENDDEFINE»
«REM» Delay Area «ENDREM»
«DEFINE Delay_Area FOR Delay_-»
delay{times};
«ENDDEFINE»
«REM» TrigEvent Area «ENDREM»
«DEFINE TrigEvent_Area FOR TrigEvent_-»
TRIG_EVENT{eventName};
«ENDDEFINE»
«REM» For Statement Area «ENDREM»
«DEFINE ForStatementArea FOR ForStatement-»
for ( int «varName» = «startValue»;
 «varName» <= «finalValue»; «varName»++)
 { «EXPAND GenericStatementArea FOR statements-»
````
APPENDIX

| «ENDDEFINE»
| «REM» While Statement Area «ENDREM»
| «DEFINE WhileStatementArea FOR WhileStatement-»
| while «EXPAND ExpressionArea FOR cond»
| { «EXPAND GenericStatementArea FOREACH whileStatements-»
| «ENDDEFINE»
| «REM» If Statement Area «ENDREM»
| «DEFINE IfStatementArea FOR IfStatement-»
| if «EXPAND ExpressionArea FOR cond»
| { «EXPAND GenericStatementArea FOREACH ifStatements-»
| «IF elseifCond!=null-»
| elseif «EXPAND ExpressionArea FOR elseifCond»
| { «EXPAND GenericStatementArea FOREACH elseifStatements-»
| «ENDIF-»
| «IF elseStatements.isEmpty!=true-»
| else
| { «EXPAND GenericStatementArea FOREACH elseStatements-»
| «ENDIF-»
| «ENDDEFINE»
| «REM» Cpp Code Area «ENDREM»
| «DEFINE CppCodeArea FOR CppCode-» «code»
| «ENDDEFINE»
| «REM» Add Task Area «ENDREM»
| «DEFINE AddTask_Area FOR AddTask_-»
| ADD_TASK(«taskName»)
| «ENDDEFINE»
| «REM» Add Task After Task Area «ENDREM»
| «DEFINE AddTaskAfterTask_Area FOR AddTaskAfterTask_-»
| ADD_TASK_AFTER_TASK(«task1Name»,«task2Name»)
| «ENDDEFINE»
| «REM» Add Task After Event Area «ENDREM»
| «DEFINE AddTaskAfterEvent_Area FOR AddTaskAfterEvent_-»
| ADD_TASK_AFTER_EVENT(«eventName»,«taskName»)
| «ENDDEFINE»
| «REM» Devices Include Area «ENDREM»
| «DEFINE DevicesIncludeArea FOR Definition_Statement-»
| «IF eb_defs!=null-» «EXPAND EncoderBoard.IncludeDirective FOR eb_defs-»
| «ELSEIF daq_defs!=null-» «EXPAND DAQ.IncludeDirective FOR daq_defs-»
| «ELSEIF fdic_defs!=null-» «EXPAND FdiCluster.IncludeDirective FOR fdic_defs-»
| «ELSEIF key2_defs!=null-» «EXPAND Keithley2k.IncludeDirective FOR key2_defs-»
| «ELSEIF lvp_defs!=null-» «EXPAND LVPowerSupply.IncludeDirective FOR lvp_defs-»
| «ELSEIF omrk_defs!=null-» «EXPAND OrientalMotorRK.IncludeDirective FOR omrk_defs-»
| «ELSEIF powco_defs!=null-» «EXPAND Power_Controller.IncludeDirective FOR powco_defs-»
| «ENDDEFINE»

149
«ENDIF »
«ENDDEFINE»

«REM» Devices Name Area «ENDREM»
«DEFINE DevicesNameArea FOR Definition_Statement->
«IF eb_defs!=null»«EXPAND EncoderBoard_NameDeclaration FOR eb_defs->
«ELSEIF daq_defs!=null»«EXPAND DAQ_NameDeclaration FOR daq_defs->
«ELSEIF fdic_defs!=null»«EXPAND FdiCluster_NameDeclaration FOR fdic_defs->
«ELSEIF key2_defs!=null»«EXPAND Keithley2k_NameDeclaration FOR key2_defs->
«ELSEIF lvp_defs!=null»«EXPAND LVPowerSupply_NameDeclaration FOR lvp_defs->
«ELSEIF max_defs!=null»«EXPAND Maxon_Epos_NameDeclaration FOR max_defs->
«ELSEIF mmc_defs!=null»«EXPAND MidiMotorController_NameDeclaration FOR mmc_defs->
«ELSEIF omrk_defs!=null»«EXPAND OrientalMotorRK_NameDeclaration FOR omrk_defs->
«ELSEIF pcu_defs!=null»«EXPAND PCU2000_NameDeclaration FOR pcu_defs->
«ELSEIF powco_defs!=null»«EXPAND Power_Controller_NameDeclaration FOR powco_defs->
«ENDIF»
«ENDDEFINE»

«REM» ---------------------------------------------------------  «ENDREM»
«REM» ---------------------------------------------------------  «ENDREM»
«REM» --------------------- ENCODER BOARD ---------------------  «ENDREM»
«REM» ---------------------------------------------------------  «ENDREM»

«DEFINE EncoderBoard_IncludeDirective FOR Def_EncoderBoard->
#include "core/devices/EncoderBoard.h"
«ENDDEFINE»

«REM» ENCODER BOARD: Name declaration «ENDREM»
«DEFINE EncoderBoard_NameDeclaration FOR Def_EncoderBoard->
std::string EncoderBoardName_<name> = "<name>";
«ENDDEFINE»

«REM» ENCODER BOARD: Creation methods «ENDREM»
«DEFINE EncoderBoard_CreationMethods FOR Def_EncoderBoard->
«IF mod==null»
EncoderBoard* EncoderBoardObject_<name> = EncoderBoard::createDevice(EncoderBoardName_<name>);
«ELSE»
EncoderBoard* EncoderBoardObject_<name> = EncoderBoard::createDevice(EncoderBoardName_<name>, "<name>");
«EXPAND ParamArea FOR mod->, «EXPAND ParamArea FOR ser_num->, «EXPAND ParamArea FOR man->);
«ENDIF»
«ENDDEFINE»
Appendix D
Architecture of FFMM
Bibliography


Foundation International Research Fellowship Program, Grant# OSIE -0502410, CERN AT-MTM, 2007.


[Capers, 2007] Capers Jones. Programming Languages Table, 1996.


