
**Department of Information
Technologies & Systems**

**SMML: Software
Measurement Modeling
Language**

*Beatriz Mora Rivas, Félix García Rubio,
Francisco Ruiz González, Mario Piattini*

Alarcos Research Group. University of Castilla-La Mancha, Spain.

Informe Técnico UCLM-TSI-003
Abril 2008



UNIVERSITY OF CASTILLA LA MANCHA

Department of Information Technologies & Systems

Informe Técnico UCLM-TSI-003
Marzo 2008

*Beatriz Mora Rivas, Félix García Rubio,
Francisco Ruiz González, Mario Piattini*

Alarcos Research Group. University of Castilla-La Mancha, Spain.

Palabras clave:

MDA, Software Measurement, FMESP, DSL

This work has been partially financed by the following projects: INGENIO (Universidad Pólitecnica de Valencia, PAC08-0154-9262) and ESFINGE (Ministerio de Educación y Ciencia, TIN2006-15175-C05-05).

Table of contents.

1.	<i>Introduction</i>	1
2.	<i>Domain Specific Modeling</i>	3
2.1.	Introduction	3
2.2.	Domain Specific Languages	3
2.2.1.	How to make a DSL usable	4
2.2.2.	Benefits and Risks of DSLs	5
3.	<i>DSL Development</i>	7
3.1.	Decision	7
3.2.	Analysis	7
3.3.	Design	8
3.4.	Implementation and Deployment	8
4.	<i>SMML</i>	10
4.1.	Definition of an abstract syntax (Domain Measurement Metamodel)	10
4.1.1.	Software Measurement Metamodel	10
4.2.	Definition of a concrete syntax	13
4.3.	Definition of semantics	18
5.	<i>Case of Study</i>	22
5.1.	Definition of a Relational Schemas Maintainability Measurement Model	22
5.2.	Data Quality Model for Web Portals	26
6.	<i>Conclusions and Future Work</i>	28
	<i>ANEXES</i>	29
A.	Software Measurement Ontology	29
B.	Software Measurement Framework	34
	Conceptual Architecture	34
	Technological aspects	36
	<i>References</i>	40

Figures.

<i>Figure 1. The payoff of DSL Development [13].</i>	7
<i>Figure 2. Structure of the packages in the Software Measurement Metamodel.</i>	10
<i>Figure 3. Basic package.</i>	11
<i>Figure 4. Characterization and objectives Package.</i>	12
<i>Figure 5. Software Measures Package.</i>	12
<i>Figure 6. Measurement Approaches Package.</i>	13
<i>Figure 7. Characterization and Objectives Instance with UML.</i>	22
<i>Figure 8. Characterization and Objectives Instance with SMML.</i>	23
<i>Figure 9. Software Measures Package Instance with UML.</i>	24
<i>Figure 10. Software Measures Package Instance with SMML.</i>	24
<i>Figure 11. Measurement Approaches Package Instance.</i>	26
<i>Figure 12. Measurement Model of PDQM represented with SMML.</i>	27
<i>Figure 13. Software Measurement Ontology.</i>	30
<i>Figure 14. Conceptual framework with which to manage software measurement.</i>	35
<i>Figure 15. Elements of the FMESP adaptation in a MDA context.</i>	36
<i>Figure 16. QVT Relations transformation model.</i>	37
<i>Figure 17. Software Measurement process.</i>	38

Tables.

<i>Table 1. Terms in the ‘software characterization and objectives’ package.</i>	14
<i>Table 2. Relationships in the ‘software measurement characterization and objectives’ package.</i>	15
<i>Table 3. Terms in the ‘Software Measures’ package.</i>	16
<i>Table 4. Relationships in the ‘Software Measures’ package.</i>	16
<i>Table 5. Terms in the ‘Measurement Approaches’ package.</i>	17
<i>Table 6. Relationships in the ‘Measurement approaches’ package.</i>	17
<i>Table 7. SMML OCL Coinstraints.</i>	18
<i>Table 8. SMML OCL Coinstraints.</i>	21
<i>Table 9. Base Measures and Measurement Methods of the Relational Schemas Measurement Model.</i>	25
<i>Table 10. Indicators, Analysis Models and Decision Criteria of the Relational Schemas Measurement Model.</i>	25
<i>Table 11. Definition of the terms in the “software measurement characterization and objectives” sub-ontology.</i>	30
<i>Table 12. Definition of the terms in the “software measures” sub-ontology.</i>	31
<i>Table 13. Definition of the terms in the “measurement approaches” sub-ontology.</i>	31
<i>Table 14. Definition of the terms in the “measurement action” sub-ontology.</i>	32
<i>Table 15. Relationships in the ‘software measurement characterization and objectives’ sub-ontology.</i>	32
<i>Table 16. Relationships in the ‘software measures’ sub-ontology.</i>	33
<i>Table 17. Relationships in the ‘measurement approaches’ sub-ontology.</i>	33
<i>Table 18. Relationships in the ‘measurement’ sub-ontology.</i>	34

1. Introduction

Software Measurement has become a fundamental aspect of Software Engineering [15]. Measurement is proving to be highly efficient in, amongst other things, the construction of high quality prediction systems for large-scale data base projects [31], in the understanding and improvement of software development and maintenance projects [9], in the evaluation and guarantee of system quality (by highlighting problematic areas) [12], and in the determination of better work practices with the end of assisting users and investigators in their work [12]. Software measures are, moreover, important tools which assist in the evaluation and institutionalization of Software Process Improvement in those organizations which develop them. Software Measurement is, in fact, a key element in initiatives such as SW-CMM (*Capability Maturity Model for Software*), ISO/IEC 15504 (SPICE, *Software Process Improvement and Capability dEtermination*) and CMMI (*Capability Maturity Model Integration*). The ISO/IEC 90003:2004 standard [27] also highlights the importance of measurement in managing and guaranteeing quality. Various methods and standards with which to carry out measurements in a precise and systematic manner exist, of which the most representative are:

- **Goal Question Metric (GQM):** The basic principle of GQM is that the carrying out of the measurement must always be orientated towards an objective. GQM defines an objective, refines that objective into questions and defines measures which attempt to answer those questions.
- **Practical Software Measurement (PSM):** The PSM methodology [32] is based upon the experience obtained from organizations through which the best manner in which to implement a software measurement programme with guarantees of success is discovered.
- **IEEE 1992 (Methodology for Software Quality Measures):** according to the IEEE 1992 standard, software quality can be considered as the extent to which the software possesses a clearly defined and desirable combination of quality attributes. This standard deals with the definition of software quality for a system by using a list of software quality attributes which are required by the system itself.
- **ISO/IEC 15939:** this international standard [26] identifies the activities and tasks which are necessary to successfully identify, define, select, apply and improve software measurement within a general project or within a business's measurement structure.

The availability of a language which allows us to represent those elements which must be taken into account in the measurement processes might, therefore, be important in decision making and in process improvement.

It is thus of interest to consider the use of Domain Specific Languages (DSLs). DSLs are a contribution of Domain Specific Modelling (DSM). Domain-Specific Modelling raises the level of abstraction beyond programming by specifying the solution with the direct use of domain concepts. The final products are generated from these high-level specifications. This automation is possible because both the language and generators need to fit the requirements of only one company and domain. Industrial experiences of DSM consistently show it to be 5-10 times more productive than current software development practices, including current UML-based implementations of MDA. DSM does to code what compilers did to assembly language. Besides this vision, more

investigation is needed in order to advance the acceptance and viability of DSM [1]. Selection of a domain is a first step towards development of domain-specific languages. Selecting a domain implies tradeoffs between more general applicability of the DSL and more specificity [38]. In other words, a trade off between the focus and size of the language is needed. A language which represents a larger domain can be weakly specialized to any particular aspect of the domain. On the contrary, a language that represents a small domain may have a limited number of target users [6].

These aspects constitute the main interest of this paper, whose objective is to propose the Software Modelling Measurement Language (SMML) which permits software measurement models to be modeled in a simple and intuitive manner. This has been done by using the Software Measurement Metamodel (SMM) as the Domain Definition Metamodel (DDMM). This language forms a part of the Software Measurement Framework (SMF) presented in [34] (for a greater detail see ANEXE □B). SMF allows us to carry out generic measurement through transformations by using two initial models as a starting point: that of software measurement and that of domain. The task of the SMML is to facilitate the user the modelling of the software measurement models, which is the starting point in the generic software measurement process.

The remainder of this work is organized as follows. Section 2 provides the state of art of Domain Specifying Modelin, and Section 3 briefly describes the DSL Development. In Section 4 the SMML Languages is explained, including the definition of the abstract syntax, concrete syntax and semantics. Section 5 illustrates the use of the Language with a case study. Finally, conclusions and future works are outlined in Section 6.

2. Domain Specific Modeling

The goal of this section is to provide current state-of-art in Domain-Specific Modeling.

2.1. Introduction

An independent organization DSM Forum [1] exists to spread the knowledge and know-how of Domain-Specific Modeling (DSM). DSM Forum embodies its vision in the following statements [1]:

“Domain-Specific Modeling raises the level of abstraction beyond programming by specifying the solution directly using domain concepts. The final products are generated from these high-level specifications. This automation is possible because both the language and generators need fit the requirements of only one company and domain. Industrial experiences of DSM consistently show it to be 5-10 times more productive than current software development practices, including current UML-based implementations of MDA. DSM does to code what compilers did to assembly language.” Besides this vision, more investigation is needed in order to advance the acceptance and viability of DSM.

Selection of a domain is first step towards development of domain-specific languages. Selecting a domain implies tradeoffs between more general applicability of the DSL and more specificity [38]. In another words, a tradeoff between the focus and size of the language is needed. A language that represents a larger domain can weakly specialize to any particular aspect of the domain. To the contrary, a language that represents a small domain may have a limited number of target users [6]. Völter [38] distinguishes two kinds of software domains:

- Technical (or Horizontal) domains address key technical issues related to software development such as Distribution, Persistence and Transactions, Graphical User Interface (GUI) Design or Concurrency.
- Functional (or Vertical) domains address business issues such as Banking, Human Resource Management and Insurance.

A typical software system consists of several domains mentioned above.

Kelly [28] argues that attempts to make a completely generic modeling language and generators have failed due to the reason that raising the level of abstraction means sacrificing fine control and complete generality. In this sense, if the modeling language is too broad, it is likely that communities would use disjoint subsets of the language with no real communication between them. When a modeling language becomes larger and more complicated, it becomes more difficult to understand models defined in that language [6]. Thus, the only way to generate full code directly from models is to make both the modeling language and generators domain-specific [28].

2.2. Domain Specific Languages

General-purpose language (GPL) refers to a language that encodes generic abstractions. To the contrary, domain specific language refers to a language that encodes abstractions used in a narrow domain [11]. A domain specific modeling language raises the level of abstraction and bridges the implementation closer to the vocabulary understood by domain experts, engineers and end-users. A DSL enables the specification of software from a specific viewpoint. A GPL such as UML is not viewpoint-based; instead it supports only generic, undifferentiated specification [25]. Even though UML 2.0 allows

users model system from four viewpoints (static structural, interaction, activity and state), it currently does not provide mechanism for creating models using user-defined viewpoints which help developers to model complex system [18]. Today business applications cover multiple functional domains, thus multiple DSLs are needed to describe them [25].

Examples of popular DSLs include:

- **SQL:** A structured query language provides an interface to relational database management systems.
- **HTML,** a markup language for the creation of web pages.

DSLs and Domain-Specific Modeling Languages (DSMLs) are considered to be the same and the terms are usually used interchangeably. DSL is represented by a metamodel and its models conform to that metamodel. DSLs could be the right technique to develop software in shorter time with better quality. DSLs are considered as a good solution to the problem of reuse both on technical and on architectural and design level [14].

2.2.1. How to make a DSL usable

Feilkas [14] cites the tasks that have to be carried out to make a DSL usable:

1. **Definition of an abstract syntax:** many DSL-tools allow the definition of the abstract syntax as a metamodel. This metamodel is defined by a data modeling technique similar to class diagrams or Entity Relationship (ER) diagrams [14].
2. **Definition of a concrete syntax:** for every language element there has to be a graphical symbol that represents the abstract model element. In the case of textual languages, both abstract and concrete syntax can be described by a grammar which specifies terminals, non-terminals and production rules [14].
3. **Definition of semantics:** the semantics of the DSL is defined by implementing the generator backend and giving translational semantics into some target language which already has some behaviour definition for its elements [14].

There are three kinds of approaches to realize the generator backend [14]:

- **Templates are the most preferred approach to code generation in DSL-tools (language workbenches).** In this approach, code-files in the target language are considered as basis. Expressions of a macro language that specify the generator instructions are inserted. Ordinary programming languages are used to specify the behaviour of the generator (C# in the MS/DSL Tools) [14].
- **Patterns approach allows specifying a search pattern on the model graph,** in this way for every match a specific output is generated [14].
- **Graph-traversing approach specifies a predetermined iteration path over the syntax graph.** For every node type, generation instructions are defined which are executed every time such a node is passed. This approach is used in classical compiler construction and textual languages [14].

Feilkas [14] cites that the generation techniques that are used by DSL-tools do not respect target language syntax. In this sense, a formal mapping between the source and target language elements is crucial to specify the semantics of a newly developed DSL.

By specifying a translation to a target language that has operational semantics a newly developed DSL implicitly gets a formal definition of its own semantics [14].

