Automatic Spreadsheet Generation from Conceptual Models

Léo Antunes, Alexandre Corrêa, and Márcio Barros
Federal University of Rio de Janeiro State (UNIRIO)
Rio de Janeiro, Brazil
{leo.antunes, alexandre.correa, marcio.barros}@uniriotec.br

Abstract **-** The flexibility and ease-of-use of spreadsheets allow users with little CS background to build computational solutions to solve their problems. However, inexperienced users may build erroneous spreadsheets. Modeling is commonly used in software development to allow users to visualize, communicate, and validate different aspects of a system before starting its construction. This work proposes a Model-Driven Engineering (MDE) approach that automatically generates spreadsheets from conceptual models aiming to reduce the number of errors introduced by users while using spreadsheets. Initially, we designed and executed an empirical study to identify the most common errors users commit when developing spreadsheets. The results of this study have shown that even users who classify themselves as experienced spreadsheet developers commit a large number of errors while developing a spreadsheet. Also, it has shown that formula-related errors are the most common errors introduced by users (55%). A second study involved humans using a spreadsheet generated by our approach and showed initial evidence that the approach can build spreadsheets that avoid the introduction of several types of errors often present in spreadsheets manually created by users. In this sense, we observed 98% correct answers to 234 questions involving spreadsheet manipulation.

Keywords: *spreadsheet, conceptual model, model-driven engineering*

I. INTRODUCTION

The flexibility and ease-of-use of spreadsheets allow users with little computing knowledge to build computational solutions for their problems. The downside of these features is that errors are commonly found in spreadsheets, especially those built or maintained by less experienced users. Studies reported the presence of errors in more than 90% of spreadsheets collected from distinct samples [1] [2] [3] [4]. Such mistakes may lead to problems for individuals and businesses, as shown by several cases: (i) a coding error in a spreadsheet caused a Harvard economics study to overstate the impact that debt burdens have on a nation's economic growth[1]; (ii) errors in spreadsheets have been reported as one of the reasons behind the so-called London Whale incident at J.P. Morgan, involving a $6.2 billion trading loss[2]; and (iii) the collapse of the Jamaican banking system in the late 90's has been directly related to spreadsheets errors[3].

[1] http://www.nextnewdeal.net/rortybomb/researchers-finally-replicated-reinhart-rogoff-and-there-are-serious-problems
[2] http://files.shareholder.com/downloads/ONE/2261602328x0x628656/4cb574a0-0bf5-4728-9582-625e4519b5ab/Task_Force_Report.pdf
[3] http://arxiv.org/abs/0908.4420

Spreadsheets are typically developed by end-users. A significant factor that contributes to the difficulty of software development is the conceptual gap between the problem and the implementation domains (e.g., identifying a problem and developing a spreadsheet to address it). Models are widely used in software engineering as they support visualization, communication, and validation of different aspects of a system before its development. Model-driven engineering (MDE) is an approach concerned with reducing the problem-to-implementation gap through the use of models that describe the system at multiple levels of abstraction and through automated support for transforming and analyzing models. In a model-driven perspective of software development, models are the primary artifacts to be created by developers, who rely on computer-based technologies to transform such models into running systems [5].

This paper presents a MDE approach to the generation of spreadsheets from conceptual models. It comprises a set of transformations applied to a conceptual model representing the problem addressed by the spreadsheet. These transformations are applied in two steps. First, a conceptual model defining classes, attributes, associations, and query operations is transformed into a spreadsheet model that defines abstract components of a target spreadsheet, e.g., worksheets, tables and columns. Next, the abstract spreadsheet model is transformed into a concrete implementation, e.g. an Excel spreadsheet. The conceptual model is a UML class model augmented with OCL expressions, which precisely define computation rules associated to derived attributes and query operations. These expressions are transformed into formulas in the target spreadsheet, allowing the generation of error-free spreadsheets from correct conceptual models.

Our major contribution is the generation of formulas from OCL expressions, since former work on the field were most concerned with generating the structure of the spreadsheet from high-level diagrams without filling calculated cells with their respective formulas. Our interest in spreadsheet formulas is due to the results of an empirical study that was designed and executed to determine the most common errors in spreadsheet development. This study found that about 55% of the errors committed by spreadsheet developers are related to formulas. Moreover, even self-declared experienced users may introduce errors in spreadsheet formulas. Therefore, an automatic approach to generate formulas might help to reduce the number of errors introduced by end-users while updating a

978-1-4673-9272-3/15 $31.00 © 2015 IEEE
DOI 10.1109/SBES.2015.11

spreadsheet. We have evaluated the usefulness of the proposed approach through a second study designed to determine whether a spreadsheet generated by our approach would avoid the introduction of certain errors by users. Results from this study provide initial evidence that users might indeed benefit from an automated approach for generating spreadsheets.

This paper is organized as follows: section II presents the experimental study designed to evaluate the most frequent types of errors introduced in spreadsheets. Section III presents an overview of the proposed approach. Section IV describes a set of transformations used to generate an abstract spreadsheet model from UML class diagrams. Section V presents a set of transformations that allows generating spreadsheet formulas from OCL expressions. Section VI describes the evaluation of the proposed approach. Section VII discusses related work. Section VIII discusses the requirements to apply the approach in practice and concluding remarks are drawn in section VIII.

II. Evaluating the Types of Errors in Spreadsheets and their Frequencies

We designed an experimental study to determine the most frequent errors made by users while manually developing a simple spreadsheet. We proposed two research questions:

RQ1: What are the most common errors found in manually built spreadsheets?

RQ2: Are these errors and their frequency related to the experience of the spreadsheet developer?

To evaluate these questions, we asked a group of 20 subjects with differing experience in spreadsheet development to build a spreadsheet for an inventory control system. The case study depicts a supermarket chain with several stores, each divided into a set of departments responsible for a subset of the store's products. The spreadsheet to be developed must control the inventory of products available on each store, along with purchases from suppliers and sales to clients.

Subjects included undergraduate students and practitioners from the IT sector. We used a questionnaire to classify their experience with spreadsheets. Each subject chose one of the following options: (i) I never created a spreadsheet; (ii) I created spreadsheets using basic aggregation and selection formulas; (iii) I created spreadsheets using complex features, such as looking formulas and data validation; and (iv) I frequently deal with highly complex spreadsheets in my work.

