ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications

Gregor Engels*
University of Paderborn
33095 Paderborn, Germany
engels@upb.de

Martin Erwig†
Oregon State University
Corvallis, OR 97331, USA
erwig@eecs.oregonstate.edu

ABSTRACT

Spreadsheets are widely used in all kinds of business applications. Numerous studies have shown that they contain many errors that sometimes have dramatic impacts. One reason for this situation is the low-level, cell-oriented development process of spreadsheets.

We improve this process by introducing and formalizing a higher-level object-oriented model termed ClassSheet. While still following the tabular look-and-feel of spreadsheets, ClassSheets allow the developer to express explicitly business object structures within a spreadsheet, which is achieved by integrating concepts from the UML (Unified Modeling Language). A stepwise automatic transformation process generates a spreadsheet application that is consistent with the ClassSheet model. Thus, by deploying the formal underpinning of ClassSheets, a large variety of errors can be prevented that occur in many existing spreadsheet applications today.

The presented ClassSheet approach links spreadsheet applications to the object-oriented modeling world and advocates an automatic model-driven development process for spreadsheet applications of high quality.

Categories and subject descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques— *object-oriented design methods*; H.4.1 [Information Systems Applications]: Office Automation—*spreadsheets*

General terms: Languages, Reliability

Key words: Spreadsheet, UML, End-user software engineering

*Work of Gregor Engels performed during a sabbatical stay at Oregon State University. His work is partially supported by the Heinrich Hertz Stiftung.

†Martin Erwig is partially supported by the National Science Foundation under the grant ITR/AP-0121542 and by the EUSES consortium (`EUSESconsortium.org`).

1. INTRODUCTION

Spreadsheets are the most popular programming systems used today. They are particularly popular in the domain of business applications. Estimates say that tens of millions of business people create hundreds of millions of spreadsheets every year [20]. This successful and wide-spread use of spreadsheets is due to several reasons. In addition to the highly intuitive, two-dimensional tabular layout combined with convenient operations for row and column insertions and deletions, it is the fact that all kind of data aggregations, which are ubiquitous in business applications, are directly expressible in spreadsheets.

On the other hand, numerous studies have demonstrated that existing spreadsheets contain errors at an alarmingly high rate [21, 15]. This situation is even more worrying as spreadsheets are often used in critical planning and control systems within highly sensitive business domains, that is, errors in spreadsheets may have a direct and significant economic impact. In addition, due to pressures of compliance with the Sarbanes-Oxley Act [2], US business managers are longingly waiting for good support to reduce information and decision risks while deploying spreadsheets.

Spreadsheet user communities as well as spreadsheet tool vendors have been working for years on improving this situation by developing methodological guidelines, application-specific sample spreadsheets (often called templates), as well as numerous plug-ins and add-ons for commercial spreadsheet tools to build, document, visualize, maintain, analyze, and test spreadsheet applications (see, for example, [15] for links to many of these efforts). However, a closer look at all these initiatives reveals that accepted and well-known software engineering principles, which have been successfully applied in large-scale professional software development projects, are in general ignored during the creation of spreadsheets. Surprisingly scientific research results on spreadsheet *design* can rarely be found at relevant software engineering conferences or in journals (see Section 7 for a survey on related work). Nearly all mentioned efforts to improve the development of spreadsheets stick to improvements on the basis of the underlying low-level cell-oriented programming model. What is still missing is a thorough development process support for spreadsheet designers and users to bridge the immanently existing semantic gap between concrete problem-domain-specific requirements and their realization in a spreadsheet.

The need to support *end user software development* has been recognized for some time now [8] and has led to several international research initiatives. Two of them, in which the authors are participating, are the NSF-funded EUSES project [14] and the European Network of Excellence EUD-Net [1] on End-User Development. Both initiatives are addressing the same research question of understanding whether and how it is possible to bring the benefits of rigorous software engineering methodologies to end users.

Two prominent research and development trends within software engineering during the last decade have been the meanwhile overall accepted *object-oriented* approach toward software development [10], as well as a *model-driven engineering* (MDE) process [17]. We build our approach on these two principles and propose to lift the development of spreadsheet applications to the level of a spreadsheet model. In particular, we introduce means to cluster cells within a spreadsheet according to underlying problem domain-related business object structures, which will reduce the semantic distance between a problem domain and a spreadsheet application and will support the spreadsheet designer in detecting erroneous design decisions as early as possible. In pursuing this goal we are always adhering to the well accepted spatial spreadsheet metaphor that has made spreadsheets so popular. Based on this approach, we provide an automatic, tool-supported transformation process from spreadsheet models to concrete model-compatible spreadsheet applications. In this sense, the approach presented here will be an MDE instance of a fully automated development process starting with a high-level, user-defined model, which is automatically transformed into an executable program.

In previous work, we took a first step toward introducing an additional model level for spreadsheets by describing the structure and evolution of spreadsheets with the help of an abstraction called *template*.[1] Spreadsheet templates can be defined in this approach through an editor that is based on a formally defined visual language ViTSL (an acronym for **vi**sual **t**emplate **s**pecification **l**anguage) [5]. These spreadsheet templates are used as input for an automatic generation tool Gencel, which produces a MS Excel spreadsheet together with customized update operations that ensure that the spreadsheet stays within the model defined by the template [11].

While this approach reveals substantial benefits for all spreadsheet users, it is still limited in its scope since it focuses on structuring means that are geared toward the tabular row and column structure of spreadsheets. What is missing, are higher-level structuring means which allow to express constraints which are related to the underlying business logic.

Thus, the overall objectives of our approach are as follows

- introduce the concept of a high-level spreadsheet model
- introduce higher-level modeling means which are oriented toward the object-based structure of business applications
- keep the tabular two-dimensional layout of spreadsheet design in order to ensure user acceptance
- embed spreadsheet development into an object-oriented model-driven process
- provide a formalization for the transformation process as a base for its automation

[1]Not to be confused with the templates available in MS Excel that represent predesigned spreadsheet samples.

- provide tool support for a definition of spreadsheet models and their automatic transformation into a customized spreadsheet editor

The contributions of this paper and the embedding into a larger context are illustrated in Figure 1. The ClassSheet spreadsheet model will be used as a high-level, object-oriented model at the top of an automatic transformation chain. An informal introduction into ClassSheets will be given in Section 3. ClassSheets are automatically transformed into ViTSL templates, which are summarized in Section 2. In Sections 4 through 6 we formalize our approach. In Section 4 we define the abstract syntax of ClassSheets, which is extended in Section 5 by introducing typing rules for ensuring a two-dimensional tiling structure to characterize well-formed ClassSheets. In Section 6, the transformation of ClassSheets into ViTSL templates is formalized, which facilitates the reuse of the existing Gencel spreadsheet generator to complete the overall process in the transformation chain on the right-hand side of Figure 1.

The transformation on the left of Figure 1 shows further potentials of our approach, which are not explored in this paper. Since ClassSheets can be viewed as 2-dimensional extensions of UML class diagrams, in which derived object attributes are defined in a visual way, ClassSheets bear the potential to be used as an intuitive alternative to, for example, textual OCL expressions [3] to express data aggregations within an UML-based software development process for business applications. In this paper, we restrict to informally illustrate how ClassSheets are directly projected into standard UML class diagrams (see Section 3).

In Section 7 we discuss related work. Conclusions and future work presented in Section 8 complete the paper.