2.2.2. Benefits and Risks of DSLs

Good DSLs are often small and simple refer to the term “little languages”. To make it useful, a DSL should be designed as a limited language tightly focused on a single problem [17]. DSL development involves both risks and opportunities. The main benefits include:

- Since DSLs allow solutions to be expressed at the domain level, domain experts can understand, validate, modify and even develop domain models expressed in DSLs [19].
- DSLs are concise, self-documenting to a large extent, and can be reused for different purposes [38].
- DSLs are small languages and very focused on a specific tasks and have very well defined semantics, thus they can be easily tooled [25].
- DSLs enable developers to separate previously connected development activities for a software system. In this way, they can concentrate on a single task at a time which leads to better results and more efficient development process [11].
- DSLs enhance productivity [13], maintainability [25], reliability [13], portability [19] and reusability of software artifacts [5].
- DSLs enable expression, validation and optimization of concepts at the level of abstraction of the problem domain [5, 19].
- A DSL can facilitate the construction of complex test cases, thus it improves testability of software [19].

On the other hand, potential risks include:

- The high expense of designing, implementing and maintaining DSLs and tools[25].
- The high expense of training DSL users and migrating developer skills [19, 25].
- The difficulty of finding the proper scope for a DSL [19]. DSLs are limited both in scope (they refer to a particular domain) and capability (they lack features that are basic for general-purpose languages) [17].
- The difficulty of balancing between domain-specificity and general-purpose programming language constructs [19].
- The potential loss of efficiency comparing to the manually written source code [13, 19].

DSLs can be implemented as standalone languages or extensions to existing languages[11]. Since it is costly to develop new DSLs from the scratch, instead existing languages like XML can be extended. Instead building new compilers, editors, and debuggers from the scratch, plug-ins for existing IDEs such as .NET or J2EE can be built [25].

In order to mitigate the risks involved in building DSLs, organizations can leverage the skills of their most talented developers by making them product line developers, who

build the tools and languages used by product developers. In this way, organizations will have better technology for harvesting, refining, documenting, testing, packaging, and supporting reusable assets [25].

To elaborate discussion, Fowler [16] divides them into two broader styles: internal (e.g. Lisp) and external DSLs (e.g. XML configuration files). External DSLs are written in a different language than the main (host) language of the application and transformed into it using some form of compiler or interpreter. The advantage of an external DSL is its ability to express domain in the easiest form to read and modify. In addition, an external DSL can be evaluated at runtime; in this way commonly changed parameters can be altered without recompiling the program. Then, a translator should be built to parse the configuration file and produce something executable. However, when the language gets more complex, the cost of building translator increases. The major disadvantage of external DSLs is that they lack symbolic integration meaning that the DSL is not linked to the base language [16]. On the other hand, internal DSLs morph the host language into a DSL itself. In contrast to external DSLs, internal DSLs eliminate the symbolic barrier with the base language and make the full power (tooling) of base language available. The disadvantage of internal DSLs is that they are limited by the syntax and structure of the base language [16].

3. DSL Development

DSL development requires both domain knowledge and language development expertise. The development process should be facilitated by using DSM tools. Mernik et al [33] cite that DSL development consists of five phases: decision, analysis, design, implementation and deployment. DSL development is not a sequential process meaning that former phases are often influenced by latter phases.

3.1. Decision

It is costly to design, build and maintain DSLs regardless of which DSM tool (MS/DSL Tools, Eclipse or other tools) is used in development. Thus, in design phase a cost-benefit analysis should be carried out to determine whether it is more beneficial to use DSLs as opposed to general-purpose languages or not.

As it is illustrated in Figure 1, DSL development requires higher startup costs due to complexity of designing and implementing DSLs. However, it reduces development life cycle costs afterwards due to productivity increase [13]. In this sense, investment in DSL development has to pay itself by more economical software development and maintenance in later applications [33]. Once the decision is made towards DSL development, next step would be to define the slope and size of the language and to select the appropriate DSL tools.

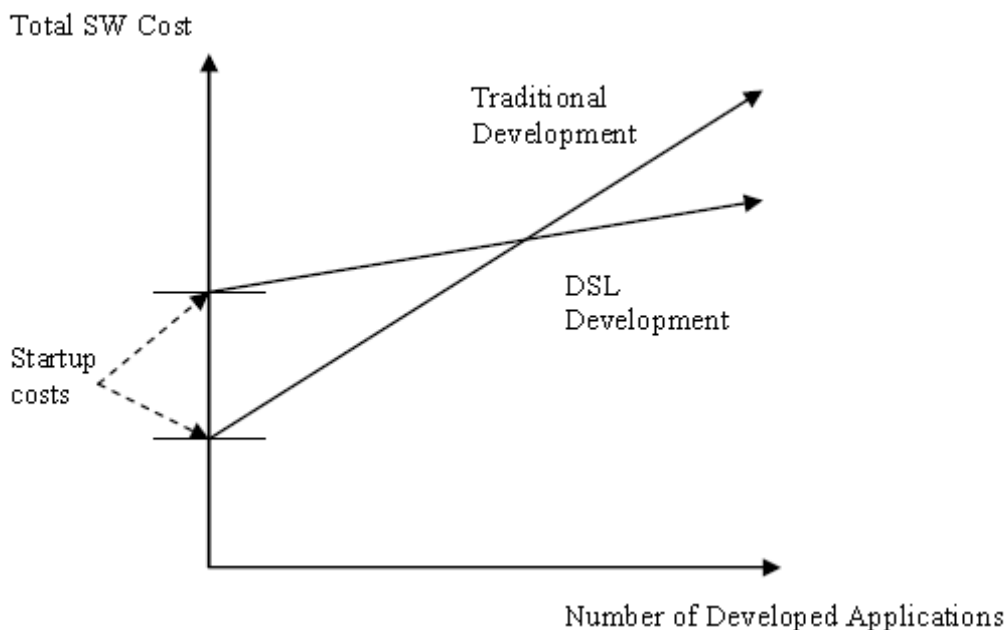


Figure 1. The payoff of DSL Development [13].

3.2. Analysis

In analysis phase of DSL development, problem domain is identified and domain knowledge is gathered. Various implicit or explicit sources for domain knowledge could be existing technical documents; knowledge provided by domain experts, existing models defined in general-purpose languages, and customer surveys. In addition, knowledge engineering practices such as knowledge capturing and knowledge representation can facilitate elicitation of domain knowledge. The output of domain analysis should include a domain model consisting of a definition of the scope of the domain, domain terminology, description of domain concepts and commonalities,

variabilities and dependencies between them [33]. Völter et al [68] cite that when trying to find an appropriate DSL for a certain domain, one should always take into account the required amount of variability. To express variants in the models, tool support (e.g. feature diagrams) should be available [39].

3.3. Design

The easiest way to design a DSL is to base it on an existing language and to extend existing language with new features that address domain concepts. Once the relationship to existing languages has been resolved, a DSL designer should be used to specify design before implementation [33].

Kolovos et al [29] cite that quality attributes of a DSL and its supporting environment (e.g. compilers, IDE) have an impact on the quality attributes of the overall development process and the resulting product. In design phase of DSL development, the quality requirements of the DSL should be specified and necessary trade-offs should be made. The core requirements for a DSL are as follows [29]:

- **Conformity:** The language constructs must correspond to important domain concepts.
- **Orthogonality:** Each language construct is used to represent exactly one distinct concept (e.g. Business Entity, Business Rule) in the domain.
- **Supportability:** DSL tools shall provide support for model management and transformation.
- **Integrability:** It is essential to integrate DSL tools with other facilities used in development process. DSL tools should provide extensibility mechanism to support additional constructs and concepts.
- **Longevity:** In order to ensure tool support and to make it possible to quantify the payoff obtained from using the DSL, the DSL should be used for a non-trivial period of time. It is assumed that the selected software domain would persist for a sufficiently lengthy period of time to justify the cost of building DSL and its tools.
- **Simplicity:** A DSL should be simple as possible in order to express the domain concepts and to support its users.
- **Quality:** The DSL shall provide general mechanisms for building quality systems that include language constructs for improving reliability, security, safety, etc.
- **Scalability:** The DSL constructs should be able to manage large-scale descriptions. However, it is not a necessary requirement, besides small languages are preferable.
- **Usability:** DSL constructs shall be expressive and easy to understand.

3.4. Implementation and Deployment

Once an executable DSL is designed, the most appropriate implementation approach (e.g. interpreters, compilers/application generators, embedded) should be chosen. Depending on the selected approach some implementation trade-offs have to be considered. Compilers/application generators provide syntax which is close to the notation used by domain experts, so that they facilitate domain-specific analysis, verification and optimization. However the required effort to design

compilers/application generators is large comparing to embedded approach [33]. It is common that generators are used for structural aspects of a system and interpreters for behavioral aspects. The main pitfall of interpreter approach is that interpreters are inherently slower than generated code. On the other hand, the main advantage of interpreters is that they allow late binding at runtime so that a model can be changed at runtime without the need for redeployment or rebuild. In case of generators, generated code is bounded at compile time, so after each modification in the model, regeneration becomes a necessity. Decision regarding whether to build generators or interpreters for a given domain draws the boundaries of the domain, thus decision can be postponed until a solid understanding of the domain is achieved [38].

Regarding code generation, an important issue is to combine generated code and hand-written code in a controlled environment. In order to avoid inconsistency problems that appear during modification of generated code, developers should be guided by IDE regarding what they are allowed to access in generated code. Another solution could be to keep generated and manually written code in separate files. An architectural description that clearly defines which artifacts are generated and how generated code is combined with manually written code is needed [38]. In the case of MS/DSL Tools, use of partial classes that can be regenerated can maintain separation of generated code and manually written code efficiently. In the case of Eclipse, descriptive tags like “@generated” are used to serve for this purpose instead of using separate files. If handwritten code must be inserted to the generated code, introduction of protected areas is required. These areas can be read by the code generator, in this way manually written code will not be overwritten during regeneration. However, the disadvantages of this approach are that code generator and versioning becomes more complex as well as the separation of generated and non-generated code dissolves.

Völter et al [38] argue that quality of generated code depends on the transformations such as templates that generate C# in MS/DSL Tools. Thus, if transformations are created with necessary care, generated code’s quality will not be worse than hand written code, besides it is more systematic and consistent than hand written code.

Regarding documentation, the DSL and its semantics must be documented to enable its efficient use in development projects. Although models document the domain concepts in a form that can be understood by domain experts, user manual and other documentation can also be generated from models. The advantage of generated documentation is that it is only required once for each DSL, not for each application[38].

Regarding deployment, Visual Studio 2005 allows to create DSL Tools Setup Project template, when it is built, a setup.exe file for installing the DSL designer is generated. This setup project automates the process of producing a Microsoft Windows Installer (MSI) package to deploy a graphical designer and its associated component. Due to seamless integration into the Eclipse IDE, graphical editors developed using Eclipse GEF/GMF can directly use Eclipse’s code generation and other plug-ins and can be readily packaged (as .jar files) and deployed.

To conclude this section, one can say that both MS/DSL Tools and Eclipse provide support for design (metamodeling), implementation and deployment phases of DSL development, however they do not provide any support for decision and domain analysis phases.

4. SMML

SMML is a language which permits software measurement models to be built in a simple and intuitive manner. The SMML development requires both domain knowledge and language development expertise [33].

Feilkas [14] cites the tasks that must be carried out to make a DSL usable: Definition of an abstract syntax, Definition of a concrete syntax and Definition of semantics. The following subsection describes how these stages have been used to develop the Software Measurement Modeling Language (SMML) [30].

4.1. Definition of an abstract syntax (Domain Measurement Metamodel)

The SMML syntax is obtained from the Software Measurement Ontology (SMO) [20]. This ontology contains all the elements (concepts and relationships) of the software measurement domain (for a greater detail see ANEXE A).

4.1.1. Software Measurement Metamodel

One of the defining entities of a DSL is a Domain Definition MetaModel (DDMM) [30]. This introduces the basic entities of the domain and their mutual relations. This base ontology plays a central role in the definition of the DSL. Such a DDMM plays the role of the abstract syntax for a DSL.

In order to develop SMML, a Domain Definition Metamodel is therefore necessary. The Software Measurement Metamodel (SMM) exists, which is derived from the Software Measurement Ontology (SMO). This metamodel is the Domain Definition Metamodel used to define the abstract syntax of SMML.

The Software Measurement Metamodel includes the packages which are alignments with the sub-ontologies of SMO (Basic, Characterization and Objectives, Measures Software, Measurement Approaches and Measurement Action). However, for the development of the Language, all the packages are of interest, with the exception of *Measurement Action*. This has been excluded as it contains the elements which are relative to measurement but not to the domain problem. Figure 2 shows the structure of the packages upon which the SMML language is based.

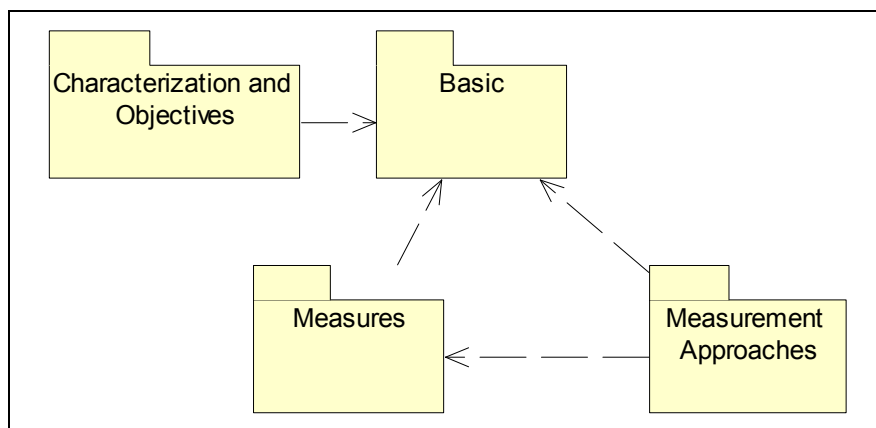


Figure 2. Structure of the packages in the Software Measurement Metamodel.

