Systematic Evolution of Model-Based Spreadsheet Applications

Markus Luckey[a,*], Martin Erwig[b], Gregor Engels[a]

[a]*University of Paderborn, 33098 Paderborn, Germany*
[b]*Oregon State University, Corvallis, Oregon 97331-3202, USA*

Abstract

Using spreadsheets is the preferred method to calculate, display or store anything that fits into a table-like structure. They are often used by end users to create applications, although they have one critical drawback—spreadsheets are very error-prone. Recent research has developed methods to reduce this error-proneness by introducing a new way of object-oriented modeling of spreadsheets before using them. These spreadsheet models, termed ClassSheets, are used to generate concrete spreadsheets on the instance level. By this approach sources of errors are reduced and spreadsheet applications become easier to understand.

As usual for almost every other application, requirements on spreadsheets change due to the changing environment. Thus, the problem of evolution of spreadsheets arises. The update and evolution of spreadsheets is the uttermost source of errors that may have severe impact.

In this article, we will introduce a model-based approach to spreadsheet evolution by propagating updates on spreadsheet models (i.e. ClassSheets) to spreadsheets. To this end, update commands for the ClassSheet layer are automatically transformed to those for the spreadsheet layer. We describe spreadsheet model update propagation using a formal framework and present an integrated tool suite that allows the easy creation and safe update of spreadsheet models. The presented approach greatly contributes to the problem of software evolution and maintenance for spreadsheets and thus avoids many errors that might have severe impacts.

Keywords: model-based, spreadsheet, evolution, update, propagation

1. Introduction

Spreadsheets are one of the most popular and important programming languages used in business applications today. Estimates say that "each year tens of millions of managers and professionals around the world create hundreds of millions of spreadsheets" [1]. Reasons for this wide-spread use of spreadsheets are the ease of creating highly sophisticated spreadsheet applications using the simple and intuitive two-dimensional tabular layout and of course the fast and broad availability of spreadsheet applications. Due to their availability to non-experts, spreadsheets belong to the category of end-user development environments. However, the simplicity of spreadsheets is misleading. Although, the spreadsheet user group usually is able to develop complex spreadsheets, the users often do not have the knowledge to prevent errors leading to error-prone and unstructured spreadsheets [2]. Studies estimate rates of at least 80 percent of erroneous spreadsheets [3, 4]. How costly these errors can be is evident in recent news stories. For example, in 2006, the Office of Government Commerce Buying Solutions erred in informing 29 suppliers that they had been successful in the public sector tendering process. In a subsequent letter, they stated: "Unfortunately, [we are] not, as we had hoped, in a position to accept your tender at this time. This is because an error in the original evaluation spreadsheet has been identified, necessitating rescoring of all tenders for this project...this error has now been corrected and this has caused a small number of changes to the original award decision" [5].

Reasons for these failures are manifold, the most important reason being the missing business model. A business model of spreadsheets specifies which business entities are represented by the specific spreadsheet. For instance, a

---

*Corresponding author
Email addresses:* `luckey@upb.de` (Markus Luckey), `erwig@eecs.oregonstate.edu` (Martin Erwig), `engels@upb.de` (Gregor Engels)

spreadsheet for budget calculation may comprise entities like *category* or *year* (cf. Figure 1, categories in rows **4** and **5** and years in columns **C** upto **H**). A spreadsheet's business model is not given explicitly to the spreadsheet application,

	A	B	C	D	E	F	G	H	I
1	**Budget**		Year			Year			
2			2009			2010			
3	**Category**	Name	Qnty	Cost	Total	Qnty	Cost	Total	Total
4		Apples	0	0	=(C4*D4)	0	0	=(F4*G4)	=SUM(E4;H4)
5		Bananas	0	0	=(C5*D5)	0	0	=(F5*G5)	=SUM(E5;H5)
6	Total				=SUM(E4;E5)			=SUM(H4;H5)	=SUM(E6;H6)