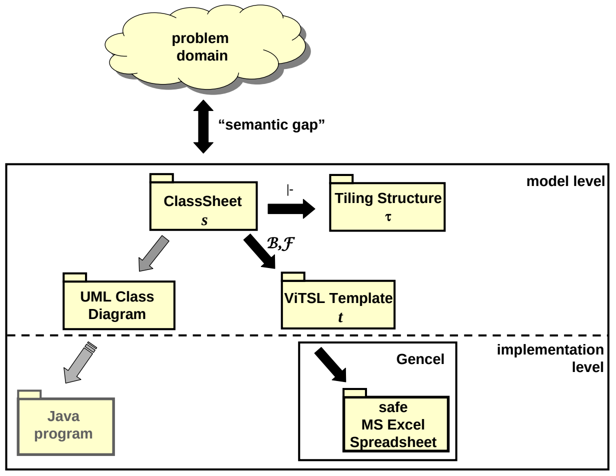
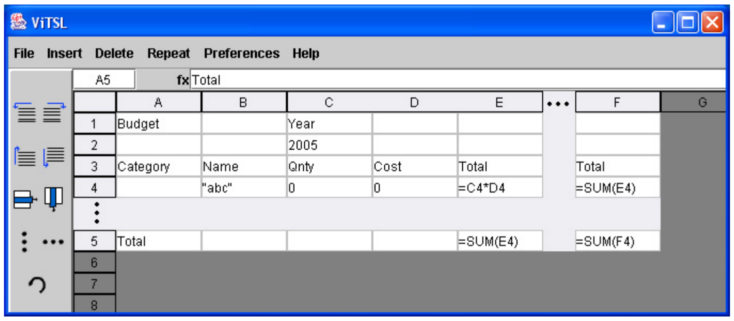


Figure 1: The ClassSheet approach.

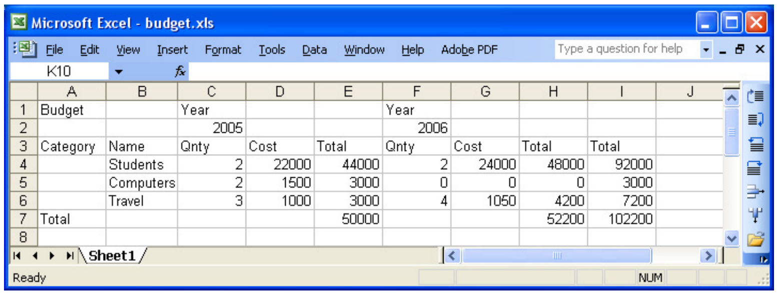
2. SPREADSHEET TEMPLATES

One approach to introduce a modeling step for spreadsheets is based on the observation that spreadsheets can be structured into vertically and horizontally composable, possibly recurring, blocks. This structured view forms the basis of the language ViTSL which allows the definition of spreadsheet templates [5].



	A	B	C	D	E	⋯	F	G
1	Budget		Year					
2			2005					
3	Category	Name	Qnty	Cost	Total		Total	
4		"abc"	0	0	=C4*D4		=SUM(E4)	
⋮								
5	Total				=SUM(E4)		=SUM(F4)	
6								
7								
8								

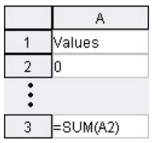
Figure 2: Budget template created with the ViTSL editor.


Microsoft Excel - budget.xls
File Edit View Insert Format Tools Data Window Help Adobe PDF
Type a question for help
K10
fx
Sheet1
Ready
NUM

	A	B	C	D	E	F	G	H	I	J
1	Budget		Year			Year				
2			2005				2006			
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total	Total	
4		Students	2	22000	44000	2	24000	48000	92000	
5		Computers	2	1500	3000	0	0	0	3000	
6		Travel	3	1000	3000	4	1050	4200	7200	
7	Total				50000			52200	102200	
8										

Figure 3: Automatically generated Gencel spreadsheet application.

A very simple example of a spreadsheet template is shown on the right. It describes the common structure of a one-column-sum spreadsheet including a header, in this case called Values. Such a template describes the following aspects:



	A
1	Values
2	0
⋮	
3	=SUM(A2)

1. Structural information, such as the existence of a header, of the one cell containing a SUM formula and of a number of “data” cells
2. Layout information, such as that the header is placed at the top and that the data cells are placed above the SUM cell
3. Data dependencies, such as that the data cells are used as the arguments of the SUM formula

Altogether, ViTSL offers the following visual elements for describing templates:

- *Cells*, represented by rectangles and containing data (representing labels and values) or formulas.
- *References*, represented by concrete cell addresses.
- *Vex groups*, represented by vertical dots that indicate the possible expansion of a (group of) cell(s) in vertical direction.
- *Hex groups*, represented by horizontal dots that indicate the possible expansion of one or more columns in horizontal direction (see Figure 2 for an example).

An important feature of ViTSL is that it is a visual language with a look-and-feel very similar to standard spreadsheets. This ensures a high acceptance of ViTSL by traditional speadsheet users. The ViTSL syntax has been formally defined in [5], too. Since our formalization will be based on the abstract ViTSL syntax, we briefly summarize the abstract syntax of templates (see Figure 4): A *template* (t) is given by a horizontal composition ($|$) of fixed (c) or expandable ($c^{\rightarrow}$) columns, where a column is given by a vertical composition ($\hat{}$) of fixed (b) or expandable ($b^{\downarrow}$) blocks. A block is given by a horizontal or vertical composition of blocks or is a formula (f). Formulas consist of basic values (φ), references to other data cells (ρ), and expressions that are built by applying functions to a varying number of arguments given by formulas ($\varphi(f,\ldots,f)$). References are given by relative offsets to cells, that is, by pairs of integers. In the concrete ViTSL syntax these offsets are represented by cell addresses, which is possible since the templates are surrounded by a global row/column frame. Since the global row/column numbers are not available in the abstract syntax, references are given by relative offsets. Therefore, references, such as C4 and F4, in Figure 2 are represented by pairs of integers $(-2,0)$ and $(0,-1)$, respectively.

$$\begin{array}{llll} f \in \mathit{Fml} & ::= & \varphi \mid \rho \mid \varphi(f,\ldots,f) & (\mathit{formulas}) \\ b \in \mathit{Block} & ::= & f \mid b \,|\, b \mid b\,\hat{}\,b & (\mathit{blocks}) \\ c \in \mathit{Col} & ::= & b \mid b^{\downarrow} \mid c\,\hat{}\,c & (\mathit{columns}) \\ t \in \mathit{Template} & ::= & c \mid c^{\rightarrow} \mid t \,|\, t & (\mathit{templates}) \end{array}$$

Figure 4: Template syntax.

In addition to the abstract, context-free syntax definition of ViTSL given by the BNF, a type system (representing context-sensitive constraints) has been defined that enforces the correct alignment of rows and columns with respect to repeating groups. The type system also guarantees correct references within a table and aggregation formulas as well as a correct typing in case of formula expressions. In particular, the alignment constraints ensure that insert/delete row/column operations are always well defined in the generated spreadsheet application.

The generated spreadsheet application consists of an MS Excel spreadsheet and tailor-made update operations for inserting rows, changing values, etc. These update operations are provided through a tool, called Gencel, which is an extension to MS Excel that restricts the definition and possible evolution of a concrete spreadsheet according to the specification given by the template. Thus, illegal update operations like a partial copying or moving of vertical (vex) or horizontal (hex) recurring groups are prevented. Furthermore, formulas are updated correctly and automatically, for example, in the case of inserting new instances of vex or hex groups in a concrete generated spreadsheet. For details on the type system and spreadsheet generation, see [11, 12].

Figures 2 and 3 demonstrate a somewhat more involved example of a ViTSL template and a corresponding Gencel spreadsheet application for a budget calculation. We will use this application to illustrate some limitations of the ViTSL/Gencel approach as a motivation for the proposed ClassSheet model.