As will be observed in Figure 2, the metamodel is made up of a basic package which represents the general characteristics of the basic constructors of the measurement models, and three other packages (Characterization and Objectives, Measurement Approaches and Measures), in accordance with the three sub-ontologies of the SMO. The conceptualization established in the Software Measurement Ontology has been taken into account in the construction of this metamodel, but the specific constructors have been added from the point of view of implementation.

All of the elements identified in the ontology (Measure, Information need, Measurable concept, etc.) are potential elements of the Software Measurement Metamodel on which the SMML language is based. On the other hand, the relationships which exist in the ontology do not correspond with the relationships which are necessary for the language. All of the Measurement Metamodel packages maintain the original definition of [24] with the exception of the basic package, which has had to be adapted to represent the measurement relationships in SMML.

In section 4.2 is gave a detailed description of the relationships of the Software Measurement Metamodel which correspond with the relationships in the SMO ontology. All the types of relationships that are identified in the ontology, and which have been defined for the metamodel, have been studied. The elements involved (a source and a target) are indicated for each relationship. In total, 4 types of *Measurement Associations* have been identified: association, nonnavigable association, aggregation and dependency. These relationships have been defined in the Basic package.

Basic Package

this basic package has been defined in order to identify and to establish the general features of the constructor necessary to define measurement model. With regard to the Software Measurement Metamodel defined in [24], 4 types of *Measurement Association* have been added: association, nonnavigable association, aggregation and dependency. Figure 3 shows the UML diagram which displays the structure of this package.

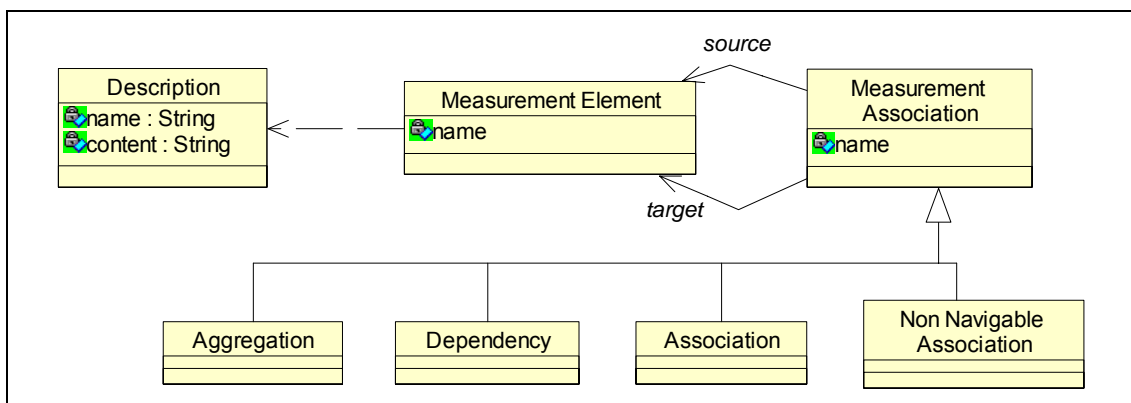


Figure 3. Basic package.

As can be observed in Figure 3, the general element from which measurement models are constructed is the “**Measurement Element**” constructor, and the general element from which the relationships of the models are constructed is the “**Measurement Association**” constructor. A measurement element has a name and can be described through elements of the “**Description**” type, which give additional information about the measurement elements, and this facilitates a better understanding of the measurement models developed. The measurement element is used as a starting point from which to specialize the measure’s fundamental constructors, obtained from the

Software Measurement Ontology concepts. A Measurement Element relates two measurement elements, a source element and a target element. The Measurement Association is used to specialize the relationship constructors defined for the metamodel: Association, Nonnavigable association, Aggregation and Dependency.

Characterization and objectives Package

This package includes the constructors required to establish the scope and objectives of the software measurement process. Figure 4 shows the UML diagram which displays the structure of this package.

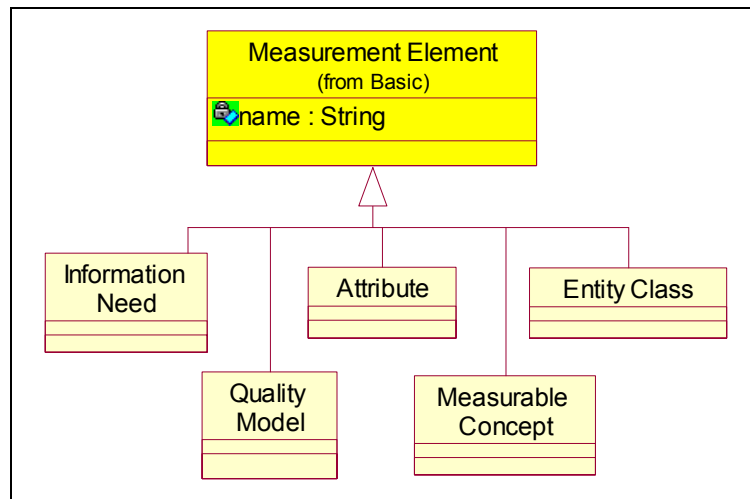


Figure 4. Characterization and objectives Package.

Software Measures Package

this package includes the constructors needed to establish and to clarify the key elements in the definition of a software measure. Figure 5 shows the UML diagram which displays the structure of this package.

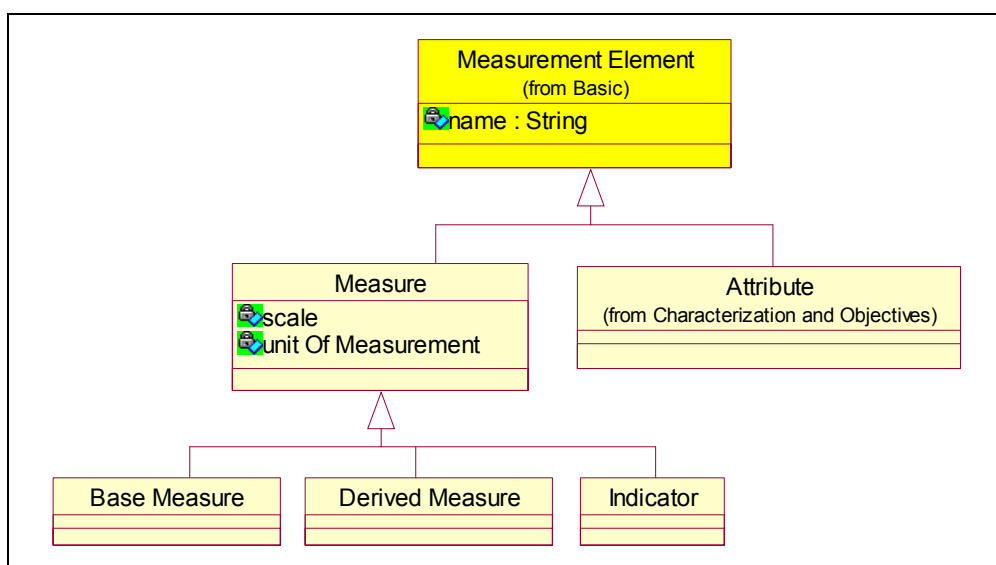


Figure 5. Software Measures Package.

Measurement Approaches Package

this package includes the constructors needed to generalize the different ‘approaches’ used by the three kinds of measures to obtain their respective measurement results. Figure 6 shows the UML diagram which displays the structure of this package.

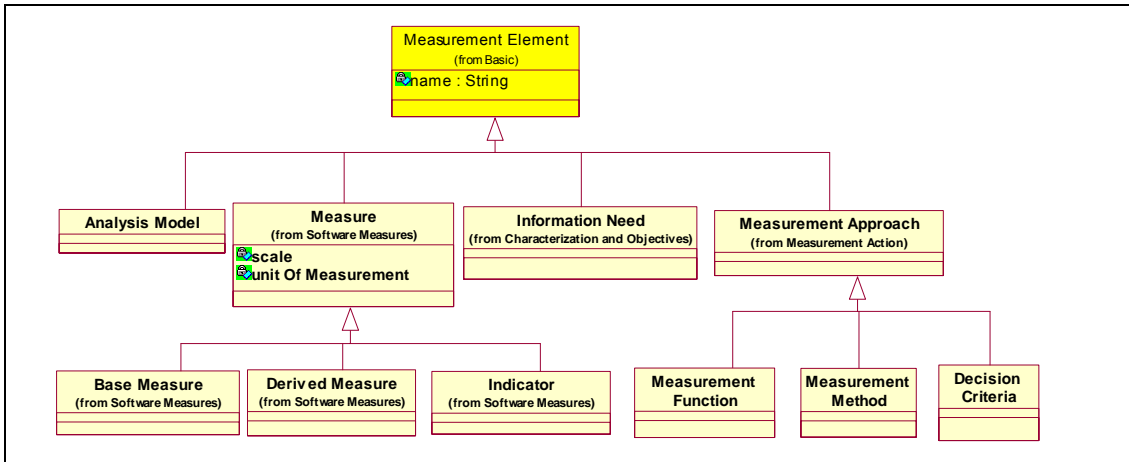


Figure 6. Measurement Approaches Package.

4.2. Definition of a concrete syntax

In order to make the language usable, a concrete syntax must be defined. All of the elements are defined in the basic package (see Figure 3).

Each of these elements of the language must be associated with a graphic icon which represents the element of the abstract model. Each language element and relationship has been associated with a representative icon in the SMML. Icons which are familiar to software engineers have been used in order to facilitate its use. For example, the *Description* element is very similar to the *UML note* element, the difference being that it includes an image of a ruler (as a symbol of measurement) in its top right-hand corner. In a similar manner, the *Entity* element is taken from the *Entity Class* in an E/R Diagram.

The following tables provide the terms and relationships of the language:

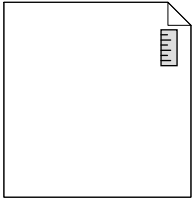

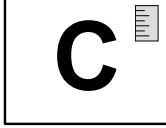
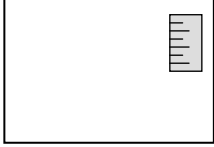

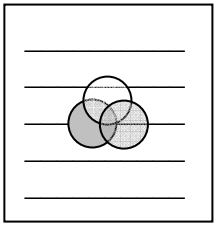
Term	Icon	Definition	Source
Description		Description of the <i>measurement element</i> .	UML not
Information Need		Insight necessary to manage objectives, goals, risks, and problems.	
Measurable concept		Abstract relationship between <i>attributes of entities</i> and <i>information needs</i> .	
Entity class		The collection of all <i>entities</i> that satisfy a given predicate	E/R Entity
Attribute		A measurable physical or abstract property of an <i>entity</i> , that is shared by all the <i>entities</i> of an <i>entity class</i> .	E/R Attribute
Quality model		The set of <i>measurable concepts</i> and the relationships between them which provide the basis for specifying quality requirements and evaluating the quality of the <i>entities</i> of a given <i>entity class</i> .	TQM

Table 1. Terms in the ‘software characterization and objectives’ package.

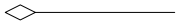
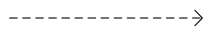


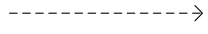
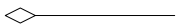
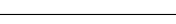
Relationships	Icon	Description	Source
Includes		An <i>entity class</i> may include several other <i>entity classes</i> . An <i>entity class</i> may be included in several other <i>entity classes</i> ,	UML Agregation
Defined for		A <i>quality model</i> is defined for a certain <i>entity class</i> . An <i>entity class</i> may have several <i>quality models</i> associated	UML Dependency
Evaluates		A <i>quality model</i> evaluates one or more <i>measurable concepts</i> . A <i>measurable concept</i> is evaluated by one or more quality models .	UML Dependency
Relates		A <i>Measurable concept</i> relates one or more <i>attributes</i> . An <i>Attribute</i> is related with one or more <i>measurable concepts</i> .	UML Association
Is associated with		A <i>measurable concept</i> is associated with one or more <i>information needs</i> . An <i>information need</i> is related to one <i>measurable concept</i> .	UML Dependency
Includes		A <i>measurable concept</i> may include several <i>measurable concepts</i> . A <i>measurable concept</i> may be included in several other <i>measurable concepts</i> .	UML Agregation
Has		An <i>entity class</i> has one or more <i>attributes</i> . An <i>attribute</i> can only belong to one <i>entity class</i> .	UML Association not navigable

Table 2. Relationships in the ‘software measurement characterization and objectives’ package.

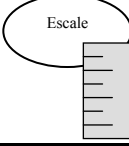
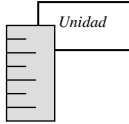

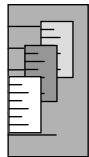
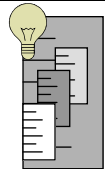
Term	Icon	Definition
Measure	Abstract Element	The defined <i>measurement approach</i> and the <i>measurement scale</i> . (A <i>measurement approach</i> is either a <i>measurement method</i> , a <i>measurement function</i> or an <i>analysis model</i>).
Scale		A set of values with defined property
Unit of measurement		Particular quantity, defined and adopted by convention, with which other quantities of the same kind are compared in order to express their magnitude relative to that quantity
Base Measure		A measure of an <i>attribute</i> that does not depend upon any other measure, and whose <i>measurement approach</i> is a <i>measurement method</i> .
Derived measure		A measure that is derived from other <i>base or derived measures</i> , using a <i>measurement function</i> as <i>measurement Approach</i> .
Indicator		A measure that is derived from other measures using an <i>analysis model</i> as <i>measurement approach</i> .

Table 3. Terms in the ‘Software Measures’ package.