Subjects received a textual description of the requirements and the class diagram presented in Figure 2. They were given 2 hours to work independently and produce a separate spreadsheet using no tool support but the spreadsheet software itself (in our case, Microsoft Excel 2007). Afterwards, we compared each manually built spreadsheet with a reference spreadsheet developed by an expert based on the same requirements and class diagram presented in Figure 2.

To compare the spreadsheets, we have produced a list of errors based on the taxonomy proposed by Panko and Aurigemma [11]. The list contains 108 distinct error types classified into three categories (lapses, planning, and qualitative). Lapses are observed when users forget to define something, such as representing a given concept or formula. Planning errors are observed when users do something wrong, like writing a wrong formula. Qualitative errors are observed when users forget to validate fields or fix the range of cells that are used by formulas or references: these are not errors by themselves, but they might lead inexperienced users to introduce errors when their spreadsheets grow to include more rows than initially planned. These types of errors are observed in three different stages of a spreadsheet development process: (i) building the spreadsheet's structure, including worksheets, tables, and columns; (ii) creating associations between concepts, such as a column to link two concepts represented in different worksheets; (iii) adding formulas and business rules.

Table 1 reports the number of errors found on spreadsheets developed by our subjects, according to the selected taxonomy and phases of the development process. For each error found in a manually built spreadsheet, we counted one point for the related subject, annotating the type of error and the stage on which it was observed. In the end, the number of points calculated for a given subject accounted for the number of errors on his/her spreadsheet.

Table 1. Error count – Stage x Category

Category/Stage	Structure	Association	Formulas	Total
Lapses	10	7	34	51
Planning	14	8	65	87
Qualitative	-	121	93	214
Total	24	136	192	352

To answer RQ1, **Table 1** shows that most errors found in the spreadsheets developed by our subjects were observed in the formula development stage (55%). The most common errors include using a non-extensible cell range for formulas (i.e., allowing the user to create new entries in the spreadsheet that do not contain those formulas), using wrong ranges in formulas (e.g. referencing a column that does not represent the desired concept), and using wrong operators in formulas. The first and second errors are found when the user does not rely on tables to represent concepts in a spreadsheet. The first is due to fixing the range of cells queries by the formulas using the dollar sign, while the second is due to referring to cells using the alphanumeric notation (e.g., A2:B2) instead of table column's names. The last error is typically introduced by users who lack proper knowledge on formula components and spreadsheet features.

Table 2. Correlation between the number of errors and subject knowledge of spreadsheet development (self-evaluation).

	Correlation
Number of errors	-0.26
Structural errors	-0.16
Association errors	-0.01
Formula errors	-0.40
Lapses	-0.30
Planning errors	-0.39
Qualitative errors	-0.02

In order to answer RQ2, we correlated the subjects' experience with the number of errors introduced by them according to error type and development phase. **Table 2**

presents these correlations (Spearman's coefficient). As expected, all correlations are negative, denoting that the more experienced the subjects, the less errors they commit. On the other hand, correlations are very small (for structural and association errors) or mild (for formula errors, lapses, and planning errors), meaning that either our subjects were unaware of their real spreadsheet skills or even experienced users might introduce a large number of errors[4].

III. General Approach

This section presents our approach to generate spreadsheet structure and formulas from a conceptual model. It adapts the concepts of model based software generation to the context of spreadsheet development. The spreadsheet is generated by using metamodel transformations to map domain elements into their corresponding spreadsheet components, such as worksheets, tables, and formulas. The approach assumes the existence of a correct and complete UML/OCL model (the conceptual model) containing information about the concepts, properties, associations, and rules describing the domain. The limitations of such assumptions are discussed in Section VIII.

We have defined a set of transformation rules that map the elements comprising the UML model (classes, attributes, associations, and operations) into spreadsheet elements (worksheets, tables, and columns) in order to generate a spreadsheet model corresponding to the knowledge expressed in the conceptual model. Besides, since the semantics of query operations in the conceptual model is defined in OCL, a second set of rules allows transforming OCL expressions into spreadsheet formulas. Once the spreadsheet model has been generated, the approach provides for its transformation into an executable spreadsheet file for a specific spreadsheet software (MS Excel 2007, in our current implementation).

UML models and OCL expressions are instances of their respective metamodels. To represent transformation rules from UML/OCL models to spreadsheet elements in a metamodel level, we developed a spreadsheet metamodel. Former research work have already proposed spreadsheet metamodels, such as the one described in [6]. Our metamodel differs from the metamodel presented in [6] in two major aspects: (i) it handles tables, instead of being limited to data groups that extend either "vertically" and "horizontally", and (ii) it can produce more complex worksheet designs than the counterintuitive two-dimensional format used in the former work to represent relationships among classes.

The components of our spreadsheet metamodel are shown in Figure 1. The most important elements are *XWorkbook*, *XWorksheet* and *XDataTable*. *XWorkbook* represents an entire spreadsheet. An *XWorkbook* may contain one or more *XWorksheet* instances, each corresponding to a workbook tab. An *XWorksheet* can be connected to zero or more instances of *XDataTable*. An *XDataTable* is a data table represented in a worksheet. A data table is structured as a matrix, with several rows and columns. Each column, represented by an instance of *XDataTableColumn*, has a header that gives a name to the contents of its cells (the name is represented in the *header* attribute). The *isIdentifier* attribute of a column indicates whether the data in its cells can be used to uniquely identify a row in the table. A table can have one or more identifier columns. *XDataTableColumn* instances also specify the type of data represented in their cells (*XDataType*) which can be either a standard Excel type (number, string, date, among others) or an enumeration. Enumerations define a set of possible values (instances of *XDataElement*) which represent the domain of values that can be assumed by its associated cells. An *XDataTableColumn* may also contain a formula (instance of *XFormulaExp*).

Although the spreadsheet metamodel has some limitations (for instance, it does not include abstractions to represent formulas, which are described in their native, textual format) it was successfully used as counterpart for our transformation rules, which are presented in Section IV and Section V.

IV. Translating a Conceptual Model into a Worksheet Model

This section describes the rules proposed to transform a UML conceptual model, containing classes, attributes, operations and associations, into a spreadsheet model.