Figure 2 shows that a hex group has been defined by clustering columns C, D, and E. The underlying problem domain requirement was that for each year (and for each category) the values Qnty, Cost and their product form a logical unit and should occur in the spreadsheet. The only way to express this logical clustering of three cells within ViTSL is by the omission of layout-oriented notations as the two small vertical bars in the header row between C, D, and E. In Gencel, this grouping causes the corresponding insertion and deletion of groups of three columns as blocks.[2]

Now imagine that the ViTSL designer would have grouped only the cells D and E, which would solely be visible in ViTSL by an additional bar within the header row between the columns C and D. This notationally minimally different ViTSL model would have resulted in a completely different spreadsheet application, in which the horizontal repetition would have been restricted to the two columns D and E. This different grouping would express that the quantity value is fixed for all years, while only the cost value might vary yearly.

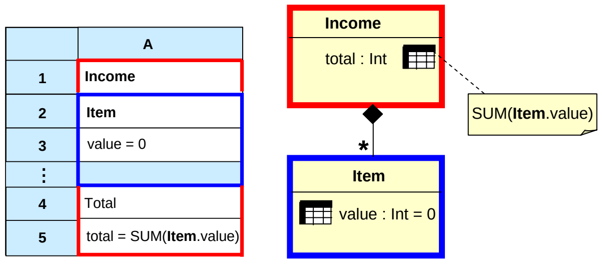
Another possible source for an error-prone spreadsheet model is due to the indication of references in formulas by means of cell-oriented addresses like C4*D4. Here, once more, the use of business logic-oriented notations like Qnty*Cost helps to prevent the design of incorrect data computations.

Therefore, since ViTSL is limited to the support of layout-oriented clustering constraints and cannot express problem-domain-oriented logical clustering according to business objects explicitly, the semantic gap between problem domain requirements and a spreadsheet application still forms a major obstacle to yield trustable spreadsheet applications.

3. CLASS SHEETS

In this section, we will introduce our approach of a high-level, object-oriented model for spreadsheet applications. A formalization of the approach is presented in the three subsequent sections.

In order to motivate the introduction of a business application-oriented structure on top of a layout-oriented ViTSL template structure, we discuss in the following three simple example spreadsheet applications. The first one (see Figure 5, left), the so-called *income sheet*, consists of a list of data values, which are summed up and the sum of which is shown in a separate cell. From an object-oriented point of view, one can see a summation object, which aggregates a list of objects bearing single data values. Looking at the layout structure, the list of data values is extended by a single header Item.[3] This structure is embedded into the layout of the summation object, consisting of a header entry **Income** and a footer with the label Total and an aggregation formula assigned to an attribute named total. We call such an object-oriented extended template a *ClassSheet* since it defines classes together with their attributes and aggregational relationships.



$$\mathsf{Income}\,\hat{}\,\mathsf{Item}\,\hat{}\,0^{\downarrow}\,\hat{}\,\mathsf{Total}\,\hat{}\,\mathsf{SUM}((0,-2))$$

Figure 5: A simple one-dimensional ClassSheet.

Thus, ClassSheets consist of a list of attribute definitions grouped by classes and are arranged on a two dimensional grid. Additional labels are used to annotate the concrete representation. Class names are set in boldface in contrast to attribute names and labels, which are set in normal face. In addition, colored borders are used to depict the different classes within a ClassSheet.[4]

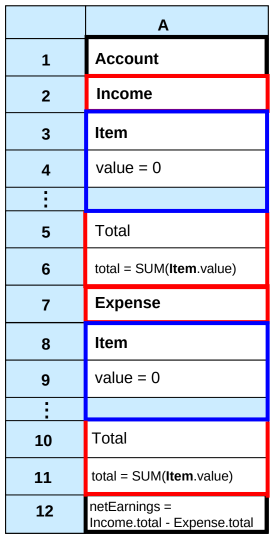
Class parts may be spread over header and footer entries, which results in a bracket-like structure indicated by a square-bracket-like notation of (open) class rectangles. For example, in Figure 5, the red class **Income** is split into a header and footer part that surrounds the blue class **Item**. References to other entries, being expressed in ViTSL by coordinates, are defined by using attribute names, as shown in the SUM formula in the example. Summarizing, ClassSheets subsume all the information of an equivalent ViTSL template and can thus easily be translated into an equivalent abstract ViTSL expression (see Figure 5, bottom[5]). Similarly, a UML-like representation may be derived from a ClassSheet (see Figure 5, right) by forgetting all layout information. Aggregation formulas are added as notes to the corresponding attribute definition. Those attributes, called derived attributes in UML, are tagged by a spreadsheet symbol on the right of the attribute definition. All other attributes the

[2]Note that when merging cells in MS Excel all but the first cell entries are lost, so that this groupwise operation is not possible at all in MS Excel.

[3]Note that this label is *not* repeated vertically because the separating line between the row numbers 2 and 3 causes the vertical repeating dots to apply only to row number 3. (A repetition of multiple cells is illustrated later in Figure 7.)

[4]We assume in the following that through the use of PDF this is visible to the reader.

[5]The downarrow is applied only to the value 0 since only the value attribute is vertically repeated, and not the label. The relative reference $(0,-2)$ would be shown in the concrete ViTSL syntax as a cell address A3.



	A
1	**Account**
2	**Income**
3	**Item**
4	value = 0
⋮	
5	Total
6	total = SUM(**Item**.value)
7	**Expense**
8	**Item**
9	value = 0
⋮	
10	Total
11	total = SUM(**Item**.value)
12	netEarnings = Income.total - Expense.total

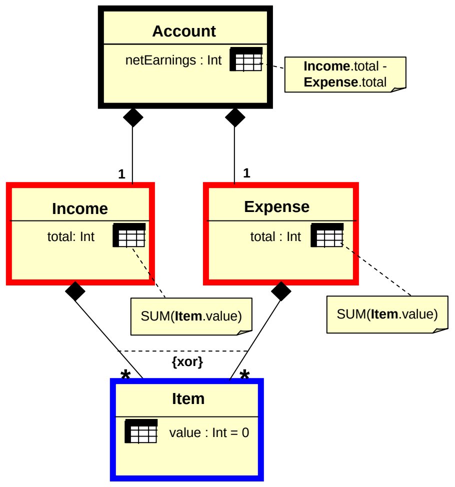
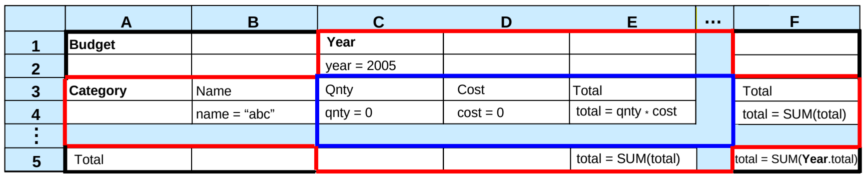


Figure 6: A more complex one-dimensional ClassSheet.



	A	B	C	D	E	…	F
1	**Budget**		**Year**				
2			year = 2005				
3	**Category**	Name	Qnty	Cost	Total		Total
4		name = "abc"	qnty = 0	cost = 0	total = qnty * cost		total = SUM(total)
⋮							
5	Total				total = SUM(total)		total = SUM(**Year**.total)

Figure 7: A two-dimensional ClassSheet.

values of which are shown in a spreadsheet are tagged by a spreadsheet symbol on the left of the attribute definition.