Relationship	Icon	Description	Source
Defined for		A <i>measure is defined for</i> one or more <i>attributes</i> . An <i>attribute</i> may have several associated <i>measures</i> .	UML Dependency

Table 4. Relationships in the ‘Software Measures’ package.

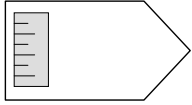
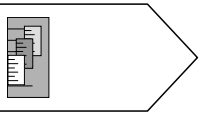
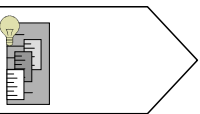
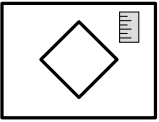
Term	Icon	Definition
Measurement Method		A <i>measurement method</i> is the <i>measurement approach</i> that defines a <i>base measure</i> .
Measurement Function		A <i>measurement function</i> is the <i>measurement approach</i> that defines a <i>derived measure</i> .
Analysis Model		An <i>analysis model</i> is the <i>measurement approach</i> that defines an <i>indicator</i> .
Decision Criteria		Thresholds, targets, or patterns used to determine the need for action or further investigation, or to describe the level of confidence in a given result.

Table 5. Terms in the ‘Measurement Approaches’ package.




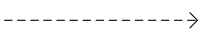
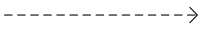
Relationship	Icon	Description	Source
Calculated with		Every <i>derived measure</i> is calculated with one <i>measurement function</i> . Every <i>measurement function</i> may define one or more <i>derived measures</i> .	UML Association
Calculated with		Every <i>indicator</i> is calculated with one <i>analysis model</i> . Every <i>analysis model</i> may define one or more <i>indicators</i> .	UML Association
Uses		Every <i>base measure</i> uses one <i>measurement method</i> . Every <i>measurement method</i> defines one or more <i>base measures</i> .	UML Association
Satisfies		An <i>indicator</i> may satisfy several <i>information needs</i> . Every <i>information need</i> is satisfied by one or more <i>indicator</i> .	UML Dependency
Uses		An <i>analysis model</i> uses one or more <i>decision criteria</i> . Every <i>decision criteria</i> is used in one or more <i>analysis models</i> .	UML Dependency

Table 6. Relationships in the ‘Measurement approaches’ package.

4.3. Definition of semantics

The most important aspect of language specification is possibly the definition of its semantics. An informal description of the language must be given in a natural language which describes its domain. The semantics of the language have been defined by using OCL constraints on the metamodel. These constraints define the cardinality and the elements involved in the associations. The OCL Constraints relating to Measures are shown in the following tables:

The OCL constraints in the Table 7 check the element in the associations will be correct.

Element	OCL Constraint
NonNavigable Association	<code>self.source.ocIsTypeOf(EntityClass) and self.target.ocIsTypeOf(Attribute)</code>
Association	<code>self.source.ocIsTypeOf(MeasurableConcept) and self.target.ocIsTypeOf(Attribute) or self.source.ocIsTypeOf(DerivedMeasure) and self.target.ocIsTypeOf(MeasurementFunction) or self.source.ocIsTypeOf(BaseMeasure) and self.target.ocIsTypeOf(MeasurementMethod) or self.source.ocIsTypeOf(Indicator) and self.target.ocIsTypeOf(AnalysisModel)</code>
Agregation	<code>self.source.ocIsTypeOf(EntityClass) and self.target.ocIsTypeOf(EntityClass)</code>
Dependency	<code>(self.source.ocIsTypeOf(QualityModel) and self.target.ocIsTypeOf(EntityClass)) or (self.source.ocIsTypeOf(QualityModel) and self.target.ocIsTypeOf(MeasurableConcept)) or (self.source.ocIsTypeOf(MeasurableConcept) and self.target.ocIsTypeOf(InformationNeed)) or (self.source.ocIsTypeOf(AnalysisModel) and self.target.ocIsTypeOf(DecisionCriteria)) or (self.source.ocIsTypeOf(Indicator) and self.target.ocIsTypeOf(InformationNeed)) or (self.source.ocIsTypeOf(Measure) and self.target.ocIsTypeOf(Attribute))</code>

Table 7. SMML OCL Coinstraints.

The OCL constraints in the Table 8 check the associations cardinality.

Condition	OCL Constraint
Association not navigable	
An <i>entity class</i> has one or more <i>attributes</i> . 1 → 1..*	<code>NonNavigableAssociation.allInstances()-> exists(a:Association a.source.ocIsTypeOf(EntityClass) and a.target.ocIsTypeOf(Attribute))</code>
An <i>attribute</i> can only belong to one entity class . 1 ← 1	<code>NonNavigableAssociation.allInstances()-> forAll(a1, a2: NonNavigableAssociation (a1 <> a2 and a1.source.ocIsTypeOf(EntityClass) and a1.target.ocIsTypeOf(Attribute) and A2.source.ocIsTypeOf(EntityClass) and a2.target.ocIsTypeOf(Attribute)) implies not (a1.source = a2.source and a1.target <> a2.target))</code>

Association	
A <i>Measurable concept</i> relates one or more <i>attributes</i> . 1 → 1..*	Association.allInstances()-> exists(a:Association a.source.ocIsTypeOf(MeasurableConcept) and a.target.ocIsTypeOf(Attribute))
An <i>Attribute</i> is related with one or more <i>measurable concepts</i> . 1..* ← 1	Implicit in previous OCL Coinstraint.
Every <i>derived measure</i> is calculated with one <i>measurement function</i> . 1..* → 1	Association.allInstances()-> exists(a:Association a.source.ocIsTypeOf(DerivedMeasure) and a.target.ocIsTypeOf(MeasurementFunction)) and Association.allInstances()-> forAll(a1, a2:Association (a1 <> a2 and a1.source.ocIsTypeOf(DerivedMeasure) and a1.target.ocIsTypeOf(MeasurementFunction) and a2.source.ocIsTypeOf(DerivedMeasure) and a2.target.ocIsTypeOf(MeasurementFunction)) implies ((a1.source <> a2.source and a1.target <> a2.target)))
Every <i>measurement function</i> may define one or more <i>derived measures</i> . 1..* ← 1..*	Implicit in previous OCL Coinstraint.
Every <i>indicator</i> is calculated with one <i>analysis model</i> . 1..* → 1	Association.allInstances()-> exists(a:Association a.source.ocIsTypeOf(Indicator) and a.target.ocIsTypeOf(AnalysisModel)) and Association.allInstances()-> forAll(a1, a2:Association (a1 <> a2 and a1.source.ocIsTypeOf(Indicator) and a1.target.ocIsTypeOf(AnalysisModel) and a2.source.ocIsTypeOf(Indicator) and a2.target.ocIsTypeOf(AnalysisModel)) implies ((a1.source <> a2.source and a1.target <> a2.target)))
Every <i>analysis model</i> may define one or more <i>indicators</i> .	Implicit in previous OCL Coinstraint.
Every <i>base measure</i> uses one <i>measurement method</i> .	Association.allInstances()-> exists(a:Association a.source.ocIsTypeOf(BaseMeasure) and a.target.ocIsTypeOf(MeasurementMethod)) and Association.allInstances()-> forAll(a1, a2:Association (a1 <> a2 and a1.source.ocIsTypeOf(BaseMeasure) and a1.target.ocIsTypeOf(MeasurementMethod) and a2.source.ocIsTypeOf(BaseMeasure) and a2.target.ocIsTypeOf(MeasurementMethod)) implies ((a1.source <> a2.source and a1.target <> a2.target)))
Every <i>measurement method</i> defines one or more <i>base measures</i> .	Implicit in previous OCL Coinstraint.

Agregation	
An <i>entity class</i> may include several other <i>entity classes</i> .	Agregation.allInstances()-> exists(a: Agregation a.source.ocIsTypeOf(EntityClass) and a.target.ocIsTypeOf(EntityClass))
An <i>entity class</i> may be included in several other <i>entity classes</i> ,	Implicit in previous OCL Coinstraint
A <i>measurable concept</i> may include several <i>measurable concepts</i> .	Agregation.allInstances()-> exists(a: Agregation a.source.ocIsTypeOf(MeasurableConcept) and a.target.ocIsTypeOf(MeasurableConcept))
A <i>measurable concept</i> may be included in several other <i>measurable concepts</i> .	Implicit in previous OCL Coinstraint
Dependency	
A <i>quality model</i> is defined for a certain <i>entity class</i> .	Dependency.allInstances()-> exists(a: Dependency a.source.ocIsTypeOf(QualityModel) and a.target.ocIsTypeOf(EntityClass))
An <i>entity class</i> may have several <i>quality models</i> associated	Implicit in previous OCL Coinstraint
A <i>quality model</i> evaluates one or more <i>measurable concepts</i> .	Dependency.allInstances()-> exists(a: Dependency a.source.ocIsTypeOf(QualityModel) and a.target.ocIsTypeOf(MeasurableConcept))
A <i>measurable concept</i> is evaluated by one or more quality models .	Implicit in previous OCL Coinstraint
A <i>measurable concept</i> is associated with one or more <i>information needs</i> .	Dependency.allInstances()-> exists(a: Dependency a.source.ocIsTypeOf(MeasurableConcept) and a.target.ocIsTypeOf(InformationNeed))
An <i>information need</i> is related to one <i>measurable concept</i> .	Implicit in previous OCL Coinstraint
A <i>measure</i> is defined for one or more <i>attributes</i> .	Dependency.allInstances()-> exists(a: Dependency (a.source.ocIsTypeOf(derivedMeasure) or a.source.ocIsTypeOf(BaseMeasure) or a.source.ocIsTypeOf(Indicator)) and a.target.ocIsTypeOf(Attribute))
An <i>attribute</i> may have several associated <i>measures</i> .	Implicit in previous OCL Coinstraint
An <i>indicator</i> may satisfy several <i>information needs</i> .	Dependency.allInstances()-> exists(a: Dependency a.source.ocIsTypeOf(Indicator) and a.target.ocIsTypeOf(InformationNeed))

Every <i>information need</i> is satisfied by one or more <i>indicator</i> .	Implicit in previous OCL Coinstraint
An <i>analysis model</i> uses one or more <i>decision criteria</i> .	Dependency.allInstances()-> exists(a: Dependency a.source.oclIsTypeOf(AnalysisModel) and a.target.oclIsTypeOf(DecisionCriteria))
Every <i>decision criteria</i> is used in one or more <i>analysis models</i> .	Implicit in previous OCL Coinstraint

Table 8. SMML OCL Coinstraints.

5. Case of Study

To illustrate the benefits of the SMML, consider the following two case studies: the development and maintenance of database applications in a software company and the definition of a Data Quality Model for Web Portals.

5.1. Definition of a Relational Schemas Maintainability Measurement Model

The first case of study is detailed in [10]. This paper presents the results and lessons learned in the application of the Framework for the Modeling and Measurement of Software Processes (FMESP)[22] in a software company dedicated to the development and maintenance of software for information systems.

All the information concerning the problem is defined in each Software Measurement Package: Characterization and Objectives, Software Measures and Measurement Approaches. This case will show the modeling of the Characterization and Objectives, Software Measures and Measurement Approaches package. We wish to illustrate how a measurement model would be represented with SMML.

Figure 7 shows all the information that is needed to represent the Characterization and Objectives Instance. The *Measurement Elements* used are: *Information Need*, *Quality Model*, *Measurable Concept*, and *Attribute*. This model has been defined by using diagrams of UML objects.

We shall, furthermore, present how the same example would be defined with SMML (Figure 8).

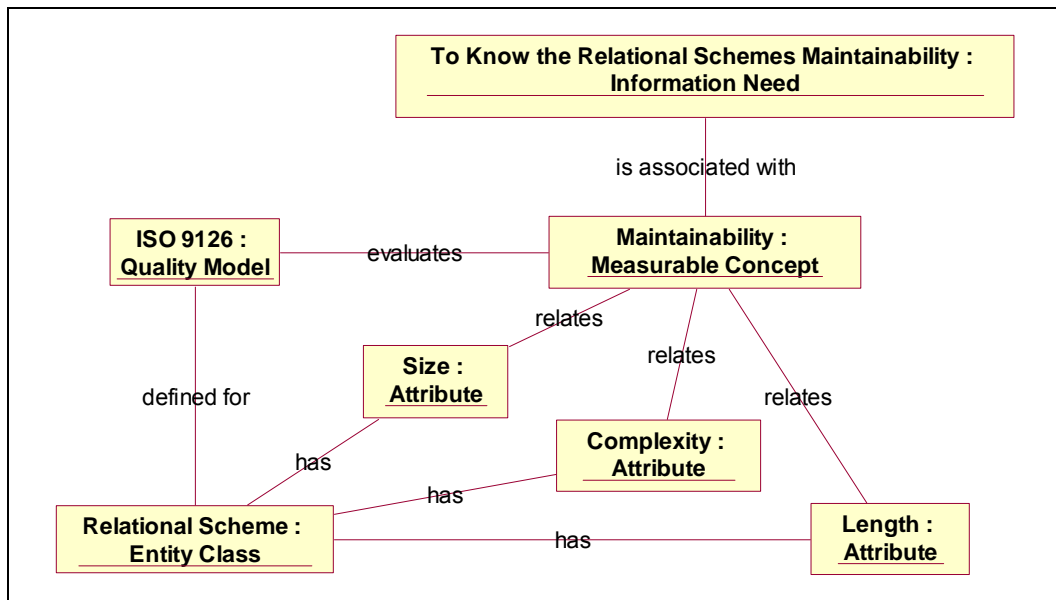


Figure 7. Characterization and Objectives Instance with UML.

As will be observed from the following figure, the representation is easier and more intuitive with the SMML language. Moreover, during the measurement model definition, no issues were found in the constructors metamodel, and no lacks were detected in the Language.