Rule 1: UML Model into Spreadsheet Workbook

- Input:
  - *model : UML::Model*
- Steps:
  - *Create instance workbook: Spreadsheet::XWorkbook | workbook.name = model.name;*
  - *Apply Rule 2 (model, workbook);*
  - *Apply Rule 3 (model, workbook);*
  - *Apply Rule 6 (model, workbook).*

Rule 1 states that an instance of the *XWorkbook* element defined in the Spreadsheet metamodel is created from an instance of the *Model* element defined in the UML metamodel. The instance of the *Model* element is the root of a UML class model, i.e., all classes, enumerations and associations in the conceptual model are descendant nodes of that root element in a tree-like representation of a UML model, such as XMI. Execution of rule 1 triggers the execution of rules 2, 3 and 6 described in this section.

Rule 2: UML Enumeration into Spreadsheet XDataType

- *Inputs:*
  - *model : UML::Model*
  - *workbook : Spreadsheet::XWorkbook*
- *Steps:*
  - *Let allEnumerations = model.ownedElements->select (e | e.oclIsType(Enumeration))*
  - *For each enum : UML::Enumeration in allEnumerations:*
    - *Create instance dataType : Spreadsheet::XDataType | dataType.name = enum.name;*
    - *For each aLiteral : Literal in enum.ownedLiteral:*
      - *Create instance elem : Spreadsheet::XDataElement | elem.value = aLiteral.name;*
      - *Link elem to dataType.*
    - *Link dataType to workbook.*

[4] Mild and small according to conventions proposed by Cohen [31].

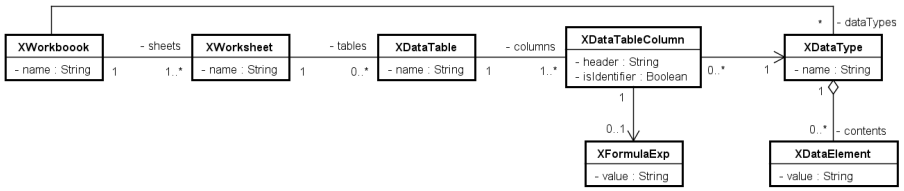

Figure 1. The spreadsheet metamodel

A UML enumeration is a classifier that defines the set of possible values that can be assumed by attributes of that type. An enumeration is transformed into an *XDataType* instance in the spreadsheet model linked to a set of instances of *XDataElement*, one for each enumeration literal defined as an attribute of the enumeration.

Rule 3: UML Classes into Spreadsheet Worksheets and Tables

- *Inputs:*
  - *model : UML::Model*
  - *workbook : Spreadsheet::XWorkbook*
- *Steps:*
  - *Let allClasses = model.ownedElements->select (e | e.oclIsType(Class))*
  - *For each aClass : UML::Class in allClasses:*
    - *Create instance worksheet : Spreadsheet::XWorksheet | worksheet.name = aClass.name;*
    - *Link worksheet to workbook;*
    - *Create instance dataTable : Spreadsheet::XDataTable | dataTable.name = aClass.name;*
    - *Link dataTable to worksheet;*
    - *Create instance dataType : Spreadsheet::XDataType | dataType.name = aClass.name;*
    - *Link dataType to workbook;*
    - *Apply Rule 4 (aClass, dataTable, workbook);*
    - *Apply Rule 5 (aClass, dataTable, workbook).*

This rule consists of generating a worksheet containing a data table in the spreadsheet model for each class in the UML conceptual model. Each instance of the *Class* UML metamodel element is transformed into an instance of *XWorksheet* linked to an instance of *XDataTable* in the spreadsheet model. Class attributes and operations are transformed according to rules 4 and 5.

Rule 4: UML Attributes into Spreadsheet Columns

- *Inputs:*
  - *class : UML::Class*
  - *dataTable: Spreadsheet::XDataTable*
  - *workbook : Spreadsheet::XWorkbook*
- *Steps:*
  - *For each attr : UML::Property in class.ownedAttributes:*
    - *Create instance column : Spreadsheet::XDataTableColumn | column.header = attr.name;*
    - *Link column to dataTable;*
    - *Let colType : Spreadsheet::XDataType = workbook.dataTypes -> select (t|t.name = attr.type.name) -> asSequence()->first();*
    - *Link column to colType;*
    - *If attr.ownedStereotypes->exists(s | s.name = 'Id') then set column.isIdentifier to true else set column.isIdentifier to false endif.*

Attributes of a UML class are represented as instances of the *Property* class defined in the UML metamodel. Each class has the property *ownedAttributes,* which contains all the attributes defined in that class. This rule states that for each attribute *attr* in *ownedAttributes*, a column is created in the data table corresponding to its class. An attribute associated to a stereotype named *Id* means that this attribute is either the identifier or part of the identifier of the corresponding class. This information is represented in the spreadsheet model by the attribute *isIdentifier* defined in *XDataTableColumn*.

Rule 5: UML Operations into Spreadsheet Columns

- *Inputs:*
  - *class : UML::Class*
  - *dataTable: Spreadsheet::XDataTable*
  - *workbook : Spreadsheet::XWorkbook*
- *Steps:*
  - *Let allQueryOperations = class.ownedOperation->select(op | op.isQuery);*
  - *For each oper : UML::Operation in allQueryOperations:*
    - *Create instance column : Spreadsheet::XDataTableColumn | column.name = oper.name;*
    - *Link column to dataTable;*
    - *Let colType : Spreadsheet::XDataType = workbook.dataTypes->select(t|t.name = oper.type.name)-> asSequence()->first();*
    - *Link column to colType.*

In the UML metamodel, the property *isQuery* present in the *Operation* metaclass defines whether the execution of an operation leaves the state of the system unchanged (*isQuery = true*) or whether side effects may occur (*isQuery=false*). Rule 5 should be applied only to side effects free operations, i.e. those having property *isQuery=true*. Each query operation of a class is transformed into a column in the data table corresponding to that class. The column type is defined according to the return type of the operation.