Within a spreadsheet design process, it is intended that the designer works in the first place with a sophisticated ClassSheet editor. In addition, within a fully-fledged design environment, the UML class diagram presentation might be offered to the (expert) designer, who is knowledgeable in UML, to illustrate her design decisions. In the following we use UML class diagrams as an additional documentation of the class structure in ClassSheets.

This first example is one of the simplest ClassSheets one can imagine. Nevertheless, it consists already of a class pattern of two classes representing a one-dimensional aggregation structure. A slightly more involved example is shown in Figure 6. Here, in addition to income values, a list of expense values together with an aggregated value is shown. In the overall footer, the difference of the two aggregated values is computed and shown. From an object-oriented point of view, we have an enclosing aggregating class **Account** with two enclosed aggregated structures **Income** and **Expense**. This structure is directly reflected in the UML diagram representation. Please note how the disjointness constraint xor in the UML class diagram for **Item** instances is spatially represented in the ClassSheet by two different occurrences of the **Item** class. In the ClassSheet, in addition to this information the layout structure is fixed, indicating that a column-oriented style is chosen.

The budget sheet presented in Figure 2 is shown as a ClassSheet in Figure 7. It is expressed that the budget consists of a repeatable (blue) kernel, with entries for quantity, cost, and their product. These entries are embedded into a two-dimensional tabular structure with a horizontal (red) row class **Category** and a vertical (red) column class **Year**. The two aggregated total values for a summation over years and categories are defined in these bracketing classes. The overall total is defined in an enclosing (black) bracket class **Budget**. The corresponding UML diagram of this well-structured ClassSheet is shown in Figure 8.

It shows how the tabular structure of this ClassSheet is directly reflected by an $n * m$-ary association between the two spanning classes **Category** and **Year**, acting as component classes of the root class Budget.

Figure 9 shows the variant of the budget sheet as it was discussed at the end of Section 2. The additional and maybe unintentionally placement of the tiny bar between columns

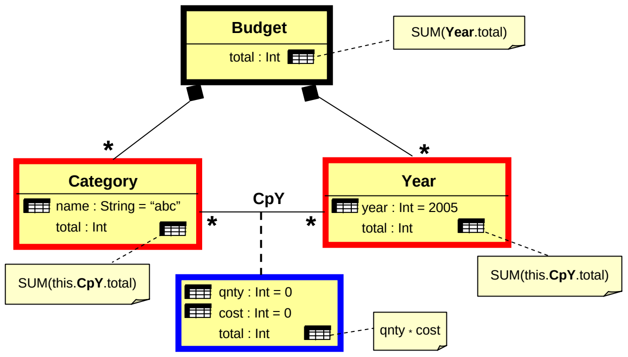
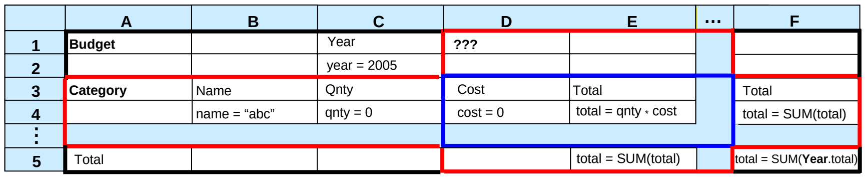


Figure 8: UML diagram for the budget ClassSheet.



	A	B	C	D	E	…	F
1	**Budget**		Year	**???**			
2			year = 2005				
3	**Category**	Name	Qnty	Cost	Total		Total
4		name = "abc"	qnty = 0	cost = 0	total = qnty * cost		total = SUM(total)
⋮							
5	Total				total = SUM(total)		total = SUM(**Year**.total)

Figure 9: Variation of the budget ClassSheet.

C and D would result in this completely different ClassSheet. The semantic differences are clearly visible in the corresponding UML class diagram (see Figure 10). In particular, it can be seen that the attribute qnty would be defined within class **Category**, which has the effect that it cannot be varied over the different years. In addition, the value of year (with the default value of 2005) would be fixed, too, since it would be defined in the overall root class **Budget**. Finally, a name for the column class would be missing (indicated by the label **???**) since Year would only be viewed as a label in the concrete layout while **Budget** defines the name of the root class.

It is important to notice that during such a design process, it is not possible to speak about "erroneous" design decisions. It might indeed be correct for the current business application that a fixed qnty value for all years is the right design decision. On the other hand, it is important that the explicit indication of class structures supports the spreadsheet designer to become aware of these different design choices. Using the ClassSheet approach, she will be supported to reason about alternatives, while consequences of design decisions become directly visible by the indicated, colored structures within the ClassSheet model. The ClassSheet approach helps to reduce the semantic gap between concrete problem domain requirements and its representation in a spreadsheet model.

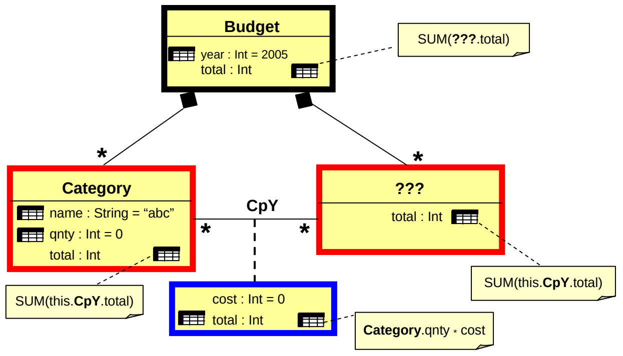


Figure 10: UML diagram for the variation budget ClassSheet.

4. A FORMAL MODEL OF CLASS SHEETS

In the following, we formalize the ClassSheet approach. In this section we define the abstract syntax. Typing rules to ensure well-formed ClassSheets are developed in the next section. Section 6 defines the translation of well-formed ClassSheets into ViTSL templates, which facilitates the connection of the ClassSheet approach to the existing transformation chain providing ClassSheet-compatible MS Excel applications.

A ClassSheet represents two aspects of a (spreadsheet) application—the structure and relationships of the involved objects and classes and the computational details of how attributes are related and derived from one another. To retain the well accepted successful table metaphor of spreadsheets, we embed the definition of these aspects into a grid-based layout that looks very much like a plain spreadsheet in which additional higher-level structures are visually exposed.

Therefore, the definition of ClassSheets is based on rectangular blocks of cells, which may contain values (used as labels) and attribute definitions. The latter are given by an attribute name (a) and a formula (f) that defines the values of the attribute. In addition, cells in a block might be left blank ($\sqcup$), for example, when used as a filler. The syntax of formulas is essentially the same as for ViTSL, except that references to other cells are expressed through qualified attribute names instead of relative references. These attribute names will be translated into cell references in the translation into ViTSL templates in Section 6.

The association of blocks with classes is expressed by explicitly assigning class names (n) to blocks. The distinction between column, row, table, and cell classes is indicated by a horizontal and/or vertical line indicating the dimension of the corresponding class type or by a "." in the case of cell classes.

To summarize, the ViTSL syntax is lifted into an abstract syntax for ClassSheets through the following additions and changes.

(1) References in formulas are replaced by qualified attribute names ($n.a$).
(2) Individual cells are not simply given by formulas (f), but either by values (φ), to represent (possibly blank) labels, or by attribute definitions ($a = f$).
(3) Each block that is used to build a class is associated with a label (ℓ), which is either a row ($\underline{n}$), a column ($|n$), a table ($|\underline{n}$), or a cell ($.n$) class label.

The syntax of ClassSheets is summarized in Figure 11. We have deliberately overloaded the nonterminals from the ViTSL syntax to illustrate the embedding into ClassSheets. The distinction between horizontal and vertical class labels has been made to facilitate the definition of well-formed ClassSheets by the use of so-called tiling rules in Section 5.

