**MEASUREMENT-DOMAIN SPECIFIC LANGUAGE
FOR MAGNETIC TEST SPECIFICATIONS AT CERN**

P. Arpaia^{1,2}, M. Buzio², L. Fiscarelli^{1,2}, V. Inglese^{1,3}, G. La Commara¹, L. Walckiers²

A Measurement-Domain Specific Language (MDSL) for test procedure definition, measurement tasks synchronization, and instrument configuration is proposed. MDSL is a formal language specially designed for a specific domain of measurement and test, aimed at specifying complete, easy-to-understand, -reuse, and -maintain applications efficiently and quickly. Owing to MDSL constructs capability of abstracting key concepts of the domain, the test engineer can write more concise and higher level programs in shorter time without being a skilled programmer. The MDSL has been applied to the specifications of superconducting magnet tests of the Large Hadron Collider at CERN.

CERN-TE-2009-002
26/06/2009

1 Università del Sannio, Dipartimento di Ingegneria, Benevento, Italy

2 CERN, Technology Department, Geneva, Switzerland

3 Università degli Studi di Napoli – Federico II, Dipartimento di Ingegneria Elettrica, Napoli, Italy

Presented at the 2009 IEEE Instrumentation & Measurement Technology Conference - I²MTC 2009
5-7 May 2009, Singapore

Measurement-Domain Specific Language for Magnetic Test Specifications at CERN

Pasquale Arpaia^{1,2}, Marco Buzio², Lucio Fiscarelli^{1,2}, Vitaliano Inglese^{1,3}, Giuseppe La Commara¹, Louis Walckiers²

¹ Dipartimento di Ingegneria, Università del Sannio, Benevento, Italy, arpaia@unisannio.it

² Technology Department, Magnets, Superconductors and Cryostats, CERN, Genève, Switzerland, marco.buzio@cern.ch

³ Dipartimento di Ingegneria Elettrica, Università degli Studi di Napoli – Federico II, Napoli, Italy, vitaliano.inglese@cern.ch

Abstract—A Measurement-Domain Specific Language (MDSL) for test procedure definition, measurement tasks synchronization, and instrument configuration is proposed. MDSL is a formal language specially designed for a specific domain of measurement and test, aimed at specifying complete, easy-to-understand, -reuse, and -maintain applications efficiently and quickly. Owing to MDSL constructs capability of abstracting key concepts of the domain, the test engineer can write more concise and higher level programs in shorter time without being a skilled programmer. The MDSL has been applied to the specifications of superconducting magnet tests of the Large Hadron Collider at CERN.

Keywords—component; measurement, magnetic variables measurement, automatic test equipment.

I. INTRODUCTION

At CERN, the European Organization for Nuclear Research, the Large Hadron Collider (LHC) required a big effort for the test of the magnets and new requirements for magnetic measurements [1].

In view of future projects, a wide range of software requirements has been recently satisfied by the Flexible Frame-work for Magnetic Measurements (FFMM) [2], designed also for integrating more performing flexible hardware [3]. 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 [4].

Concise and bug-free specific applications have to be generated so that test engineers using FFMM do not have to be skilled programmers. On-field experience, at the CERN magnet test facility SM18 with the current FFMM release 2.0, highlights that a significant part of the ongoing operation costs are related to the development and maintenance of test applications [5].

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 [6]-[7]. These advantages are making DSLs very popular and their design and implementation is 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.

In this paper, A Measurement-Domain Specific Language (MDSL) for defining test procedures, synchronizing measurement tasks, and configuring instruments is proposed. In particular, in Section 2, the test domain of the FFMM at CERN is highlighted. In Section 3, the approach and the main components of the proposed DSL are illustrated. Finally, in Sections 4 and 5, the MDSL implementation for FFMM is described.

II. MAGNETIC TEST DOMAIN AND FFMM ARCHITECTURE

At CERN, during the series tests of superconducting magnets for the LHC, measurement systems were developed under different conditions and with variable requirements. The result is a number of systems whose software have 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 realization of FFMM was based on the following basic ideas: (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 re-using software modules; (ii) reusability is achieved by object-oriented approach and modularity: a suitable design of the code allows modules to be re-used; (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; and (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

Figure 1. The multi-layer FFMM architecture

through three main layers (Fig. 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 the concrete devices to be coordinated during measurement processes. Event handling was implemented to let Virtual Devices 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.

III. THE MEASUREMENT-DOMAIN SPECIFIC LANGUAGE

A. The proposed approach

The advantage of a DSL (domain-specific language) [8] 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 artefacts, actors and processes [9].

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.