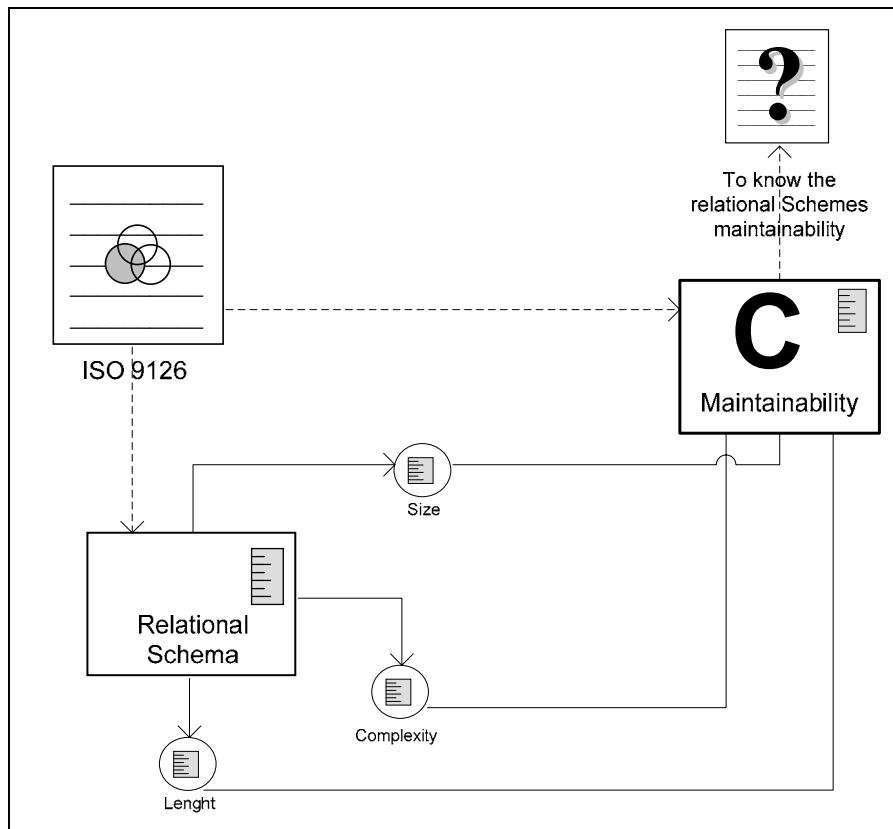


Figure 8. Characterization and Objectives Instance with SMML.

The next package to define is Software Measures Package. Figure 9 shows this information by using UML diagram. The *Measurement Elements* used are: *Attribute*, *Measure* (*Base Measure*, *Derived Measure* and *Indicator*), *Unit* and *Scale*.

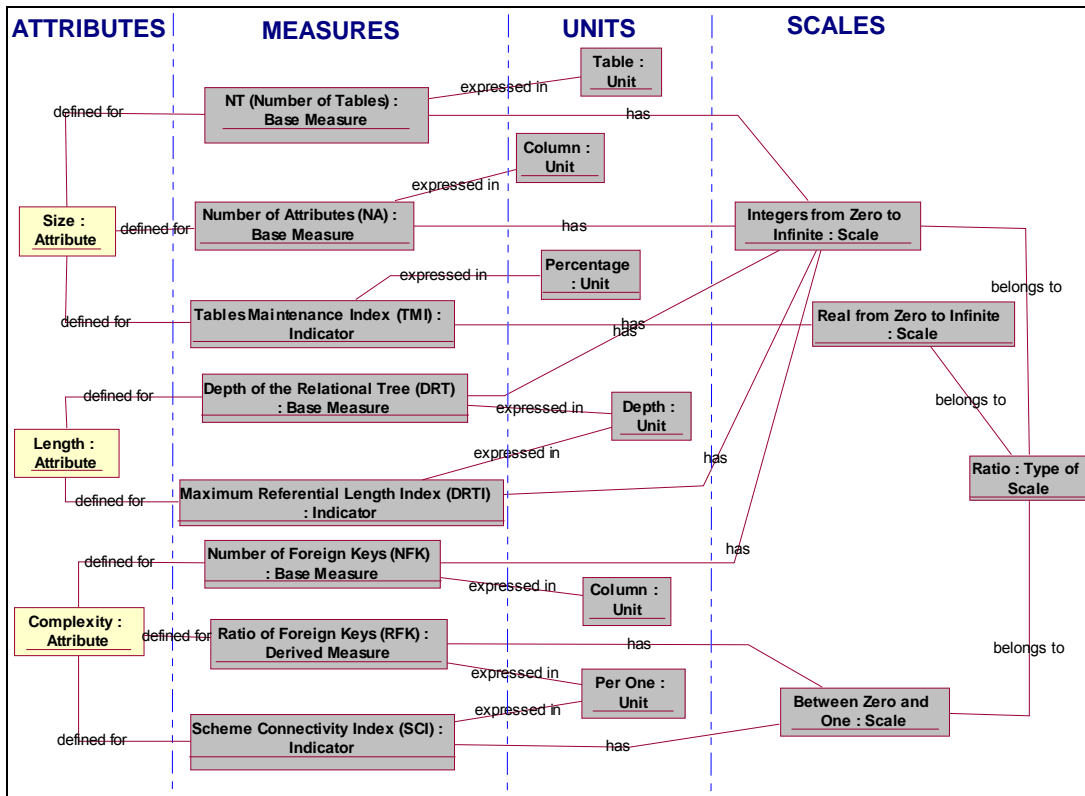


Figure 9. Software Measures Package Instance with UML.

Figure 10 represents the previous representation by using SMML.

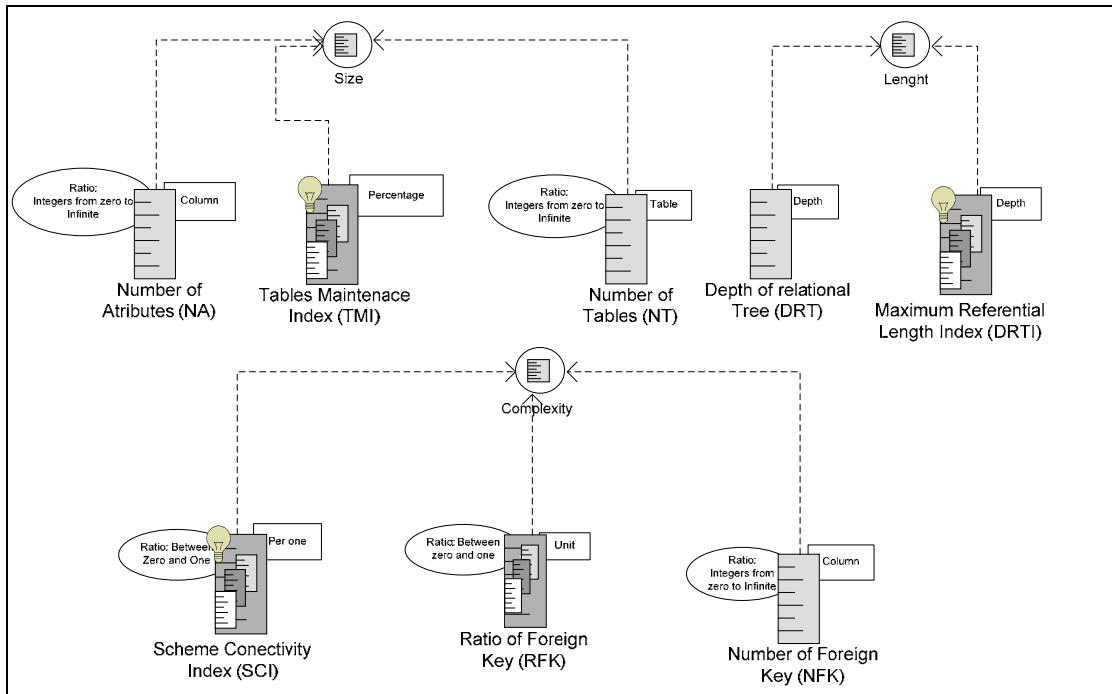


Figure 10. Software Measures Package Instance with SMML.

The last package to define is Measurement Approaches Package, all the elements are shown in Table 9 and Table 10.

Base Measure	Description	Measurement Method
NT	Number of <i>Tables</i> in the Scheme	To count the <i>Tables</i> in the schema
NFK	Number of <i>Foreign Keys</i> in the Scheme	To count the <i>Foreign Keys</i> in the schema
NA	Number of <i>Attributes</i> in the Scheme	To count the <i>Attributes</i> in the schema
DRT	Depth of Relational Tree (DRT). The number of tables which are part of the longest path obtained by following the referential integrity relationships between tables	To calculate the maximum depth of the paths obtained by following all the possible foreign keys in the schema

Table 9. Base Measures and Measurement Methods of the Relational Schemas Measurement Model.

Indicator	Description	Analysis Model	Decision Criteria
TMI	Tables Maintenance Index. This is obtained by establishing the proportion of attributes and tables in the relational scheme. The higher this measure is the more difficult it is to maintain the tables in the scheme	$TMI = NA/NT$	If $TMI > 18 \rightarrow TMI = \text{'Very High'}$ If $12 < TMI \leq 18 \rightarrow TMI = \text{'High'}$ If $6 < TMI \leq 12 \rightarrow TMI = \text{'Medium'}$ If $0 \leq TMI \leq 6 \rightarrow TMI = \text{'Low'}$
SCI	Scheme Connectivity Index. This is the proportion of foreign keys and tables in the scheme. The higher this measure is the more difficult it is to maintain the relational scheme.	$SCI = NFK/NT$	If $SCI \geq 2 \rightarrow SCI = \text{'Very High'}$ If $1,5 \leq SCI < 2 \rightarrow SCI = \text{'High'}$ If $1 < SCI < 1,5 \rightarrow SCI = \text{'Medium'}$ If $0,5 \leq SCI \leq 1 \rightarrow SCI = \text{'Low'}$ If $0 < SCI < 0,5 \rightarrow SCI = \text{'Very Low'}$
DRTI	Maximum Referential Length Index. Indicator based on the DRTI measure. The higher this measure is the more difficult it is to maintain the relational scheme	$DRTI = DRT$	If $DRTI > 15 \rightarrow DRTI = \text{'Very High'}$ If $8 < DRTI \leq 15 \rightarrow DRTI = \text{'High'}$ If $2 < DRTI \leq 8 \rightarrow DRTI = \text{'Medium'}$ If $0 \leq DRTI \leq 2 \rightarrow DRTI = \text{'Low'}$

Table 10. Indicators, Analysis Models and Decision Criteria of the Relational Schemas Measurement Model.

According Table 9 and Table 10, the Measurement Approaches Package Instance by using SMML is been defined in Figure 11.

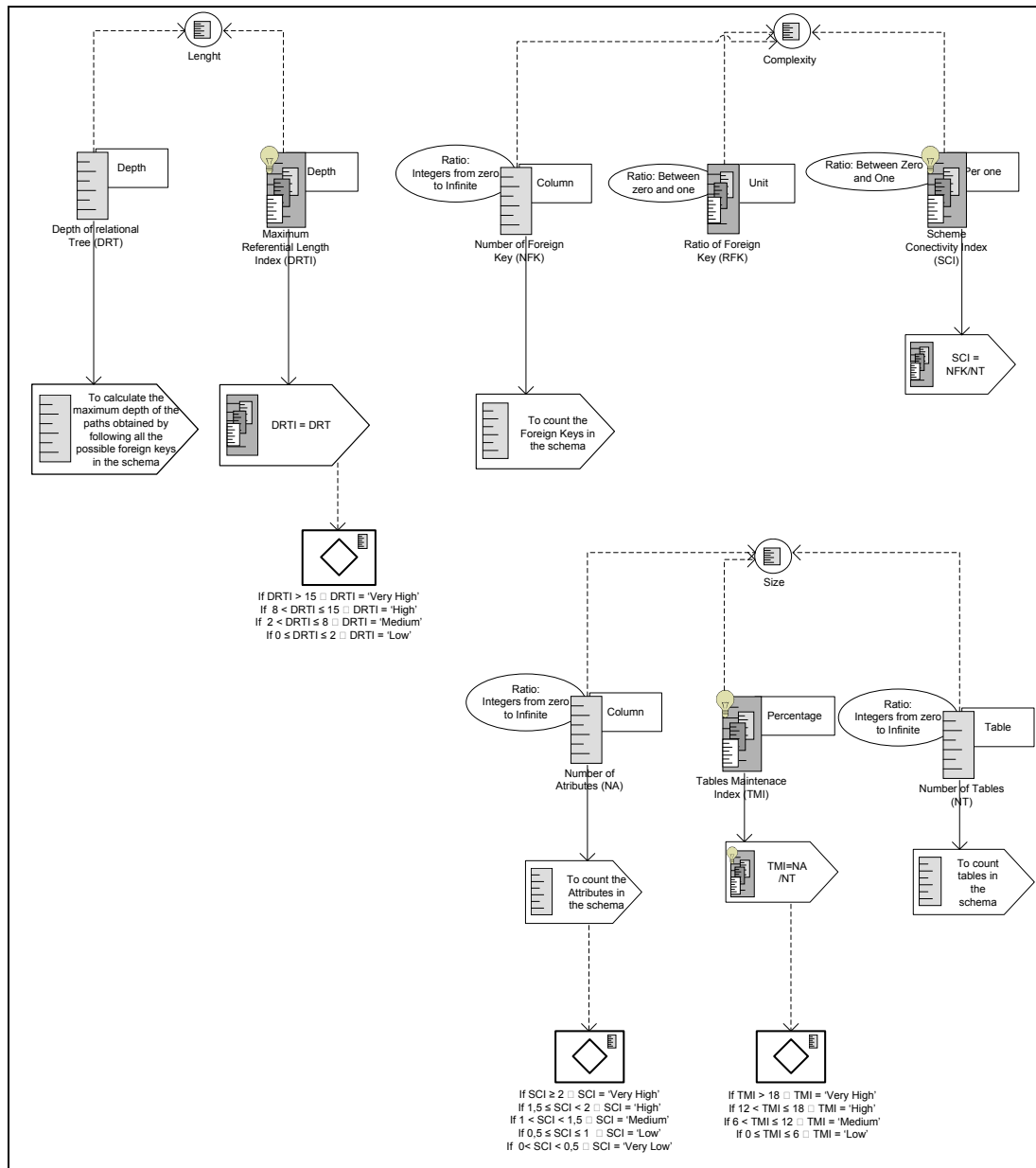


Figure 11. Measurement Approaches Package Instance.