Rule 6: UML Associations into Spreadsheet Columns

- *Inputs:*
  - *model : UML::Model*
  - *workbook : Spreadsheet::XWorkbook*
- *Steps:*
  - *Let allClasses = model.ownedElements->select (e | e.oclIsType(Class))*
  - *For each aClass : UML::Class in allClasses:*

- *Let binAssoc : Set(Association) = aClass.association-> select(a | a.memberEnd-> size() = 2)*
- *Let manyToOneAssoc : Set(Association) = binAssoc-> select (a | a.memberEnd->forAll(e1,e2 | (e1.type <> e2.type and e1.type = aClass) implies (e1.upper = 1 and e2.upper > 1)))*
- *Let assocTypes:Set(Type) = manyToOneAssoc.memberEnd-> select(m| m.type <> aClass)*
- *For each type in assocTypes*
  - *Let dataTable = workbook.sheets.tables->select(table | table.name = type.name)*
  - *Create instance col : Spreadsheet::XDataTableColumn | col.name = aClass.name;*
  - *Link col to colType where colType : XDataType = workbook.dataTypes->select(t|t.name=aClass.name)-> asSequence()-> first()*
  - *Link column to dataTable.*

Finally, an association is represented by the *Association* element in the UML metamodel. An instance of *Association* has two or more *memberEnds*, which represent the classes connected by the association and their related multiplicity. Each *memberEnd* defines minimum (lower) and maximum cardinality (upper) for the number of instances of the classifier in the association end. By its turn, a *Classifier* element maintains a list called *association* keeping all associations on which it participates. We restrict our approach to many-to-one binary associations in which the target classifier corresponds to the one side of the association. This transformation adds a new column to each table corresponding to the many side of each many-to-one binary association.

V. Transforming OCL Expressions into Formulas

This section describes how OCL expressions defining the body of query operations are transformed into spreadsheet formulas. The description of the set of rules involved in that transformation uses the model shown in Figure 2 annotated with the OCL constraints listed in Figure 3.

A. Literal Expressions

A *LiteralExp* is an expression with no arguments that produces a value of a given type. The OCL metamodel defines a hierarchy of types starting from the *LiteralExp* superclass and its specializations: *BooleanLiteralExp*, *RealLiteralExp*, *IntegerLiteralExp* and *StringLiteralExp.* Instances of these types represent literal values and are directly translated into a textual representation in the formula. For example, in the OCL expression *quantity > 0 and quantity <= 10* (Figure 3, line 2), 0 and 10 are instances of *IntegerLiteralExp* and are translated into their textual representations when an instance of the *XFormulaExp* type is generated from that OCL expression.

B. Self Variable Expressions

In OCL, a *VariableExp* is a reference to a variable in an expression. Each variable has a type represented by a class in the underlying UML model. For example, the semantics of the *Purchase::total* operation (Figure 3, lines 3 and 4) is defined by the expression *unitPrice * quantity*, where *unitPrice* and *quantity* are abbreviations of the OCL property call expressions *self.unitPrice* and *self.quantity,* respectively. The transformation of a *VariableExp* instance involving the OCL implicit variable *self* generates a reference to the spreadsheet table associated to the classifier that defines the variable type. Since the expression *unitPrice * quantity* is defined in the context of an operation of the *Purchase* class, the type of *self* is *Purchase* and each reference to *self* in the expression is translated into a reference to the *Purchase* table.

C. Property Call Expressions

A *PropertyCallExp* is an expression that references an attribute of a *class* defined in the UML model and its evaluation results in the value of that attribute. A *PropertyCallExp* expression has the form <*source*>.<*attribute*>, where <*source*> is an instance of *VariableExp* and <*attribute*> is the name of an attribute of the class that defines the type of the <*source*> expression. Therefore, an OCL property call expression is translated into an expression of the form <*table*>[<*column*>] in the spreadsheet model, where <*table*> is the result of the transformation applied to the <*source*> node of the OCL expression and <*column*> is the column name associated to the <*attribute*>. The expression *unitPrice * quantity* (Figure 3, line 4) contains two property call expressions: *self.unitPrice* and *self.quantity.* The expression *self.unitPrice* is translated into *Purchase[unitPrice],* while the expression *self.quantity* is translated into *Purchase[quantity]* in the spreadsheet formula.

D. Basic Operation Call Expressions

An *OperationCallExp* is an expression that references a query operation defined in a class and its evaluation equals to the value resulting from the evaluation of the OCL body expression associated to that operation. An *OperationCallExp* has a source expression and may contain a list of argument expressions corresponding to the operation's arguments. Transformations applied to OCL operation call expressions vary according to the operation involved.

Arithmetic (+,-,*,/) and relational (>,<,<=,<=,=,<>) operation call expressions have three components: two operands defined by <*source*> and <*argument*> expressions, and an <*operation*> applied to these operands. OCL types *Integer* and *Real* contain four arithmetic operations, while the relational operations are defined in the *Boolean* OCL type. The transformation of an arithmetic or relational operation call expression generates an expression of the form <*exp1*><*operator*><*exp2*>, where <*exp1*> is the result of the transformation applied to <*source*>, <*operator*> is the symbol associated to the <*operation*>, and <*exp2*> is the result of the transformation applied to <*argument*>.

For example, in the expression *unitPrice * quantity* (Figure 3, line 4), *unitPrice* is the <*source*> expression (i.e., *self.unitPrice*) and is translated into *Purchase[unitPrice] (<exp1>)*; *quantity* (i.e*., self.quantity*) is the <*argument*> expression, which is translated into *Purchase[quantity] (<exp2>).* The * corresponds to the *times* operation defined in the *Integer* type, since the <*source*> expression results in an integer value. *Therefore*, * is the symbol associated to the <*operator*> in the resulting formula. In this example, the table column generated from the *Purchase::total* operation (Figure 3, line 3) is associated to an instance of *XFormulaExp* with the text "=*Purchase[unitPrice] * Purchase [quantity]"*.

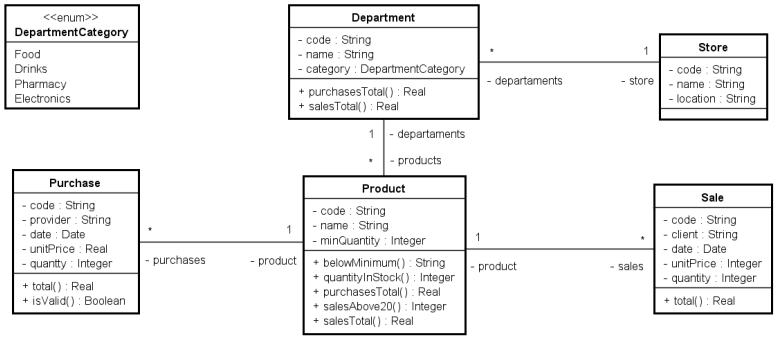

Figure 2. UML class diagram that supports the discussion on transformation rules

```
01  context Purchase::isValid() : Boolean
02    body: quantity > 0 and quantity <= 10
03  context Purchase::total() : Real
04    body: unitPrice * quantity
05  context Sale::total() : Real
06    body: unitPrice * quantity
07  context Product::belowMinimum() : String
08    body: if quantityInStock() < minQuantity then "Purchase" else "InStock" endif
09  context Product::quantityInStock() : Integer
10    body: purchases.quantity->sum() –  sales.quantity->sum()
11  context Product::salesAbove20() : Integer
12    body: sales->select(s | s.quantity > 20)->size()
13  context Product::purchasesTotal() : Real
14    body: purchases.quantity->sum()
15  context Product::salesTotal() : Real
16    body: sales.quantity->sum()
17  context Department::purchasesTotal() : Real
18    body: products.purchases.total()->sum()
19  context Department::salesTotal() : Real
20    body: products.sales.total()->sum()
```