Let us illustrate the definition with some examples. A ClassSheet for a labeled list of numeric items can be represented by two cells, a label Item used as header and a vertically repeated attribute definition. The block formed from those two cells through vertical composition is labeled with the class name **Item**, which is marked as a vertical class label.

$$\begin{array}{llll}
f \in \mathit{Fml} & ::= & \varphi \mid n.a \mid \varphi(f,\ldots,f) & (\mathit{formulas})\\
b \in \mathit{Block} & ::= & \varphi \mid a = f \mid b \,\vert\, b \mid b\hat{\ }b & (\mathit{blocks})\\
\ell \in \mathit{Lab} & ::= & h \mid v \mid {}_{.}n & (\mathit{class\ labels})\\
h \in \mathit{Hor} & ::= & \underline{n} \mid \vert\underline{n} & (\mathit{horizontal})\\
v \in \mathit{Ver} & ::= & \vert n \mid \vert\underline{n} & (\mathit{vertical})\\
c \in \mathit{Class} & ::= & \ell : b \mid \ell : b^{\downarrow} \mid c\hat{\ }c & (\mathit{classes})\\
s \in \mathit{Sheet} & ::= & c \mid c^{\rightarrow} \mid s \,\vert\, s & (\mathit{sheets})
\end{array}$$

Figure 11: ClassSheet syntax.

$$\begin{array}{l}
\vert\mathbf{Item} : \mathsf{Item}\\
\qquad\quad (\mathsf{value} = 0)^{\downarrow}
\end{array}$$

For better readability, we employ the abbreviation that a new line implies vertical composition ˆ.

Next, we build a one-dimensional aggregation structure to represent the summation of values as, for example, used in the income sheet that is shown in Figure 5.

$$\begin{array}{l}
\vert\mathbf{Income} : \mathsf{Income}\\
\vert\mathbf{Item} : \mathsf{Item}\\
\qquad\quad (\mathsf{value} = 0)^{\downarrow}\\
\vert\mathbf{Income} : \mathsf{Total}\\
\qquad\qquad\quad \mathsf{total} = \mathsf{SUM}(\mathbf{Item}.\mathsf{value})
\end{array}$$

By renaming the **Income** class and changing the topmost label we can obtain from the income ClassSheet a corresponding ClassSheet for expenses. Combining the income and expense ClassSheets in a further aggregation structure leads to the following representation of the accounting sheet from Figure 6. To simplify the translation of ClassSheets, we require that attributes (qualified by the class names that label the block in which they occur) are unique in the abstract syntax. Since the **Item** class is used twice in the accounting sheet, the second occurrence has to be renamed to ensure this constraint for the value attribute.

$$\begin{array}{l}
\vert\mathbf{Account} : \mathsf{Account}\\
\vert\mathbf{Income} : \mathsf{Income}\\
\vert\mathbf{Item} : \mathsf{Item}\\
\qquad\quad (\mathsf{value} = 0)^{\downarrow}\\
\vert\mathbf{Income} : \mathsf{Total}\\
\qquad\qquad\quad \mathsf{total} = \mathsf{SUM}(\mathbf{Item}.\mathsf{value})\\
\vert\mathbf{Expense} : \mathsf{Expenses}\\
\vert\mathbf{Item'} : \mathsf{Item}\\
\qquad\quad (\mathsf{value} = 0)^{\downarrow}\\
\vert\mathbf{Expense} : \mathsf{Total}\\
\qquad\qquad\quad \mathsf{total} = \mathsf{SUM}(\mathbf{Item'}.\mathsf{value})\\
\vert\mathbf{Account} : \mathsf{netEarnings} = \mathbf{Income}.\mathsf{total}\text{-}\mathbf{Expense}.\mathsf{total}
\end{array}$$

Finally, we demonstrate in Figure 12 how the two-dimensional budget ClassSheet can be defined.

This ClassSheet is built by horizontally composing three blocks: The first block is two columns wide and consists of three parts, two blocks belonging to the root class **Budget** and a row aggregation class **Category**, whose block is vertically repeated. The middle block is three columns wide and is horizontally repeated. Like the first block it consists of two classes, the column aggregation class **Year**, which, together with **Category**, encloses the vertically repeated association cell class **CpY**. The final one-column block is very similar to

$$\begin{array}{l}
\vert\underline{\mathbf{Budget}} : \mathsf{Budget} \,\vert\, \sqcup\\
\qquad\qquad\ \ \sqcup \,\vert\, \sqcup\\
\ \underline{\mathbf{Category}} : \big(\mathsf{Category} \,\vert\, \mathsf{Name}\\
\qquad\qquad\qquad \sqcup \,\vert\, \mathsf{name} = \text{"abc"}\big)^{\downarrow}\\
\vert\underline{\mathbf{Budget}} : \mathsf{Total} \,\vert\, \sqcup\\
\vert\\
\ (\vert\mathbf{Year} : \mathsf{Year} \,\vert\, \sqcup \,\vert\, \sqcup\\
\qquad\qquad \mathsf{year} = 2005 \,\vert\, \sqcup \,\vert\, \sqcup\\
\quad {}_{.}\mathbf{CpY} : \big(\mathsf{Qnty} \,\vert\, \mathsf{Cost} \,\vert\, \mathsf{Total}\\
\qquad\qquad \mathsf{qnty} = 0 \,\vert\, \mathsf{cost} = 0 \,\vert\, \mathsf{total} = \mathsf{qnty}{*}\mathsf{cost}\big)^{\downarrow}\\
\quad \vert\mathbf{Year} : \sqcup \,\vert\, \sqcup \,\vert\, \mathsf{total} = \mathsf{SUM}(\mathbf{CpY}.\mathsf{total})\big)^{\rightarrow}\\
\vert\\
\vert\underline{\mathbf{Budget}} : \sqcup\\
\qquad\qquad\ \ \sqcup\\
\ \underline{\mathbf{Category}} : \big(\mathsf{Total}\\
\qquad\qquad\qquad \mathsf{total} = \mathsf{SUM}(\mathbf{CpY}.\mathsf{total})\big)^{\downarrow}\\
\vert\underline{\mathbf{Budget}} : \mathsf{total} = \mathsf{SUM}(\mathbf{Year}.\mathsf{total})
\end{array}$$

Figure 12: Abstract syntax representation of the budget ClassSheet.

the first block, in particular, it consists of the same classes to complete the enclosure of the middle block and the cell class.

5. ENFORCING CLASS SHEET STRUCTURE THROUGH TILINGS

The abstract syntax of ClassSheet cannot capture structural constraints resulting from the two-dimensional layout. For example, the income sheet in Figure 5 consists of two classes, one of which is spatially nested within the other. The abstract syntax allows the definition of a sheet like the following.

$$\begin{array}{l}
\vert\mathbf{Income} : \mathsf{Income}\\
\vert\mathbf{Item} : \mathsf{Item}\\
\vert\mathbf{Income} : (\mathsf{value} = 0)^{\downarrow}\\
\vert\mathbf{Item} : \mathsf{Total}\\
\vert\mathbf{Income} : \mathsf{total} = \mathsf{SUM}(\mathbf{Item}.\mathsf{value})
\end{array}$$

The implied spatial structure suggests that **Income** aggregates **Item**, which itself aggregates **Income**, which doesn't make any sense and should therefore be ruled out as a possible ClassSheet.

In general, the annotation of cells with classes has to result in a regularly nested class structure. This condition can be formalized through a type system that infers a spatial structure called *tiling* for ClassSheets. Only ClassSheets for which the rule system is able to infer a proper tiling are considered to be well formed.