A Domain-Specific Description (DSD) is a program written in a DSL. Such a program is compiled, interpreted, or analyzed by a Domain-Specific Processor. When the DSD is compiled, the output can be in textual format (e.g., another DSD, or a program in a general-purpose programming language), or in binary format. In our case the output is a set of C++ code files.

B. The requirements

Test engineers are not skilled programmers and have to produce concise and bug-free FFMM specific applications. Thus, a new Measurement 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.

C. The architecture

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. The semantic model is separated from the MDSL in order to:

- think about the semantics of this domain without getting tangled up in the DSL syntax or parse;
- be able to test the semantic model by creating objects in the model and manipulating them directly;
- have an incremental approach, starting with simple internal DSL and after add an external DSL; this is possible because having an explicit semantic model can support multiple DSLs, since both DSLs can parse easily into the same Semantic Model;
- 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; or new syntaxes can be experimented by just verifying the creation of the same objects in the 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.

Figure 2. MDSL transformation in code

In Fig. 2, the proposed approach for the transformation of the Measurement Domain-Specific Description (MDSL) into the final code is shown.

The external MDSL, written by the Test Engineer, is parsed to create an internal file treated by the semantic model (Fig. 2). The external DSL, the DSL scripts i.e. the MDSL, the parser, and the Semantic Model are very clearly separated. The MDSL scripts are written in a 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.

In Fig. 2, another benefit of using a Semantic Model is highlighted. 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.

D. 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:

```

events
  EncBoard_StartTrigger EB_ST
  MrController_StartRotation MC_SR
End

```

In this composite structure, 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 MDSL 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 MDSL than the words; it can be manipulated in many ways by walking up and down in the tree. Basically, the parser reads the textual MDSL, 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.

E. 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). 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 (i) produce models of the application and business logic, and (ii) 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) [10]. 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 artefacts, 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 artefacts are created in the early stages if 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 artefacts 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++.

IV. FFMM MDSL IMPLEMENTATION

FFMM MDSL implementation is based on meta-environment framework [11]. The Meta-Environment is a framework for language development, source code analysis and source code transformation consisting of:

- syntax analysis tools;
- semantic analysis and transformation tools;
- interactive development environment.

It is an open framework that:

- can be easily extended with third-party components;
- can be easily tailored, modified, or extended;
- is supported by an open source community.

The Meta-Environment is a generalization of the ASF+SDF Meta-Environment [12], successfully used in a wide variety of analysis, transformation and renovation

projects. ASF+SDF is intended for the high-level and modular description for the analysis and transformation of computer-based formal languages.

SDF [13]-[14] stands for Syntax Definition Formalism, and it's just that: a formalism to define syntax. It is a very modular formalism used to define lexical and context-free grammars alike, and it is supported by a parser generator.

ASF [15] stands for Algebraic Specification Formalism. In ASF we use SDF to describe the signature of terms. An SDF production is an ASF term constructor. A non-terminal in SDF is an algebraic sort in ASF. ASF specifications are collections of equations. The algebraic equations define which terms are equal to which other terms. This is where the algebra stops, and term rewriting begins. In fact, the equations are interpreted as rewrite rules. Each pattern on the left-hand side is searched in the input term, and replaced by the right-hand side.

Basically, in FFMM, ASF+SDF is used to define the syntax (form) and semantics (meaning) MDSL. The Syntax Definition Formalism (SDF) is used to define syntactic aspects including:

- Lexical syntax (keywords, comments, string constants, whitespace, ...).
- Context-free syntax (declarations, statements, ...).

The Algebraic Specification Formalism (ASF) is used to define semantic aspects such as:

- Type checking (are the variables that are used declared and are they used in a type-correct way?);
- Formatting (display the original program using user-defined rules for indentation and formatting);
- Fact extraction (extract all procedure calls or all declarations and uses of variables);
- Execution (run the program with given input values).

An ASF+SDF specification consists of a collection of modules. A module can import other modules. A single module has the following structure:

```
module ModuleName
  ImportSection*
  ExportOrHiddenSection*
  equations
  ConditionalEquation*
```

In the following, the module used to defines the two MDSL types, numeric and string, is shown:

```
module languages/MDSL/syntax/Types
exports
  sorts TYPE
  context-free syntax
  "numeric" -> TYPE
  "string" -> TYPE
```

As an example, the module used to define the identifier is reported:

```
module languages/MDSL/syntax/Identifiers
exports
  sorts IDENTIFIER
  lexical syntax
```

```
[a-z] [a-z0-9]* -> IDENTIFIER
context-free restrictions
IDENTIFIER -/- [a-z0-9]
```