5.2. Data Quality Model for Web Portals

The second case study is shown in [23]. This paper shows how the SMO can be instantiated to define a Data Quality Model for Web Portals, and can also be used to define a DSL for measuring software entities.

Figure 12 shows all the information that is needed to represent the Measurement Model of PDQM. The Measurement Elements used are: *Information Need*, *Quality Model*, *Measurable Concept*, *Attribute*, *Base Measure*, *Derived Measure*, *Indicator*, *Measurement Method*, *measurement Function* and *Analysis Model*.

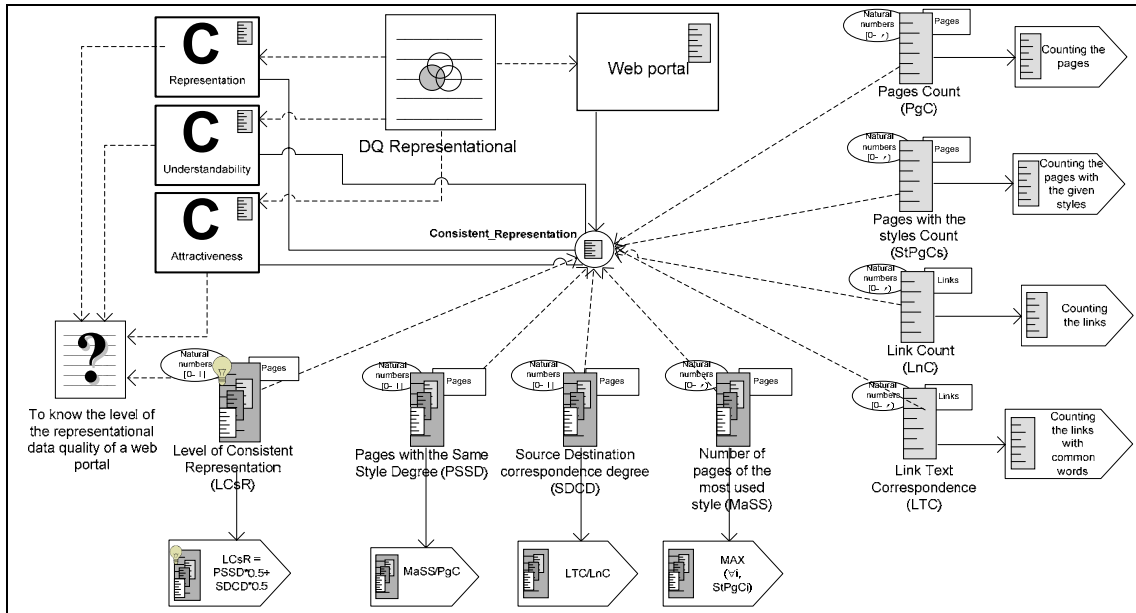


Figure 12. Measurement Model of PDQM represented with SMML.

In this case study, in spite of having to define numerous Measurement Elements, the representation continues to be easy and intuitive. What is more, it is easier to identify Measurement Elements by using this model than by using another General Purpose Language such as UML.

6. Conclusions and Future Work

SMML permits software measurement models to be defined in a manner which is easy and intuitive for the user. The set of icons which form a part of the language have been selected in order for them to be as familiar as possible to Software engineers. These engineers will thus be able to use the language to define measurement models with ease. The use of general purpose languages to define domain measurement models is thus avoided. Until this moment, no graphic representation permitting a better representation of the model was available.

SSML is a complete language, with a clear syntactic and semantic definition and a solid ontological base. With regard to expected requirements [29], we shall now show the requirements which are valid in our Language:

- **Conform:** the language constructs correspond to important domain concepts.
- **Orthogonal:** Each language construct is used to represent exactly one distinct concept (*Attribute*, *Base Measure*, etc.) in the domain.
- **Supportable:** The SMML language is supported by tools such as MS/DSL Tools or GMF [2].
- **Simple:** the DSL is simple in order to express the domain concepts and to support its users.
- **Usable:** DSL constructs are expressive and easy to understand.

SMML allows us to represent measurement models in various domains.

This language plays a fundamental role in SMF [35] as it allows us to define the measurement models which are the input for the software measurement process. The visual representation of the measurement models mean that SMF is a more usable and intuitive framework for the user. In other words, it makes the measurement process more comfortable.

Among related future works, one important work is that of the extension of SMMM with the con la jerarquía de *Measurement Approach* package hierarchy included in the included in Software Metrics Meta-Model [36].

We shall, moreover, test the usability of the language through a series of experiments based on the ISO 9126 standard. Our study will focus on usability and maintainability. Our idea is to select a group of modeling experts and to test the usability of this new language on them in order to define measurement models.

Finally, we shall apply SMF to real complex environments in order to obtain further refinements and validation.

ANEXES

A. Software Measurement Ontology

In this section Software Measuremen Ontology is presented.

To represent the SMO, REFSENO (Representation Formalism for Software Engineering Ontologies) [16] has been chosen. REFSENO provides constructs to describe concepts (each concept represents a class of experience items), their attributes, and relationships. The tables are used to represent these elements: the terms and the relationships. REFSENO also allows the description of similarity-based retrievals, and incorporates integrity rules such as cardinalities and value ranges for attributes, and assertions and preconditions on the elements instances. Several main reasons moved us to use REFSENO for defining our ontology.

The following tables provide a summary of the REFSENO representation of the Software Measurement Ontology by describing their concepts and relationships. For simplicity, all the SMO terms has been agruped in four tables (Tables 11-14), their relationships in four tables (Tables 15-18), and have omitted the description of the concepts' attributes. In addition, Figure 13 shows the graphical representation of the SMO terms and relationships, using the UML (Unified Modeling Language).

The SMO has been organized around four main sub-ontologies (see [21]):

- **Software Measurement Characterization and Objectives**, which includes the concepts required to establish the scope and objectives of the software measurement process. The main goal of a software measurement process is to satisfy certain information needs by identifying the entities (which belong to an entity class) and the attributes of these entities (which are the object of the measurement process). Attributes and information needs are related via measurable concepts (which belong to a quality model).
- **Software Measures**, which aim at establishing and clarifying the key elements in the definition of a software measure. A measure relates a defined measurement approach and a measurement scale (which belongs to a type of scale). A measure is expressed in a unit of measurement, and can be defined for more than one attribute. Three kinds of measures are defined: base measures, derived measures, and indicators.
- **Measurement Approaches**. This sub-ontology introduces the concept of measurement approach to generalize the different approaches used by the three kinds of measures for obtaining their respective measurement results. A base measure applies a measurement method. A derived measure uses a measurement function (which rests upon other base and/or derived measures). Finally, an indicator uses an analysis model (based on a decision criteria) to obtain a measurement result that satisfies an information need.
- **Measurement**. This establishes the terminology related to the act of measuring software. A measurement (which is an action) is a set of measurement results, for a given attribute of an entity, using a measurement approach. Measurement results are obtained as the result of performing measurements (actions).

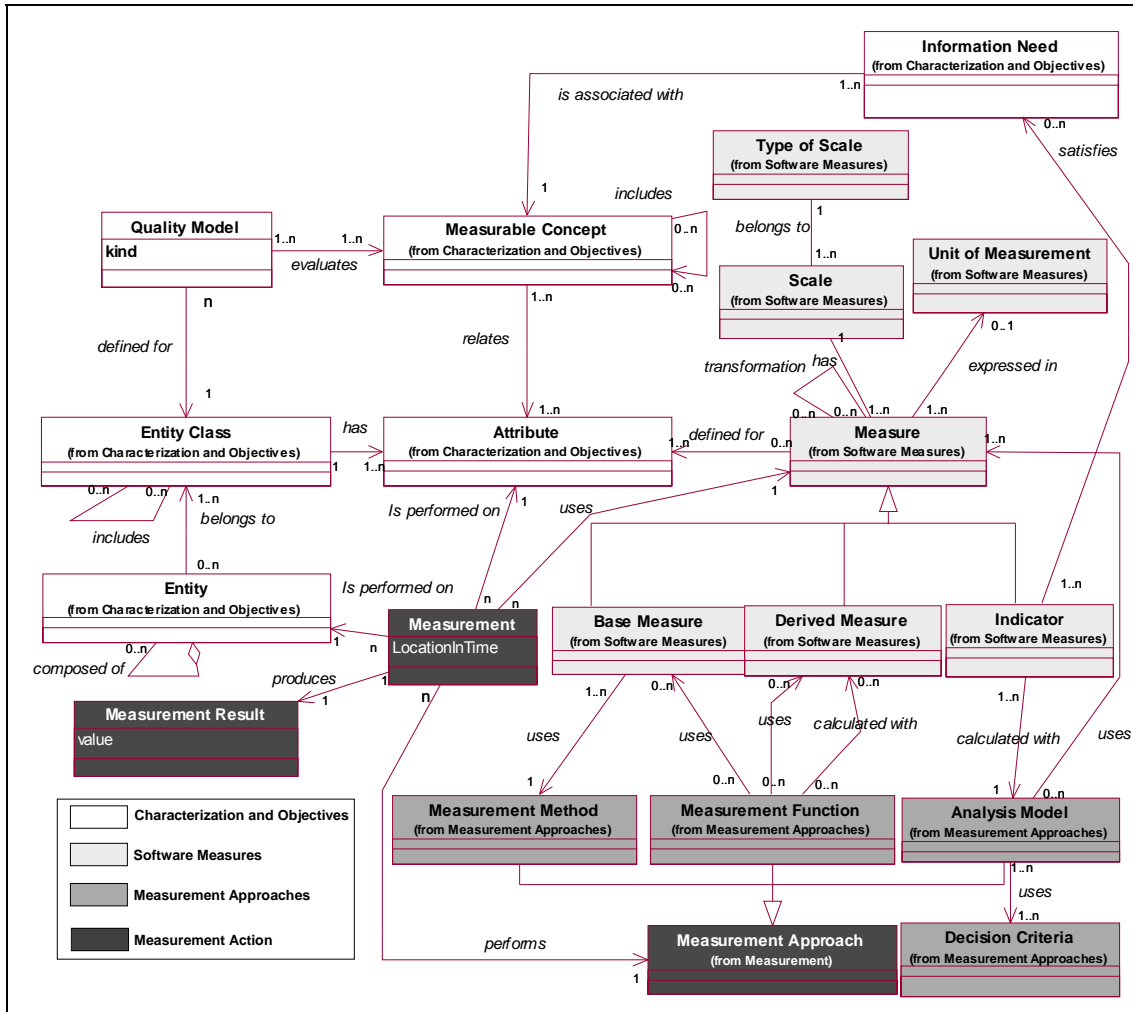


Figure 13. Software Measurement Ontology.

Term	Supercon	Definition	Source
Information Need	Concept	Insight necessary to manage objectives, goals, risks, and problems	15939
Measurable concept	Concept	Abstract relationship between attributes of entities and information needs	15939
Entity	Concept	Object that is to be characterized by measuring its attributes	15939
Entity class	Concept	The collection of all entities that satisfy a given predicate	New
Attribute	Concept	A measurable physical or abstract property of an entity, that is shared by all the entities of an entity class	Adapted from 14598
Quality Model	Concept	The set of measurable concepts and the relationships between them which provide the basis for specifying quality requirements and evaluating the quality of the entities of a given entity class	Adapted from 14598

Table 11. Definition of the terms in the “software measurement characterization and objectives” sub-ontology

Term	Supercon	Definition	Source
Measure	Concept	La <i>forma de medir</i> (método de medición, función de cálculo o modelo de análisis) y la <i>escala de medición</i> .	Adaptado de FD4 “métrica”
Scale	Concept	Un conjunto de valores con propiedades definidas.	FD4
Type of scale	Concept	Indica la naturaleza de la relación entre los valores de la escala	FD6
Unit of measurement	Concept	Una cantidad particular, definida y adoptada por convención, con la que se puede comparar otras cantidades de la misma clase para expresar sus magnitudes respecto a esa cantidad particular	FD5
Base measure	Measure	Una medida de un <i>atributo</i> que no depende de ninguna otra medida, y cuya <i>forma de medir</i> es un <i>método de medición</i> .	Adaptado de FD4 “métrica directa”
Derived measure	Measure	Una medida que es derivada de otra medida base o derivada, utilizando una <i>función de cálculo</i> como <i>forma de medir</i> .	Adaptado de FD4 “métrica indirecta”
Indicator	Measure	Una medida que es derivada de otras medidas utilizando un <i>modelo de análisis</i> como <i>forma de medir</i> .	Nuevo

Table 12. Definition of the terms in the “software measures” sub-ontology.