Figure 3. OCL expressions associated to the UML model depicted in Figure 2

A boolean operation call expression has the form <*source*><*operation*><*argument*>, where <*source*> and <*argument*> are boolean expressions. The transformation of a boolean operation call expression generates an expression in the form <*func*>(<*exp1*>,<*exp2*>), where <*func*> is the spreadsheet function (AND or OR) corresponding to the Boolean <*operation*>, whilst <*exp1*> and <*exp2*> are transformed versions of <*source*> and <*argument*>.

The expression *quantity > 0 and quantity <= 10* (Figure 3, line 2) corresponds to the body expression of the *Purchase::isValid* operation and it is translated into the formula *AND(<exp1>, <exp2>)*, where *exp1* and *exp2* are the result of the transformation of the source expression *quantity > 0* and the argument expression *quantity <= 10,* respectively. Applying the transformations to literal expressions *(0, 10)*, relational operation call expressions *(>, <=)*, and property call expressions *(quantity)*, the table column generated from the *Purchase::isValid* operation would be associated to an *XFormulaExp* instance containing "*=AND(Purchase[quantity] > 0, Purchase[quantity] <= 10)*".

Finally, model operation call expressions represent calls to query operations defined in classes of the conceptual model. Such operation call expression is translated into a structured reference to the table column associated to the operation, similarly to the transformation applied to property call expressions. In the OCL body expression associated to the operation *Product::belowMinimum()* (Figure 3, lines 7 and 8) *quantityInStock()* is an abbreviation for *self.quantityInStock()*. It is translated into a reference to the column *quantityInStock* of the *Product* table defined in the spreadsheet model, i.e. "*Product[quantityInStock]*".

E. Conditional Expressions

A conditional expression results in one of two alternative expressions according to the value of a condition. A conditional expression has a condition, a *then* expression and an *else* expression. It is translated into the IF() spreadsheet function, which has three arguments: a condition to be evaluated, a formula associated to a positive (true) evaluation of the condition, and a formula associated to a negative (false) evaluation. The OCL expression defined in Figure 3, line 8, *if quantityInStock() < minQuantity then "Purchase" else "InStock" endif* is translated into an *IF(<exp1>,<exp2>,<exp3>)* spreadsheet function, where <*exp1*> corresponds to the translation of *quantityInStock() < minQuantity*, while <*exp2*> and <*exp3*> correspond to the translation of the string literal expressions *Purchase* and *InStock,* respectively. The table column generated from the *Purchase::belowMinimum* operation (Figure 3, line 7) would be linked to an instance of

XFormulaExp containing the text *IF (Product [quantityIn Stock] < Product [minQuantity], "Purchase", "InStock").*

F. Association Navigation Expressions

An association navigation expression references an association end of an association between classifiers. We have defined transformations for navigation expressions through binary associations where the target association end has multiplicity '*'. For instance, in the expression *purchases. quantity->sum()* (Figure 3, line 14), *purchases* corresponds to the collection of *Purchase* instances (target association end) linked to an instance of *Product* (source). In this work, association navigation expressions are used in aggregation formulas involving the OCL collection operations *size* and *sum*, described in section G.

Each association navigation expression is translated into a formula fragment of the form *A[B],C[D]* where: A is the table name corresponding to the target classifier (*Purchase*); B is the column in the target table that holds the key to identify the source instance for the navigation (*Product*); C is the table name corresponding to the source classifier (*Product*); D is the column that holds the source's identifier (*Code*). Applying this transformation to the *purchases* navigation expression *(i.e., self.purchases)* present in *purchases.quantity->sum(),* results in the fragment *Purchase[Product], Product[Code]*.

In Figure 3, line 20, the expression *products.sales.total()->sum()* is associated to the *Department::salesTotal* operation. This operation returns a real number corresponding to the sum of the total price of all sales for all products in a department. This expression contains a two-level navigation in the form *source-middle-target*, since it traverses two associations (*Department-Product* and *Product-Sale*). The transformation of a two-level navigation requires two steps.

The first step creates a new column in the table associated to the classifier corresponding to the *target* association end in the traversal (e.g., *Sale*). This column has the name of the *source* class in the traversal. The formula associated to this column is generated using the spreadsheet functions INDEX and MATCH to collect the source class (e.g. *Department*) *id* from the table corresponding to the intermediate class (e.g. *Product*) which is linked to the target element (e.g. *Sale*). The generated formula has the form:

```
=INDEX(<middle>, MATCH([<middle>],middle[lastId],0),COL(middle[source]))
```

The INDEX function returns the value stored in a certain cell of a table (first argument), identified by its row (second argument) and column (third argument). The MATCH function returns the position of a value in a list. It also has three arguments: the value sought, the range of cells to be searched, and a flag indicating the matching type (0 for exact match). Finally, the COL function retrieves the number of a given column in a table. The transformation of the navigation *self.products.sales* listed in Figure 3, line 20, would result in a new column defined in the table corresponding to the *Sale* class. This column would be associated to an instance of *XFormulaExp* having the value:

```
=INDEX(Product, MATCH([Product], Product[Code], 0),
COL(Product[Department]))
```

The formula above queries the *Product* table to return the department code of the product having its code value stored in the *[Product]* column of the *Sale* table. The second step creates the *A[B],C[D]* formula fragment described earlier. *A[B]* refers to the column created in the first step (navigation source), while *C[D]* refers to the target of the navigation. In the example, the fragment formula would be *Sale[SalesTotal]*, *Sale[Department]*. It is used as part of a collection operation call that will be described in the next section.