In the following, the module used for defining the measurement task, intended as a block of simple measurement action, is shown:

```
module languages/MDSL/syntax/MeasureTask

imports languages/MDSL/syntax/Identifiers
imports languages/MDSL/syntax/Types
imports languages/MDSL/basic/NumericConst
imports languages/MDSL/basic/StringConst

exports
  sorts MEASUREMENT_TASK DECLS ID-TYPE STATEMENT EXP
  Context-free syntax
  "begin_task" DECLS {STATEMENT"," }* "end_task"
    -> MEASUREMENT_TASK
  "declare" {ID-TYPE "," }* ";" -> DECLS
  MDSL-ID ":" TYPE -> ID-TYPE

Context-free syntax
  MDSL-ID ":"= EXP -> STATEMENT

Context-free syntax

  MDSL-ID -> EXP
  NUMCONST -> EXP
  STRCONST -> EXP
  EXP "+" EXP -> EXP
  EXP "-" EXP -> EXP
  EXP "*" EXP -> EXP
  EXP "/" EXP -> EXP
  "(" EXP ")" -> EXP
```

As described before, SDF is used to describe syntax of (programming) languages using context-free production rules. The Parser is based on the collection of C and Java libraries provided by The Meta-Environment. The Parser produce parse trees. The nodes of the parse trees are the productions of the SDF specification. The leaves of the parse trees are the characters of the input text.

The parse trees produced by Parser represent our Semantic Model. ASF programs take parse trees as input and produce parse trees as output. A node in the parse tree is a "function" for ASF. All ASF does is replace functions by other functions, generating a new parse tree that can be unparsed to obtain an output C++ source.

CONCLUSIONS

A new Measurement Domain Specific Language (MDSL) with 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.

Partial implementation has been applied for demonstration purpose. The final implementation is under development and it will be described in a future publication.

REFERENCES

- [1] S. Amet, L. Bottura, L. Deniau, and L. Walckiers, "The multipoles factory: an element of the LHC control", IEEE Trans. on Appl. Supercond., Vol. 12, pp. 1417-1421, 2002.
- [2] P. Arpaia, L. Bottura, M. Buzio, D. Della Ratta, L. Deniau, V. Inglese, G. Spiezia, S. Tiso, L. Walckiers, "A software framework for flexible

- magnetic measurements at CERN", in Proc. of IEEE IMTC 07, Warsaw, Poland, May 2007.
- [3] P. Arpaia, A. Masi, G. Spiezia, "A Digital Integrator for Fast Accurate Measurement of Magnetic Flux by Rotating Coils", IEEE Transactions on Instrumentation and Measurement, Vol. 56, No. 2, April 2007.
 - [4] P. Arpaia, M. Bernardi, G. Di Lucca, V. Inglese, G. Spiezia, "An Aspect Oriented Programming-based approach to software development for measurement system fault detection", in press on Computer, Standards & Interfaces.
 - [5] P. Arpaia, L. Bottura, V. Inglese, G. Spiezia, "On-field Validation of the New Platform for Magnetic Measurements at CERN", in press on Measurement.
 - [6] J. Bosch and G. Hedin. Editors's Introduction. In Proceedings ALEL'96 Workshop on Compiler Techniques for Application Domain Languages and Extensible Language Models, Technical Report LU-CS-TR:96-173. Lund University, April 1996.
 - [7] A. Van Deursen and P. Klint, Little Languages: Little Maintenance?, Journal of Software Maintenance: Research and Practice, 10(2):75–92, March/April 1998
 - [8] M. Fowler, Domain Specific Language. Online: <http://www.martinfowler.com/bliki/DomainSpecificLanguage.html>
 - [9] Object Management Group. MDA Guide Version 1.0.1. <http://www.omg.org/docs/omg/03-06-01.pdf>, June 2003.
 - [10] Object Management Group. <http://www.omg.org>
 - [11] The Meta-Environment Framework. <http://www.meta-environment.org/>
 - [12] M.G.J. van den Brand et al., The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In Proc. Compiler Construction, LNCS, 2001.
 - [13] Heering, J., P.R.H. Hendriks, P. Klint, and J. Rekers (1989a). "The syntax definition formalism SDF - Reference manual." SIGPLAN Notices 24:43--75
 - [14] Jong, H.A. de and P.A. Olivier (2003). "Generation of abstract programming interfaces from syntax definitions." Journal of Logic and Algebraic Programming .
 - [15] J.A. Bergstra, J. Heering and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering and P. Klint editors, Algebraic Specification, ACM Press in co-operation with Addison-Wesley, 1989, Chapter 1.