Term	Supercon	Definition	Source
Measurement method	Measurement Approach	Logical sequence of operations, described generically, used in quantifying an attribute with respect to a specified scale. (A measurement method is the measurement approach that defines a base measure)	Adapted from 15939
Measurement function	Measurement approach	An algorithm or calculation performed to combine two or more base or derived measures. (A measurement function is the measurement approach that defines a derived measure)	Adapted from 15939
Analysis model	Measurement Approach	Algorithm or calculation combining one or more measures with associated decision criteria. (An analysis model is the measurement approach that defines an indicator)	Adapted from 15939
Decision criteria	Concept	Thresholds, targets, or patterns used to determine the need for action or further investigation, or to describe the level of confidence in a given result	15939

Table 13. Definition of the terms in the “measurement approaches” sub-ontology.

Term	Supercon	Definition	Source
Measurement approach	Concept	Sequence of operations aimed at determining the value of a measurement result. (A measurement approach is either a measurement method, a measurement function or an analysis model)	new
Measurement	Concept	A set of operations having the object of determining a value of a measurement result, for a given attribute of an entity, using a measurement approach	Adapted from VIM
Measurement result	Concept	The number or category assigned to an attribute of an entity by making a measurement	Adapted from 14598 'Measure'

Table 14. Definition of the terms in the “measurement action” sub-ontology.

Relationships	Concepts	Description
Includes	Entity class–entity class	An entity class may include several other entity classes. An entity class may be included in several other entity classes
Defined for	Quality model–entity class	A quality model is defined for a certain entity class. An entity class may have several quality models associated
Evaluates	Quality model–measurable concept	A quality model evaluates one or more measurable concepts. A measurable concept is evaluated by one or more quality models
Belongs to	Entity–entity class	An entity belongs to one or more entity classes. An entity class may characterize several entities
Relates	Measurable concept–attribute	A measurable concept relates one or more attributes
Is associated with	Measurable concept–information Need	A measurable concept is associated with one or more information needs. An information need is related to one measurable concept
Includes	Measurable concept–measurable Concept	A measurable concept may include several measurable concepts. A measurable concept may be included in several other measurable concepts
Composed of	Entity–entity	An entity maybe composed of several other entities
Has	Entity class–attribute	An entity class has one or more attributes. An attribute can only belong to one entity class

Table 15. Relationships in the ‘software measurement characterization and objectives’ sub-ontology.

Relationships	Concepts	Description
Belongs to	Scale–type of scale	Every scale belongs to a type of scale. A type of scale may characterize several scales
Defined for	Measure–attribute	A measure is defined for one or more attributes. An attribute may have several associated measures
Transformation	Measure–measure	Two measures can be related by a transformation function; the kind of function will depend on the scale types of the scales
Expressed in	Measure–unit of measurement	A measure is expressed in one unit of measurement (only for measures whose type is interval or ratio). A unit of measurement is used to express one or more measures of interval or ratio types
Has	Measure–scale	Every measure has a scale. A scale may serve to define more than one measures

Table 16. Relationships in the ‘software measures’ sub-ontology.

Relationships	Concepts	Description
Calculated with	Derived measure–measurement function	Every derived measure is calculated with one measurement function. Every measurement function may define one or more derived measures
Calculated with	Indicator–analysis model	Every indicator is calculated with one analysis model. Every analysis model may define one or more indicators
Uses	Base measure–measurement method	Every base measure uses one measurement method. Every measurement method defines one or more base measures
Satisfies	Information need–indicator	An indicator may satisfy several information needs. Every information need is satisfied by one or more indicators
Uses	Measurement function–derived measure	A measurement function may use several derived measures. A derived measure may be used in several measurement functions
Uses	Measurement function–base measure	A measurement function may use several base measures. A base measure may be used in several measurement functions
Uses	Analysis model–measure	An analysis model uses one or more measures. A measure may be used in several analysis models
Uses	Analysis model–decision criteria	An analysis model uses one or more decision criteria. Every decision criteria is used in one or more analysis models

Table 17. Relationships in the ‘measurement approaches’ sub-ontology.

Relationships	Concepts	Description
Performs	Measurement– measurement approach	A measurement is the action of performing a measurement approach (the kind of measurement approach will be dictated by the kind of measure used for performing the measurement). A measurement approach may be used for performing several measurements
Produces	Measurement– measurement result	Every measurement produces one measurement result. Every measurement result is the result of one measurement
Is performed on	Measurement–attribute	Every measurement is performed on one attribute of an entity (the attribute should be defined for the entity class of the entity)
Is performed on	Measurement–entity	Every measurement is performed on an entity, through one of its attributes (the attribute should be defined for the entity class of the entity)
Uses	Measurement–measure	Every measurement uses one measure. One measure may be used in several measurements

Table 18. Relationships in the ‘measurement’ sub-ontology.

B. Software Measurement Framework

In order to carry out this proposal it was considered of interest to adapt FMESP to the MDE paradigm. The objective of this was to exploit the benefits that the paradigm could contribute to software measurement by, on one hand adopting the software measurement metamodel defined in FMESP, and on the other by evolving GenMETRIC to an environment which would allow the definition of software measurement models and the computation of the models defined. All this would take place within the context of models and model transformations of the MDA architecture. The Software Measurement Framework (SMF) is the evolution of the FMESP, but is adapted to the MDE paradigm and uses MDA technology.

The following subsections explain the conceptual, technological and methodological elements which are part of SMF.

Conceptual Architecture

Due to the necessity of having a generic and homogeneous environment for software measurement [20, 22, 24], a conceptual architecture and a tool with which to integrate the software measurement are proposed. In the following section, the main characteristics of this proposal are described. In [24] a more detailed description can be found.

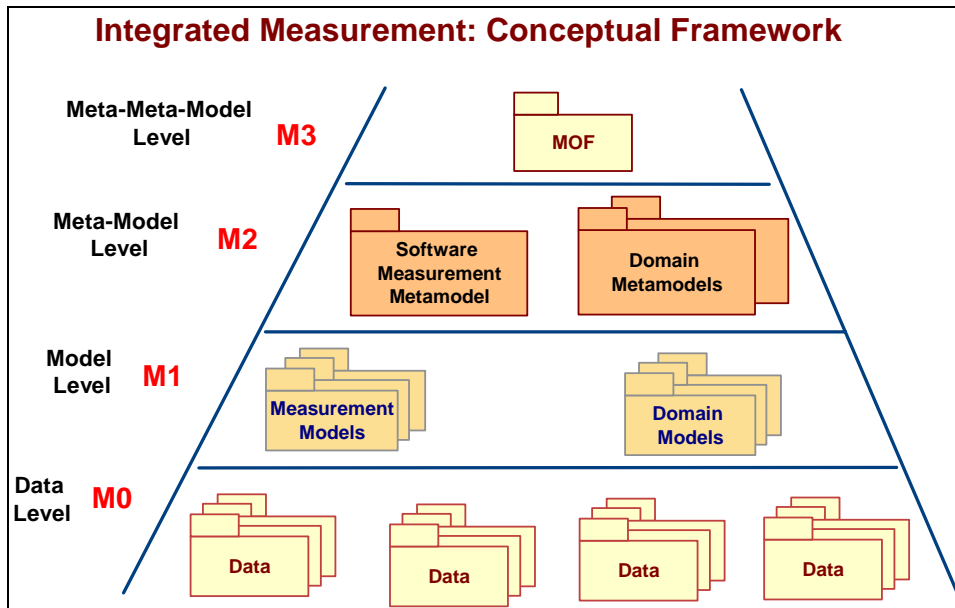


Figure 14. Conceptual framework with which to manage software measurement.

The proposed software measurement described in this paper is part of the FMESP framework [22]. The FMESP framework permits representing and managing software processes from the perspectives of modeling and measurement. We focus on the measurement support of the framework whose elements are detailed according to the three layers of abstraction of metadata that they belong to, according to the MOF standard. In Figure 14, the conceptual architecture for integrated measurement is represented.

As can be observed in Figure 14, the architecture has been organized into the following conceptual levels of metadata:

- Meta-MetaModel Level (M3). At this level, an abstract language for the definition of metamodels, is found. This is the MOF language.
- Metamodel Level (M2). In the M2 level, two generic metamodels which conform with this framework are required. These are: the Measurement Metamodel, to define specific measurement models; and Domain Metamodels, to represent the kinds of entities which are candidates for measurement in the context of the evaluation of the software processes, such as, UML and Process metamodels.
- Model Level (M1). Specific models are included at this level. These models may be of two types: Measurement Models, which are examples of the measurement metamodel in the M2 level and which are defined in such a way as to satisfy some of the company's information needs; and Domain Models, which are defined according to their corresponding domain metamodels.

In order to establish and clarify the concepts and relationships that are involved in the software measurement domain before designing the metamodel, an ontology for software measurement was developed [20]. The measurement metamodel was derived by using the concepts and relationships stated in the ontology as a base. The Software Measurement metamodel (which is integrated in SMF) is organized around four main packages (for greater detail see [20]) which correspond with the four sub-ontologies described in ANEXE A.

Technological aspects

In this section the technological aspects of SMF are explained.

Adaptation to MDA

In Figure 15 the necessary elements for the FMESP adaptation to MDA are presented according to MOF levels.

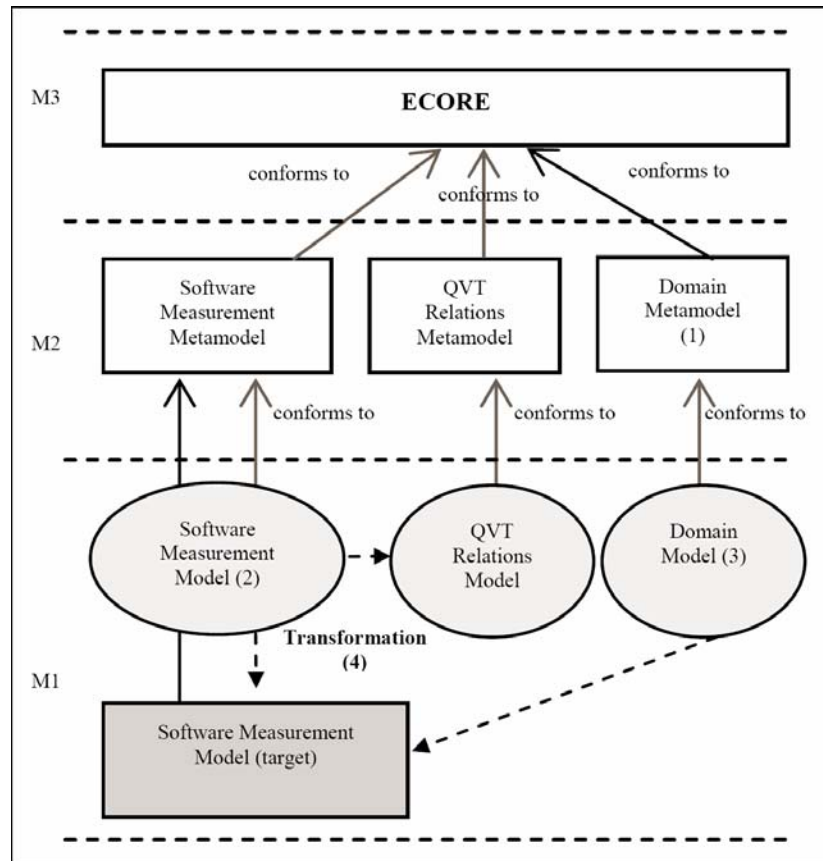


Figure 15. Elements of the FMESP adaptation in a MDA context.

As can be observed in Figure 15, two new elements, namely the QVT Relations model and metamodel, have been added to adapt the conceptual architecture illustrated Figure 14 to MDA. The QVT Relations Model (which is described in greater detail in the following section) is obtained automatically through a transformation from a Measurement model. It contains all the information necessary to carry out the transformation of the SMF proposal. Ecore language has been selected because it is a common modeling language based on EMOF. EMOF is the part of the MOF 2.0 specification that is used for defining simple metamodels using UML-like concepts.

QVT Relations transformation

The QVT Relations model is the transformation needed to perform the measurement. In this transformation two source models are involved: a Software Measurement model and a domain model; the target model is the Software Measurement Model with the measurement results (see Figure 15). Due to the fact that the proposal is about generic measurement, it is very important that the QVT model is obtained in a generic way. The MDE paradigm and MDA technology are applied for this reason.

This transformation is obtained automatically from the previous QVT transformation shown in Figure 16. The QVT Relations model, called the extended or final QVT Relations model, is obtained from a QVT transformation, where there are two source models: the basic or initial QVT Relations model (which conforms to the QVT Relations metamodel) and the Software Measurement model (previously defined).

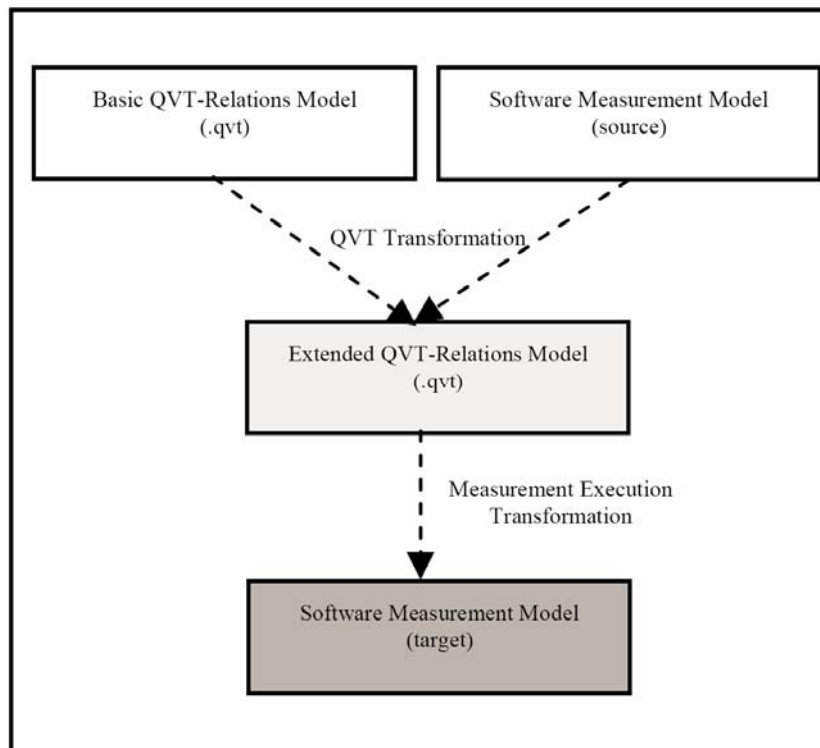


Figure 16. QVT Relations transformation model.