G. Collection-related Operation Call Expressions

OCL defines several collection operations. We have defined transformations for two of these operations: *sum()* and *size()*. The *sum* operation is translated into the *SUMIFS* spreadsheet function. This function expects three arguments: a range of spreadsheet cells to be summed, a range of cells to be evaluated against a condition that defines which elements should take part in the sum, and the condition itself. The first argument corresponds to the source expression associated to the *sum()* operation, usually a collection or an OCL iterator expression. The second and third arguments filter the elements resulting from the evaluation of the first argument to select those that must be summed up.

The OCL expression *purchases.quantity->sum()* (Figure 3, line 14) defines the body of the *Product::purchasesTotal* operation as the sum of the quantity of each purchase associated to a product. The source expression is *purchases.quantity*, which is an abbreviation for the iterator expression p*urchases->collect(p | p.quantity).* An iterator expression traverses a collection and selects a subset of its elements. This expression is translated into a call to the element that should be collected, either an attribute or an operation. The iterator expression *purchases->collect(p | p.quantity)* collects the value of the *quantity* attribute from each entry of the *purchases* collection. Thus, applying the *PropertyCallExp* transformation, the resulting formula fragment for the iterator expression is *Purchase[quantity]*, denoting that quantities will be summed in the expression.

The second argument indicates that the *Product* column in the *Purchase* table is used to select which entries from *Purchase[quantity]* should be collected and summed. Finally, the last argument refers to an equality condition that indicates that only purchases from a given product (identified by its code) should be summed. The translation of the expression leads to the following formula associated to the column corresponding to the *Product::purchasesTotal* operation:

```
=SUMIFS(Purchase[Quantity],Purchase[Product],Product[Code])
```

The *size* operation is translated into the *COUNTIFS* spreadsheet function, which receives similar arguments to the SUMIFS function. Consider the OCL expression *sales->select(s | s.quantity > 20)->size()* (Figure 3, line 12). It returns the number of products associated to sales where the quantity sold is greater than 20 units. The first two arguments of the COUNTIFS function correspond to the transformation of the association navigation expression *self.sales*, which generates the formula fragment *Sale[Product], Product[Code]*. The condition in the *select* expression is translated into the last argument of the COUNTIFS function. The text "*Sale[Quantity]>20"* corresponds to the translation of

"*s.quantity*" (property call expression) and "*> 20*" (relational operation call and literal expressions). Thus, the produced formula is:

```
=COUNTIFS(Sale[Product],Product[Code], Sale[Quantity] > 20)
```

VI. Evaluation

All transformations presented in this work have been implemented in C#. The input is a UML model annotated with OCL expressions defined in Visual Studio 2010 and the implementation generates a Microsoft Excel spreadsheet. To evaluate whether the generated spreadsheet would be robust enough to inhibit users in introducing errors while new data is fed into its worksheets, we designed and executed a human-based experimental study.

The proposed study involved six IT professionals. Four of these subjects were undergraduate students and two had completed their courses on Computer Science. Three subjects worked on the IT sector for less than 5 years, while the remaining three had more than 10 years of work experience. In addition, four subjects have basic experience in building spreadsheets with simple aggregation and selection features, while two subjects were experienced spreadsheet developers who know how to deal with complex features, such as table look up formulas and data validation.

Subjects were asked to perform a set of activities related to the same domain underlying the spreadsheet developed as part of the experiment reported in Section II[5]. Those activities were designed to expose subjects to situations on which the errors found in the first experiments might be committed. The proposed activities were:

A1: to add a new product to a department in a given store. This activity was used to evaluate if the calculated columns that exist for every product would be created for the new product, regardless of which means the user might select to include this product;

A2: to add a new line in the end of the list of sold products, to represent that a new product was sold. Besides verifying whether formulas related to sales are copied, this activity was designed to determine whether the user might be able to create an invalid association between sales and products or create a sale without a product;

A3: to add a new line in the start of the list of bought products, to represent that a new product was bought from a supplier. As with the former activity, A3 was designed to determine whether formulas and associations would also be consistent if the user add the new information before the first line that registers data about products being bought;

A4: to change the department that sells a given product to an inexistent department. This activity was designed to determine whether a user might be able to block the data validation mechanisms generated by the proposed approach on regard of associations between concepts handled by the spreadsheet;

A5: to eliminate a product that has already been sold or bought before. This activity was designed to determine whether a user might be able to remove information about a concept that is referred to in other parts of the spreadsheet.

After performing the five proposed activities, each user filled a questionnaire containing 39 questions. The questions referred to specific cells of the worksheet and required the subject to copy the values presented on these cells. One of the authors filled the same questionnaire with the correct answers and these answers were used as a baseline for comparison with each subject's answers. Wrong answers meant that either the subject introduced an error while performing the proposed activities or he/she misinterpreted the information required by the related question.

Four out of six subjects (66%) correctly answered all 39 questions. One subject presented an incomplete answer for the first question, which requested the number and names of the worksheets comprising the spreadsheet after the activities were performed: the subject presented only the number of worksheets without providing their names. All 38 remaining questions were correctly answered by this subject. The last subject answered 35 out of 39 questions correctly. For the remaining 4 questions, the subject forgot to copy the sign of the number presented in the related spreadsheet cell to the hard-copy questionnaire.

Despite of having 5 incorrect answers out of 195 questions, we observe that errors most frequently committed by users were inhibited by the spreadsheet generated through our proposed approach. Nevertheless, this experimental study is limited due to the small number of subjects involved (only six practitioners) and its limited diversity in terms of spreadsheet and domain used in the evaluation (single spreadsheet for a single domain). These limitations are threats to the validity of the experimental study, particularly to our ability to generalize its results (external validation). Thus, we have initial evidence that our proposed approach might help these users to build spreadsheets that are more robust when a conceptual model for the domain of their interest is available. Replication of the proposed study with a large number of users and by using different spreadsheets generated from distinct conceptual models is required for further generalization.

VII. Related Works

Several spreadsheet related studies have been reported in the technical literature. Most of these studies focus on reducing the number of spreadsheet errors, although recent work addresses different ways to visualize the spreadsheet's structure and rules to support understanding and evolution. Jannach et al. [20] present a recent survey of automated approaches for spreadsheet quality assurance. The authors classify the selected papers into two categories: tools and techniques designed to help the user to detect and fix errors, and tools and techniques to help developers in creating spreadsheets that do not have errors in the first place.