When we consider the previous examples, we can identify four principal tiling structures, namely for non-aggregated single classes, for one-dimensional, horizontally or vertically aggregated classes, and for two-dimensional aggregations. Aggregation tilings can be principally nested, and tilings can also be horizontally and vertically composed. These considerations lead to the definition of tilings shown in Figure 13.

Let us illustrate the tiling syntax with some examples. The simple item sheet has tiling "■". The income sheet (as well as the expense sheet) is defined as a vertical, one-dimensional aggregation and thus has the tiling $\langle$■$\rangle$. We

$$\begin{array}{rcll} \tau & ::= & \blacksquare \mid \theta \mid \phi & (\textit{tilings}) \\ \theta & ::= & [\tau] \mid \boxed{\tau} \mid \theta \,|\, \theta & (\textit{horizontal tilings}) \\ \phi & ::= & \langle\tau\rangle \mid \boxed{\tau} \mid \phi \,\hat{}\, \phi & (\textit{vertical tilings}) \end{array}$$

Figure 13: Tilings.

could express the income sheet alternatively as a horizontal aggregation, which would have the tiling $[\blacksquare]$. The accounting sheet is a vertical aggregation over two vertical aggregations, which is reflected by the tiling $\langle\langle\blacksquare\rangle\,\hat{}\,\langle\blacksquare\rangle\rangle$. Finally, the budget sheet provides an example of a two-dimensional aggregation, which leads to the simple table tiling $\boxed{\blacksquare}$. A two-dimensional visualization of the different tiling structures is shown in Figure 14.

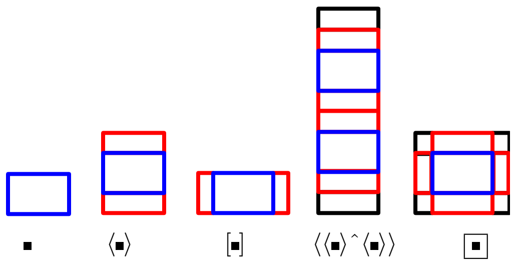

$\blacksquare$
$\langle\blacksquare\rangle$
$[\blacksquare]$
$\langle\langle\blacksquare\rangle\,\hat{}\,\langle\blacksquare\rangle\rangle$
$\boxed{\blacksquare}$

Figure 14: Visualization of tiling structures.

In the following, we define a tiling judgment $\vdash s :: \tau$ that expresses that the tiling of the ClassSheet s is τ. We use the metavariable β to range over b and $b^{\downarrow}$. The inference rules express constraints on the width and height of tilings, which are essentially given by the nesting depth in horizontal and vertical dimension, respectively. Formally, $\updownarrow\tau$ and $\overleftrightarrow{\tau}$ are defined as follows.

$$\begin{array}{lcllcl} \overleftrightarrow{\blacksquare} & = & 1 & \updownarrow\blacksquare & = & 1 \\ \overleftrightarrow{\tau_1 \,|\, \tau_2} & = & \overleftrightarrow{\tau_1} + \overleftrightarrow{\tau_2} & \updownarrow\tau_1 \,|\, \tau_2 & = & \max(\updownarrow\tau_1, \updownarrow\tau_2) \\ \overleftrightarrow{\tau_1 \,\hat{}\, \tau_2} & = & \max(\overleftrightarrow{\tau_1}, \overleftrightarrow{\tau_2}) & \updownarrow\tau_1 \,\hat{}\, \tau_2 & = & \updownarrow\tau_1 + \updownarrow\tau_2 \\ \overleftrightarrow{[\tau]} & = & \overleftrightarrow{\tau} + 1 & \updownarrow[\tau] & = & \updownarrow\tau \\ \overleftrightarrow{\langle\tau\rangle} & = & \overleftrightarrow{\tau} & \updownarrow\langle\tau\rangle & = & \updownarrow\tau + 1 \\ \overleftrightarrow{\boxed{\tau}} & = & \overleftrightarrow{\tau} + 1 & \updownarrow\boxed{\tau} & = & \updownarrow\tau + 1 \end{array}$$

The tiling judgment is formalized by the rules shown in Figure 15.

The rules implicitly exploit the following equivalence relationship of classes

$$\ell : \beta \,\hat{}\, \ell : \beta' \equiv \ell : \beta \,\hat{}\, \beta'$$

This equivalence says that two vertically composed blocks with the same class name can be combined into one since it makes no differene as far as access to field names is concerned. This equivalence can be employed (from left to right) to transform ClassSheets into a normal form that allows the rule system to consider vertical compositions of classes with different labels only.

We illustrate the rule system by deriving the tiling structure for the budget sheet. To apply rule Tab, we first let $v = |\underline{\textsf{\textbf{Budget}}}$, $\beta_1 = \textsf{Budget} \,|\, \sqcup \,\hat{}\, \sqcup \,|\, \sqcup$, $\beta_2 = \textsf{Total} \,|\, \sqcup$, and $c_1 = \underline{\textsf{\textbf{Category}}} : (\textsf{Category} \,|\, \textsf{Name} \,\hat{}\, \sqcup \,|\, \textsf{name} = \textsf{"abc"})^{\downarrow}$ (recall that $|$ binds stronger than $\hat{}$). Now we can directly apply rule Base to show that $\vdash c_1 :: \blacksquare$. Next we can apply rule Col (with $c = c_1$, $\beta = \beta_1$, $\beta' = \beta_2$, and $\tau = \blacksquare$) to conclude $\vdash c_1' :: \langle\blacksquare\rangle$, which provides the first premise for rule Tab. The premises for c_2' and c_3' are obtained in a similar way. Finally, the rule Row* can be applied to show the last premise since the horizontal class labels in c_1 and in c_3 are the same (namely, $\underline{\textsf{\textbf{Category}}}$).

$$\textsc{Base}\ \frac{}{\vdash \ell : \beta :: \blacksquare} \qquad \textsc{Row}\ \frac{\vdash c :: \tau}{\vdash h : \beta \,|\, c \,|\, h : \beta' :: [\tau]}$$

$$\textsc{Row*}\ \frac{\vdash c :: \tau}{\vdash h : \beta \,|\, c^{\rightarrow} \,|\, h : \beta' :: [\tau]} \qquad \textsc{Col}\ \frac{\vdash c :: \tau}{\vdash v : \beta \,\hat{}\, c \,\hat{}\, v : \beta' :: \langle\tau\rangle}$$

$$\textsc{Hor}\ \frac{\vdash s_1 :: \tau_1 \quad \vdash s_2 :: \tau_2 \quad \updownarrow\tau_1 = \updownarrow\tau_2}{\vdash s_1 \,|\, s_2 :: \tau_1 \,|\, \tau_2}$$

$$\textsc{Ver}\ \frac{\vdash c_1 :: \tau_1 \quad \vdash c_2 :: \tau_2 \quad \overleftrightarrow{\tau_1} = \overleftrightarrow{\tau_2}}{\vdash c_1 \,\hat{}\, c_2 :: \tau_1 \,\hat{}\, \tau_2}$$

$$\textsc{Tab}\ \frac{\begin{array}{c} c_1' = v : \beta_1 \,\hat{}\, c_1 \,\hat{}\, v : \beta_2 \\ c_2' = v' : \beta_3 \,\hat{}\, c_2 \,\hat{}\, v' : \beta_4 \\ c_3' = v : \beta_5 \,\hat{}\, c_3 \,\hat{}\, v : \beta_6 \\ \vdash c_1' :: \langle\tau\rangle \quad \vdash c_2' :: \langle\tau\rangle \quad \vdash c_3' :: \langle\tau\rangle \\ \vdash c_1 \,|\, c_2 \,|\, c_3 :: [\tau] \end{array}}{\vdash c_1' \,|\, c_2' \,|\, c_3' :: \boxed{\tau}}$$

Figure 15: Tiling inference rules.