The extended QVT Relations model extends the basic QVT Relations model with the following aspects:

- **Transformation Model:** to obtain the extended QVT Relations model, the source model specification is needed. In this case, there are two source models: the Software Measurement model and the domain model. Due to the fact that the Software Measurement model is always the same, this model is already defined in the basic QVT Relations model. Therefore, only the domain model needs to be defined. This information is taken from the Software Measurement model which contains all the measurement information.
- **Relation Domain:** in order to perform the transformation, it is necessary to define the *checkonly domain* rules. In this case there are two, one for each source model: the domain model and the Software Measurement model. It is only necessary to define *checkonly domain* of the domain model, because *checkonly domain* of the measurement model is already defined in the basic QVT Relations model.

- **Function:** this contains the necessary OCL queries to carry out the measurement. These OCL queries are the implementations of the “*Measurement Action*” package defined in the Software Measurement Metamodel.

These elements are empty in the basic QVT Relations model, and they are extended to obtain the extended QVT Relations model, the transformation model necessary to carry out the measurement. In the Figure 17 all the Software Measurement process is shown.

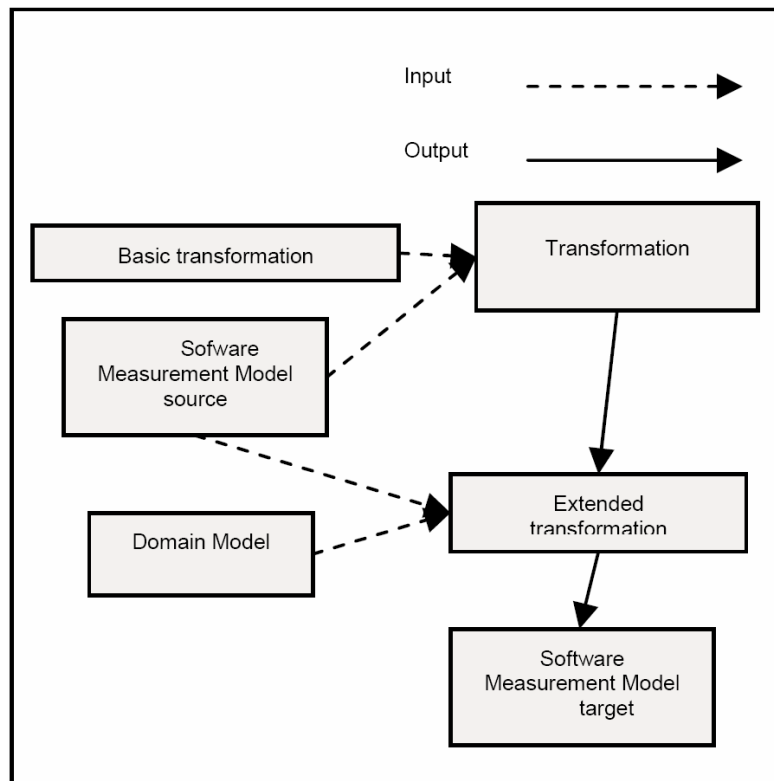


Figure 17. Software Measurement process.

Technological environment

The tool selected has been the model management environment called MOMENT (MOment manageMENT)[7]. This framework is integrated in the Eclipse platform. It provides a set of generic operators to deal with models through the Eclipse Modeling Framework (EMF)[3]. The underlying formalism of the model management approach is the algebraic language Maude [4].

From a functional point of view, MOMENT has two components: OCL query execution (MOMENT-OCL) and QVT Transformations (MOMENT-QVT).

MOMENT-OCL [8] has implemented an editor integrated in the Eclipse platform to check OCL invariants and to execute OCL queries over instances of ecore models. It uses Ecore models in the entire software development process to store the OCL expressions by following the Model Driven Engineering approach. One advantage of this is the persistence mechanisms in XMI that EMF provides automatically. In this work, it has been used in order to check and validate the OCL queries used in the QVT transformations. The results have been shown by screen.

On the other hand, the MOMENT-QVT tool [37] is a model transformation engine that provides partial support for the QVT Relations language. It implements the metamodel definition QVT, given in the QVT standard, and provides an editor for the QVT

Relations language, which permits the definition of model transformations between EMF metamodels.

In order to carry out a QVT transformation in MOMENT, a transformation textual specification (coded by the Textual QVT Editor and stored in a .qvtext) or, its equivalent QVT Relations model (stored in a .qvt) can be used. This model conforms to the QVT Relations metamodel and it is possible to obtain it by parsing the textual specification.

Method

The necessary steps to carry out the software measurement by using the SMF are explained below (see Figure 15):

1. **Incorporation of domain metamodel:** the measurement is made in a specific domain. This domain must be defined according to its metamodel (it is situated in the M2 level and it conforms to the Ecore meta-metamodel).
2. **Creation of measurement model:** the measurement model is created according to the Software Measurement metamodel which is integrated in SMF. This first model is the source model, so the results are therefore still not defined, i.e. the “*Measurement Action*” package from the Software Measurement metamodel is still not instantiated.
3. **Creation of domain model:** which is defined according to its corresponding domain metamodel (created in the first step). The domain models are the entities whose attributes are measured by calculating the measurements defined in the corresponding measurement models. Examples of domain models are: the UML models (use cases, class diagrams, etc.), or the E/R models.
4. **Measurement execution:** the measurement execution is carried out through QVT transformation, in which, the measurement model is obtained by starting from the two source models (the measurement model and the domain model) where the results are defined, i.e. the “*Measurement Action*” package is instantiated. The target measurement model is the extension of the source measurement model. The measurement results are calculated by running OCL queries on the domain model.

An example of the method application is shown in the following section.

References

- [1] *DSM (Domain-Specific Modeling) Forum Main Page*. <http://www.dsmforum.org/>. (2007).
- [2] *Eclipse Graphical Modeling Framework (GMF) Main Page*. <http://www.eclipse.org/gmf/>. (2007).
- [3] *Eclipse Modelling Framework (EMF) Main Page*, Eclipse Tools. EMF Home. <http://www.eclipse.org/emf>. (2007).
- [4] *The Maude System*, Department of Computer Science, University of Illinois, Urbana-Champaign. <http://maude.cs.uiuc.edu/>. (2007).
- [5] *Microsoft Visual Studio 2005 SDK including Domain-Specific Language Tools Main Page*. <http://msdn.microsoft.com/vstudio/DSLTools/>. (2007).
- [6] Allen, N. A., Shaffer, C. A. y Watson, L. T.; *Building modeling tools that support verification, validation, and testing for the domain expert*, 37th conference on Winter simulation. Orlando, Florida. (2005). pp.419-426.
- [7] Boronat, A. y Meseguer, J., *Algebraic Semantics of EMOF/OCL Metamodels*, Technical Report UIUCDCS-R-2007-2904, CS Dept., University of Illinois at Urbana-Champaign. (2007). http://www.cs.le.ac.uk/people/ab373/papers/20070606_UIUC-TR-MOF-OCL-Boronat-Meseguer.pdf.
- [8] Boronat, A., Ramos, I. y Carsí, J. Á.; *Definition of OCL 2.0 Operational Semantics by means of a Parameterized Algebraic Specification*, WAFOCA'06. First International Workshop on algebraic foundations for OCL and Applications. (2006).
- [9] Briand, L. C., Morasca, S. y Basili, V. R.; *An Operational Process for Goal-Driven Definition of Measures*, IEEE Trans. Softw. Eng. 28. (2002). pp.1106-1125.
- [10] Canfora, G., García, F., Ruiz, F. y Visaggio, C. A.; *Applying a framework for the improvement of software process maturity*, Software: Practice & Experience. 36. (2006). pp.283-304.
- [11] Czarnecki, K. y Eisenecker, U., *Generative Programming: Methods, Tools, and Applications*, (2000).
- [12] Champeaux, D., *Object-oriented Development Process and Metrics*, (1997).
- [13] Christensen, N. H., *Domain-Specific Languages in Software Development and the relation to partial evaluation*, Department. of Computer Science, University of Copenhagen, Denmark, 2003.
- [14] Feilkas, M.; *How to represent Models, Languages and Transformations?*, Proceedings of th 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06). (2006). pp.204-213.
- [15] Fenton, N. y Pfleeger, S. L., *Software Metrics: A Rigorous & Practical Approach, Second Edition*, (1997).
- [16] Fowler, M.; *Language Workbenches: The Killer-App for Domain Specific Languages?*, (2005).
- [17] Fowler, M.; *DSL Boundary*. <http://martinfowler.com/bliki/DslBoundary.html>. (2006).
- [18] France, R. B., Ghosh, S. y Dinh-Trong, T.; *Model-Driven Development Using UML 2.0: Promises and Pitfalls*, Computer. 39. (2006). pp.59-66.
- [19] Furtado, A. W. B., *Sharpludus: improving game development experience through software factories and domain-specific languages*, Universidade Federal

- de Pernambuco (UFPE) Mestrado em Ciência da Computação centro de Informática (CIN), 2006.
- [20] García, F., Bertoa, M. F., Calero, C., Vallecillo, A., Ruíz, F., Piattini, M. y Genero, M.; *Towards a consistent terminology for software measurement*, Information and Software Technology 48. (2006). pp.631-644
- [21] García, F., Bertoa, M. F. y Vallecillo, A., *An ontology for software measurement, Technical Report*, UCLM DIAB-04-02-2, (2004).
- [22] García, F., Piattini, M., Ruiz, F., Canfora, G. y Visaggio, C. A.; *FMESP: Framework for the modeling and evaluation of software processes*, Journal of Systems Architecture - Agile Methodologies for Software Production 52. (2006). pp.627-639
- [23] García, F., Ruiz, F., Calero, C., Bertoa, M. F., Vallecillo, A., Mora, B. y Piattini, M.; *On the Effective Use of Ontologies in Software Measurement*, JERJ (sended). (2008).
- [24] García, F., Serrano, M., Cruz-Lemus, J., Ruiz, F. y Piattini, M.; *Managing Software Process Measurement: A Metamodel-Based Approach*, Information Sciences. (2007).
- [25] Greenfield, J.; *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*, Third International Conference, GPCE 2004. Vancouver, Canada. (2004). pp.488.
- [26] ISO/IEC, *ISO 15939: Software Engineering - Software Measurement Process.*, (2002).
- [27] ISO/IEC; *Software and Systems Engineering - Guidelines for the application of ISO/IEC 9001:2000 to Computer Software*, International Standards Organization. (2004).
- [28] Kelly, S.; *Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM*, Proceedings of the OOPSLA & GPCE Workshop on Best Practices for Model Driven Software Development at OOPSLA'04. (2004).
- [29] Kolovos, D. S., Paige, R. F., Kelly, T. y Polack, F. A. C.; *Requirements for Domain-Specific Languages*, First ECOOP Workshop on Domain-Specific Program Development (ECOOP'06). Nantes, France. (2006).
- [30] Kurtev, I., Bézivin, J., Jouault, F. y Valduriez, P.; *Model-based DSL Frameworks*, (2006).
- [31] MacDonell, S. G., Shepperd, M. J. y Sallis, P. J.; *Metrics for Database Systems: An Empirical Study*, 4th International Symposium on Software Metrics. IEEE Computer Society. Albuquerque. (1997).
- [32] McGarry, J., Card, D., Jones, C., Layman, B., Clark, E., Dean, J. y Hall, F., *Practical Software Measurement. Objective Information for Decision Makers*, (2002).
- [33] Mernik, M., Heering, J. y Sloane, A. M.; *When and how to develop domain-specific languages*, ACM Computing Surveys (CSUR). Volume 37. (2005). pp.316-344.
- [34] Mora, B., García, F., Ruiz, F., Piattini, M., Boronat, A., Gómez, A., Carsí, J. Á. y Ramos, I.; *Marco de Trabajo basado en MDA para la Medición Genérica del Software*, Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos, JISBD'2007. Zaragoza. (2007). pp.211-220.
- [35] Mora, B., García, F., Ruiz, F., Piattini, M., Boronat, A., Gómez, A., Carsí, J. Á. y Ramos, I.; *Software Measurement by using QVT Transformation in an MDA context*, ICEIS 2008 (In Press). Barcelona (Spain). (2008).

- [36] OMG, *Architecture-Driven Modernization (ADM): Software Metrics Meta-Model (SMM)*. *OMG Document: dmtf/2007-08-01*, Object Management Group. (2007).
- [37] Queralt, P., Hoyos, L., Boronat, A., Carsí, J. Á. y Ramos, I.; *Un motor de transformación de modelos con soporte para el lenguaje QVT relations*, Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06). Sitges, Spain. (2006).
- [38] Völter, M., *A categorization of DSLs*, (2006). <http://www.voelterblog.blogspot.com/2006/10/categorization-of-dsls.html>.
- [39] Völter, M., Stahl, T., Bettin, J., Haase, A. y Helsen, S., *Model-Driven Software Development: Tecnology, Engineering, Management*, (2006).