[5] The spreadsheet is available at http://www.uniriotec.br/~marcio.barros/sbes2015/mdespreadsheets.zip

In line with the first category [20], some papers propose approaches to analyze spreadsheets in order to increase understanding, support audits, identify and correct errors. Rothermel et al. [12] [13] [14] presented a spreadsheet testing method based on a test case generation procedure that uses formula representations and cell association graphs. Panko and Halverson [21] proposed a taxonomy to categorize different kinds of spreadsheet errors. This taxonomy was later extended by Panko and Aurigemma [11]. The new version uses concepts of more recent studies in human error research and includes error types that were not included in the original taxonomy. For example, the distinction between developer and end-user accidental errors, like inputting incorrect data or overwriting a formula with a number. Cunha et al. [9] [10] presented a method to identify inadequate constructions in spreadsheets ("*bad smells*"), like empty cells, references to empty cells, and constant values in formulas. Ronen et al. [7] proposed a spreadsheet development lifecycle comprised of four activities: problem identification, construction, testing, and documenting. The authors also proposed a standard format to structure spreadsheet information and a diagram named *Spreadsheet Flow Diagram* (SFD) to specify formulas. Later, Davis [15] proposed a tool called "*Online Data Dependency Tool*" that automatically extracts SFD from spreadsheets. Clermont [16] introduced *Data Dependency Graph* to show relations amongst spreadsheet cells to users. Hermans et al. [17] extended Clermont's work to show the breakdown for the calculation implied in a given formula. The authors propose an approach that shows multilevel data flow diagrams under three different views: workbooks, worksheets, and cells. Higher-level diagrams are used to abstract the details from lower level ones. For instance, a worksheet data flow represents dependencies among worksheets without showing which cells from these worksheets depend on each other. Clermont et al. [8] proposed a tool to support spreadsheet auditing through the visualization of its structure and according to distinct types of relations found amongst cells. Hermans et al. [19] proposed a transformation-based, reverse-engineering approach to build class diagrams from the structure of a spreadsheet to support users in understanding its contents and calculations.

Our work does not identify nor correct spreadsheet errors. In line with the second category proposed by Jannach et al [20], it focus on avoiding spreadsheet errors by building correct spreadsheet structures and formulas from UML class diagram models made precise by means of OCL constraints. Our major contributions relates to formula generation, given that previous research presented below have already explored ways to build spreadsheet structures from diagrams representing high-level concepts. Rajalingham et al. [3] showed a method to build spreadsheets based on techniques, recommendations, and rules adapted from Software Engineering. The EuSpRIG group [4] highlighted the need for modeling and suggested the use of influence graphs to represent formulas based on their inputs and outputs before building a spreadsheet. The seminal work from Engels and Erwig [6] introduced ClassSheets, a model to represent spreadsheet structure and rules. Succeeding research works, such as Belo et al. [22] and Cunha et al. [18][23][24][25][27], used this model to allow users to understand, maintain, and evolve spreadsheets by analyzing their models instead of a direct examination of spreadsheet data. Francis et al. [26] proposed an approach to feed spreadsheets as models in a MDE toolkit. The authors query and modify the spreadsheets containing the models under interest using an OCL-based language (*Epsilon Object Language*). Instead of using OCL to generate spreadsheets, they use it to represent the information conveyed on the spreadsheet so that it can be used as a first-class model to support other kinds of transformations.

VIII. APPLICABILITY OF THE PROPOSED APPROACH

One major limitation of the proposed approach is that it requires a rich conceptual model, including OCL expressions for the calculated attributes, to be available for the domain for which the spreadsheet must be developed. Certainly, it should not be expected that end-users would develop these models before building their spreadsheets. To build such models, one must have extensive knowledge on the UML metamodel and OCL expressions, and such knowledge might be rare even on a group of professional software developers.

However, UML models and OCL are one of the ways on which conceptual models can be expressed. Ontology languages and notations, such as OWL, represent alternative formal representations that can be used to build such models. Indeed, end-users should also not be expected to develop ontology models, but the trend of semantic web has been producing and using such models for some years. It might be expected that, in the future, many ontologies are extensively documented and available for general use, including the generation of spreadsheets. Taking software development as an example for the domain of interest, we already have several software-related ontologies available [28][29][30].

Therefore, while the proposed approach was presented with UML and OCL expressions in the background, the proposed transformations can be easily adapted to other formalisms that might be available, leveraging the potential application of the proposed approach.

One might also consider the usefulness of the proposed approach in the light of other tools, such as Microsoft Access, which already deal with existent structured data. From our experience, we observe that non-IT users tend to use spreadsheets more frequently than solutions such as Microsoft Access. The latter is a useful model, particularly when information is well structured and the questions to be answered are known beforehand. Spreadsheets are used not only to answer previously stated questions (our approach may apply to this case), but also to save data still to be organized and to visualize such data on several perspectives. Thus, although a similar model can be applied to Microsoft Access or even to general software development, it is our belief that spreadsheets will remain useful in the long run.

IX. CONCLUSIONS

This paper presented an approach to generate spreadsheets from UML/OCL conceptual models. The approach creates spreadsheets whose structure prevents the user from introducing certain kinds of errors. Transformations based on the UML/OCL metamodels and a spreadsheet metamodel are

applied to produce a target spreadsheet from the conceptual model. We have executed an experimental study that provided initial evidence that spreadsheets built from the proposed approach are resilient to the introduction of errors which have been formerly identified as recurrent in spreadsheet development.

Acknowledgment

The authors would like to thank the 26 subjects that have participated in the proposed experimental studies for the time and effort they invested in our research.

References

[1] PANKO, R.R.; HALVERSON JR., R. P. (2001) "An Experiment in Collaborative Spreadsheet Development". Journal of the Association for Information Systems 2(4), July.

[2] POWELL, S.; BAKER, K. (2003) "The Art of Modeling with Spreadsheets". John Wiley & Sons, Inc. New York, NY, USA.

[3] RAJALINGHAM, K.; CHADWICK, D.; KNIGHT, B; EDWARDS, D. (2000) "Quality Control in Spreadsheets", IN: Proc. of the 33rd Hawaii International Conference on System Sciences, page 10.

[4] EuSpRIG. European Spreadsheet Risks Interest Group. http://www.eusprig.org/, 2011. pp. 3, 5 and 133.

[5] FRANCE, R.; RUMPE, B. (2007) "Model-driven Development of Complex Software: A Research Roadmap", In 2007 Future of Software Engineering (FOSE '07). IEEE Computer Society, Washington, DC, USA, 37-54, 2007.