With the formalization of the tiling structure of ClassSheets, we can now define that a ClassSheet s is *well formed* only if there exists a tiling τ such that $\vdash s :: \tau$.

6. TRANSLATING CLASS SHEETS INTO VITSL

Whereas the translation of ClassSheets into UML ignores layout and labels that are not attribute or class names, the translation into ViTSL templates retains exactly the block structure with all labels. However, attribute names used as references in formulas are translated into references based on row/column numbers, and class and attribute names are forgotten.

The translation from ClassSheets to ViTSL templates can be described in two steps: First, the block structure is translated by a function $\mathcal{B}$, which is defined by a rather straightforward traversal of the structure that simply forgets all class annotations for blocks. After that, attribute names in formulas are translated into references by a function $\mathcal{F}$.

The block translation is defined as follows. The actual work happens in the last two lines.

$$\begin{array}{lcl} \mathcal{B}(s_1 \,|\, s_2) & = & \mathcal{B}(s_1) \,|\, \mathcal{B}(s_2) \\ \mathcal{B}(c^{\rightarrow}) & = & \mathcal{B}(c)^{\rightarrow} \\ \mathcal{B}(c_1 \,\hat{}\, c_2) & = & \mathcal{B}(c_1) \,\hat{}\, \mathcal{B}(c_2) \\ \mathcal{B}(\ell : b) & = & b \\ \mathcal{B}(\ell : b^{\downarrow}) & = & b^{\downarrow} \end{array}$$

The result of applying $\mathcal{B}$ to a ClassSheet is a kind of “hybrid” template, that is, it is a template t according to the abstract syntax of ViTSL that contains formulas as defined for class sheets, which have not yet been translated.

The second step is performed by the function $\mathcal{F}$, which takes as additional arguments (a) the current position (to be able to translate references) and (b) the original hybrid block as returned by $\mathcal{B}$ (to be able to find attribute definitions even after the attribute names have been stripped off). The function makes use of several auxiliary functions. First, it employs two functions for computing the width and height of a template, which are defined similarly to the functions from Section 5. To compress the definitions for horizontal and vertical composition and repetition, we use the metavariable u to range over b, c, and t.

$$\begin{array}{llll}
\overleftrightarrow{f} & = 1 & \updownarrow f & = 1 \\
\overleftrightarrow{u_1 \mid u_2} & = \overleftrightarrow{u_1} + \overleftrightarrow{u_2} & \updownarrow u_1 \mid u_2 & = \max(\updownarrow u_1, \updownarrow u_2) \\
\overleftrightarrow{u_1 \hat{} u_2} & = \max(\overleftrightarrow{u_1}, \overleftrightarrow{u_2}) & \updownarrow u_1 \hat{} u_2 & = \updownarrow u_1 + \updownarrow u_2 \\
\overleftrightarrow{u^{\downarrow}} & = \overleftrightarrow{u} & \updownarrow u^{\downarrow} & = \updownarrow u \\
\overleftrightarrow{u^{\rightarrow}} & = \overleftrightarrow{u} & \updownarrow u^{\rightarrow} & = \updownarrow u
\end{array}$$

Second, a function for determining the location of a particular attribute definition is needed. The function $\mathcal{L}$ traverses the template in search of the definition of attribute $n.a$ while updating the current position (x, y). Once a class with a matching name n is found, the search continues for the attribute name a within the block.

$$\begin{array}{ll}
\mathcal{L}^{(x,y)}(n.a, s \mid s') & = \mathcal{L}^{(x,y)}(n.a, s) \cup \mathcal{L}^{(x+\overleftrightarrow{s},y)}(n.a, s') \\
\mathcal{L}^{(x,y)}(n.a, c^{\rightarrow}) & = \mathcal{L}^{(x,y)}(n.a, c) \\
\mathcal{L}^{(x,y)}(n.a, c \hat{} c') & = \mathcal{L}^{(x,y)}(n.a, c) \cup \mathcal{L}^{(x,y+\updownarrow c)}(n.a, c') \\
\mathcal{L}^{(x,y)}(n.a, n : b) & = \mathcal{L}^{(x,y)}(a, b) \\
\mathcal{L}^{(x,y)}(n.a, n : b^{\downarrow}) & = \mathcal{L}^{(x,y)}(a, b) \\
\mathcal{L}^{(x,y)}(a, b \mid b') & = \mathcal{L}^{(x,y)}(a, b) \cup \mathcal{L}^{(x+\overleftrightarrow{b},y)}(a, b') \\
\mathcal{L}^{(x,y)}(a, b \hat{} b') & = \mathcal{L}^{(x,y)}(a, b) \cup \mathcal{L}^{(x,y+\updownarrow b)}(a, b') \\
\mathcal{L}^{(x,y)}(a, a = f) & = \{(x, y)\} \\
\mathcal{L}^{(x,y)}(a, f) & = \varnothing
\end{array}$$

$\mathcal{L}$ computes a set of locations. If the resulting set does not consist of a single address, an error situation has occurred: If the resulting set is empty, the attribute was not found, if the set contains more than one value, the qualified attribute name was not unique. Although described here as part of the translation, the resolving of all references can be added as an additional test to the type checking phase to avoid the generation of spreadsheet applications for ambiguous and ill-defined ClassSheets.

With these helper functions the function $\mathcal{F}$ can be defined as follows.

$$\begin{array}{ll}
\mathcal{F}_s^{(x,y)}(u_1 \mid u_2) & = \mathcal{F}_s^{(x,y)}(u_1) \mid \mathcal{F}_s^{(x+\overleftrightarrow{u_1},y)}(u_2) \\
\mathcal{F}_s^{(x,y)}(u^{\rightarrow}) & = \mathcal{F}_s^{(x,y)}(u)^{\rightarrow} \\
\mathcal{F}_s^{(x,y)}(u_1 \hat{} u_2) & = \mathcal{F}_s^{(x,y)}(u_1) \hat{} \mathcal{F}_s^{(x,y+\updownarrow u_1)}(u_2) \\
\mathcal{F}_s^{(x,y)}(b^{\downarrow}) & = \mathcal{F}_s^{(x,y)}(b)^{\downarrow} \\
\mathcal{F}_s^{(x,y)}(a = f) & = \mathcal{F}_s^{(x,y)}(f) \\
\mathcal{F}_s^{(x,y)}(\varphi) & = \varphi \\
\mathcal{F}_s^{(x,y)}(\varphi(f_1, \ldots, f_n)) & = \varphi(\mathcal{F}_s^{(x,y)}(f_1), \ldots, \mathcal{F}_s^{(x,y)}(f_n)) \\
\mathcal{F}_s^{(x,y)}(n.a) & = \begin{cases} (i - x, j - y) & \text{if } \mathcal{L}^{(1,1)}(n.a, s) = \{(i, j)\} \\ \perp & \text{otherwise} \end{cases}
\end{array}$$

Finally, the complete translation of a ClassSheet s into a ViTSL template is given by $\mathcal{F}_s^{(1,1)}(\mathcal{B}(s))$.

This completes the stepwise formalization of the automatic transformation chain from a user-defined ClassSheet into a running MS Excel spreadsheet application. As illustrated in Figure 1, ClassSheets that are well-formed according to an underlying tiling structure are transformed into an equivalent ViTSL expression. Such a ViTSL expression forms the input for the Gencel tool, which controls a spreadsheet application that is compliant with the class structure as defined in the ClassSheet. In this paper, we have made use of UML class diagrams to illustrate the underlying class structure. It is not intended to show such an abstract view of the ClassSheet to the average spreadsheet designer. We rather want to overlay class-structure information with the well-accepted two-dimensional spreadsheet representation within one single notation, ClassSheets, which we have introduced and formally defined.

7. RELATED WORK

Any kind of system development support activities might be distinguished into constructive and analytical activities. Constructive activities aim at the development of high-quality systems with the objective to avoid erroneous decisions as much as possible from the beginning, while analytical activities are striving for the detection of (possibly) erroneous situations before or during a system execution.

Previous scientific work on spreadsheets has mainly addressed questions of analyzing spreadsheet applications like testing/debugging [23, 22] and consistency checking [13, 9, 6, 7, 4].

The "What You See Is What You Test" methodology for testing spreadsheets [23] employs data-flow adequacy criteria and coverage monitoring to provide feedback about the "testedness" of a spreadsheet. This approach has been extended by fault localization to isolate sources of errors [22].

User-provided assertions about permissible cell ranges have been used in [9] to identify erroneous formulas. Assertions are systematically propagated through the spreadsheet formulas to generate warnings whenever assertions are violated. A formal reasoning system for detecting spreadsheet errors based on a classification of spreadsheet contents into units was proposed in [13]. A related approach is reported in [6], which extends the unit system by a new kind of relationship. The described approach requires the user to manually annotate spreadsheets with header information. User annotations are also required in the approach of [7], which performs unit checking based on the actual physical or monetary units of the data in the spreadsheet. We have recently implemented an automatic unit checker, called UCheck, that is based on the automatic inference of header information based on different aspects of the spatial layout of spreadsheets [4].

The FFR model presented in [24] helps to analyze visualization mechanisms for spreadsheets. In this approach, errors in spreadsheet formulas show up as anomalies in the visualizations. A similar approach of identifying recurring structure (regions) in spreadsheets and then presenting anomalies as potential problem areas to the user is followed by the system described in [18].

While all these approaches focus on analytical activities, the ViTSL/Gencel approach [5, 11, 12] was targeted at the construction process of spreadsheets. Our ClassSheet approach inherits all the safety/correctness features of ViTSL/Gencel and provides an additional object-oriented modeling layer to further reduce the semantic gap between the application domains and the (automatically generated) applications.

Guidelines for the construction of spreadsheets can be found in numerous, popular business-oriented contributions (for example, [21, 19]). Since the underlying base is the cell-oriented tabular spreadsheet paradigm, they have to rely on methodological guidelines and are not able to take advantage of higher-level structuring means as they have been introduced in this paper.

8. CONCLUSIONS AND FUTURE WORK

ClassSheets present an effective combination of three successful approaches from different areas of software development into an automatic development approach: (1) Object-oriented modeling, (2) Spreadsheets, and (3) Template specifications.

The formalized ClassSheet language forms the basis for the development of corresponding spreadsheet design and development tools. In particular, the block-oriented structure of ClassSheets will lead to a new compositional, stepwise approach to build spreadsheet applications. Reuse is strongly supported and encouraged by this approach of composing new ClassSheets because predefined building blocks can be single classes or ClassSheets. The underlying formalization will ensure that only consistent, well-formed ClassSheets can be constructed, which are automatically translated into model-compliant spreadsheet applications.

As part of the ongoing evaluation work within the EUSES project [14] end-user experiments will be performed to obtain feedback from different types of spreadsheet users. Diversity will be covered according to different degrees of skills as well as gender and cultural aspects.

In this paper, we did not explore the transformation of ClassSheets to UML class diagrams in detail. It will be the task of future work to embed ClassSheets as UML 2.0 profile into a UML-driven development process. In particular, we will investigate whether the visual paradigm of ClassSheets might serve as an alternative for textual OCL expressions in case of defining derived attributes within classes. To this end, the expressive power of ClassSheet formulas has to be determined and compared to alternative visual approaches to express OCL constraints, such as spider or constraint diagrams [16].

9. REFERENCES

[1] EUD-Net. `http://giove.cnuce.cnr.it/eud-net.htm`.

[2] Sarbanes-Oxley Act. `http://news.findlaw.com/hdocs/docs/gwbush/sarbanesoxley072302.pdf`.

[3] UML 2.0 OCL Specification. `http://www.uml.org/`.

[4] R. Abraham and M. Erwig. Header and Unit Inference for Spreadsheets Through Spatial Analyses. *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pp. 165–172, 2004.

[5] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert. Visual Specifications of Correct Spreadsheets. *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, 2005. To appear.

[6] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A Type System for Statically Detecting Spreadsheet Errors. *18th IEEE Int. Conf. on Automated Software Engineering*, pp. 174–183, 2003.

[7] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the Unit Correctness of Spreadsheet Programs. *26th IEEE Int. Conf. on Software Engineering*, pp. 439–448, 2004.

[8] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, K. C. Bradford, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece, editors. *Software Cost Estimation with COCOMO II.* Prentice-Hall International, Upper Saddle River, NJ, 2000.

[9] M. M. Burnett, C. Cook, J. Summet, G. Rothermel, and C. Wallace. End-User Software Engineering with Assertions. *25th IEEE Int. Conf. on Software Engineering*, pp. 93–103, 2003.

[10] G. Engels and L. Groenewegen. Object-Oriented Modeling: A Roadmap. *ICSE'00: Conf. on The Future of Software Engineering*, pp. 103–116, 2000.

[11] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic Generation and Maintenance of Correct Spreadsheets. *27th IEEE Int. Conf. on Software Engineering*, pp. 136–145, 2005.

[12] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Gencel — A Program Generator for Correct Spreadsheets. *Journal of Functional Programming*, 2005. To appear.

[13] M. Erwig and M. M. Burnett. Adding Apples and Oranges. *4th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 2257, pp. 173–191, 2002.

[14] EUSES. End Users Shaping Effective Software. `http://EUSESconsortium.org`.

[15] EuSpRIG. European Spreadsheet Risks Interest Group. `http://www.eusprig.org/`.

[16] J. Gil, J. Howse, and S. Kent. Formalising Spider Diagrams. *15th IEEE Symp. on Visual Languages*, pp. 130–137, 1999.

[17] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture Practice and Promise.* Addison-Wesley, 2003.

[18] R. Mittermeir and M. Clermont. Finding High-Level Structures in Spreadsheet Programs. *9th Working Conference on Reverse Engineering*, pp. 221–232, 2002.

[19] P. O'Beirne. Agile Spreadsheet Development (ASD), 2003. `http://www.sysmod.com/agile.htm`.

[20] R. R. Panko. Spreadsheet Errors: What We Know. What We Think We Can Do. *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000.

[21] S. G. Powell and K. R. Baker. *The Art of Modeling with Spreadsheets: Management Science, Spreadsheet Engineering, and Modeling Craft.* Wiley, 2004.

[22] S. Prabhakarao, C. Cook, J. Ruthruff, E. Creswick, M. Main, M. Durham, and M. Burnett. Strategies and Behaviors of End-User Programmers with Interactive Fault Localization. *Int. Symp. on Human-Centric Computing Languages and Environments*, pp. 203–210, 2003.

[23] G. Rothermel, M. M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A Methodology for Testing Spreadsheets. *ACM Transactions on Software Engineering and Methodology*, pp. 110–147, 2001.

[24] J. Sajaniemi. Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization. *Journal of Visual Languages and Computing*, 11:49–82, 2000.