[6] ENGELS G.; ERWIG, M. (2005) "ClassSheets: Automatic generation of spreadsheet applications from object-oriented specifications", IN: Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering, pages 124-133, USA.

[7] RONEN, B.; PALLEY, M.; LUCAS JR., HENRY. (1989) "Spreadsheet analysis and design", Communications of the ACM, 32(1):84–93, January 1989.

[8] CLERMONT, M.; HANIN, C.; MITTERMEIR, R. T. (2008) "A Spreadsheet Auditing Tool Evaluated in an Industrial Context", IN: ACM Computing Research Repository (CoRR).

[9] CUNHA, J.; FERNANDES, J. P., MARTINS,P.; MENDES, J.; SARAIVA, S. (2012) "SmellSheet Detective: A Tool for Detecting Bad Smells in Spreadsheets", IN: Proc. of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2012), Austria, pages 243–244, 2012.

[10] CUNHA, J.; FERNANDES, J.P.; RIBEIRO, H.; SARAIVA, J. (2012) "Towards a catalog of spreadsheet smells", IN: Proc. of the 12th International Conference on Computational Science and Its Applications (ICCSA'12). Vol. IV, pp. 202-216.

[11] PANKO, R. R.; AURIGEMMA, S. (2010) "Revising the Panko-Halverson taxonomy of spreadsheet errors". Decision Support Systems, 49(2):235–244.

[12] ROTHERMEL, G.; LI, L.; DUPUIS, C.; BURNETT, M. (1998) "What you see is what you test: a methodology for testing form-based visual programs", Proceedings of the 1998 International Conference on Software Engineering, 1998, vol., no., pp.198,207, 19-25 April.

[13] ROTHERMEL, K. J.; COOK, C. R.; BURNETT, M. M.; SCHONFELD, J.; GREEN,T. R. G.; ROTHERMEL, G. (2000) "WYSIWYT testing in the spreadsheet paradigm: an empirical evaluation". IN: Proc. of the 22nd international conference on Software engineering (ICSE '00). ACM, NY, USA, pp. 230-239.

[14] ROTHERMEL, G.; BURNETT, M.; LI, L.; DUPUIS, C.; SHERETOV, A (2001) "A methodology for testing spreadsheets", ACM Transactions Software Engineering Methodology, Vol. 10, Issue 1, pp. 110-147.

[15] DAVIS, J. S. (1996) "Tools for spreadsheet auditing". International Journal of Human Computer Studies, Vol. 45, Issue 4, pp. 429–442.

[16] CLERMONT, M. (2004) "A Scalable Approach to Spreadsheet Visualization". PhD thesis, Universitaet Klagenfurt

[17] HERMANS, F.; PINZGER, M; VAN DEURSEN, A. (2011) "Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams", IN: Proc. of the International Conference on Software Engineering (ICSE'11), pp. 451-460.

[18] CUNHA, J., SARAIVA, J., VISSER, J. (2009) "From spreadsheets to relational databases and back", IN: Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, pp. 179–188, NY, USA.

[19] HERMANS, F.; PINZGER, M.; VAN DEURSEN, A. (2010) "Automatically extracting class diagrams from spreadsheets", IN: Proceedings of the European Conference on Object Oriented Programming (ECOOP), pages 52-75.

[20] JANNACH, D.; SCHMITZ, T.; HOFER, B.; WOTAWA, F. (2014) "Avoiding, finding and fixing spreadsheet errors – A survey of automated approaches for spreadsheet QA", Journal of Systems and Software, Volume 94, Pages 129-150, August.

[21] PANKO, R.R.; HALVERSON JR., R. P. (2001) "An Experiment in Collaborative Spreadsheet Development". Journal of the Association for Information Systems 2(4), July

[22] BELO, O.; CUNHA, J.; FERNANDES, J. P.; MENDES, J.; PEREIRA, R.; SARAIVA, S. (2013) "QuerySheet: A Bidirectional Query Environment for Model-Driven Spreadsheets", In the Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13), September 15-19. San Jose, CA, USA.

[23] CUNHA, J.; FERNANDES, J. P.; SARAIVA, S. (2012a) "From Relational ClassSheets to UML+OCL", 27th Annual ACM Symposium on Applied Computing (SAC 2012), track on Software Engineering, Trento, Italy, pages 1151-1158, March.

[24] CUNHA, J.; FERNANDES, J. P.; MENDES, J.; SARAIVA, S.; PACHECO, H. (2012b) "Bidirectional Transformation of Model-Driven Spreadsheets", 5th International Conference on Model Transformation (ICMT 2012), Prague, Czech Republic, pages 105-120, May.

[25] CUNHA, J.; FERNANDES, J. P.; MENDES, J.; SARAIVA, S. (2012c) "MDSheet: A Framework for Model-driven Spreadsheet Engineering", 34th International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland, pages 1395-1398, June.

[26] FRANCIS, M.; KOLOVOS, D. S.; MATRAGKAS, N. D.; PAIGE, R. F. (2013) "Adding Spreadsheets to the MDE Toolkit", MoDELS, p. 35-51

[27] CUNHA, J.; MENDES, J., FERNANDES, J. P.; SARAIVA, S. (2011) "Embedding and Evolution of Spreadsheet Models in Spreadsheet Systems", In proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2011), Pittsburgh, USA, pages 179-186, IEEE Computer Society, September.

[28] KITCHENHAM, B.; TRAVASSOS, G.; MAYAUSER, A.; NIESSINK, F.; SCHNEIDEWIND, N.; SINGER, J.; TAKADA, S.; VEHLAINEN, R.; YANG, H. "Towards an ontology of software maintenance", Journal of System Maintenance, Vol. 11, No. 6, pp. 365-389, 1999

[29] ALIAS, M.; MIRIAM, D.; ROBIN, C.; "An ontology-based domain model to enhance the software development process", Intl Journal of Metadata, Semantics and Ontologies, Vol. 9, Issue 3, pp. 204-214, 2014

[30] FALBO, R.; BERTOLLO, G.; "A software process ontology as common vocabulary about software processes", Int. Journal of Business Process Integration and Management, Vol. 4, Issue, 4, pp. 239-250, 2009

[31] COHEN, J.; "A Power Primer", Quantitative Methods in Psychology, Vol. 112, Issue 1, pp. 155-159, 1992.