Figure 1: A Spreadsheet for Budget Calculation

but usually is kept in the developer's idea and captured implicitly in the spreadsheet's layout and data. Taking into account the complexity of today's spreadsheets, the gap between the *implicit business model* of a spreadsheet and the resulting implementation (i. e. the spreadsheet itself) is too large. Figure 2 shows the current approach of spreadsheet development. The user has an idea of the spreadsheet's business model in mind and develops the spreadsheet accordingly. However, spreadsheet development is very low-level and current spreadsheet applications do not allow to implement all elements of the business model. The semantic gap between the implicit business model and the spread-
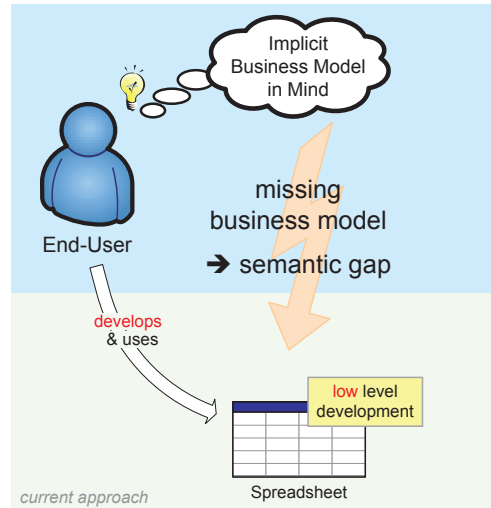

Figure 2: The current Spreadsheet Development Approach

sheet leads to misleading error reports (e. g. the spreadsheet application complains about a formula that differs from surrounding formulas) or even missed errors. Spreadsheet applications like Microsoft Excel fail to sufficiently mitigate this situation. The reason for this failure is that the business model is not described explicitly and thus cannot be used to automatically validate the current state of a spreadsheet (i. e. the inserted data, formulas, and references). Bridging the described gap between the business model (captured in the developer's idea) and the IT implementation is recently known under *Business/IT Alignment*. The need for Business/IT alignment was emphasized by the Sarbanes-Oxley Compliance (see [6]) leading to plenty of tools that are concerned with increasing the quality of use and development of spreadsheets. However, these approaches rather provide process guidance and support security aspects but do not tackle the problem at its root, the gap between business and IT. But how can the business model help to prevent errors?

Usually, errors are produced while changing a spreadsheet. We distinguish two different kinds of spreadsheet changes, namely instance evolution and model evolution. Both kinds of changes are part of *spreadsheet evolution*. Instance evolution describes changes that concern a spreadsheet's data but not its representation and interrelations. For instance, inserting a new category in the budget spreadsheet mentioned above, is part of instance evolution. In

turn, model evolution describes changes that concern the relation between data (e. g. formulas) or the insertion and deletion of data *types*. For instance, the insertion of a new column that holds a new *type* of data (e. g. an exchange rate for given costs) is a change at the underlying business model. Those changes must be applied to all inserted data, e. g. the exchange rate must be inserted for all categories and years. See Figure 3 for the distinction of instance evolution

Figure 3: Model and Instance Evolution for Spreadsheets

and model evolution. For reasons of understandability, we chose UML class diagrams [7] to represent the business model. Of course every other representation is applicable, e. g. the Entity Relationship Model [8]. Both instance and model evolution are sources of errors. The most frequent error in instance evolution is a missed or wrong update of a formula, such as the missed update of the total formulas in Figure 3. While Microsoft Excel supports updating formulas that range over connected regions (e. g. **A17:A42**) it lacks support for automatically updating formulas that range over unconnected regions, e. g. every second cell in a column (**A1;A3;A5**). Thus, insertion of data in complex large spreadsheets that contain many formulas over regions that are not connected is very prone to errors. The problem of model evolution is that a change at the implicit business model cause necessary changes on each inserted data record. For instance, adding the mentioned exchange rate to the budget sheet (cf. Figure 3 on the left) requires the insertion of multiple columns and the change of multiple formulas as shown in Figure 4. For complex large spreadsheets, this task

Figure 4: Model Evolution requires Multiple Changes at the Spreadsheet
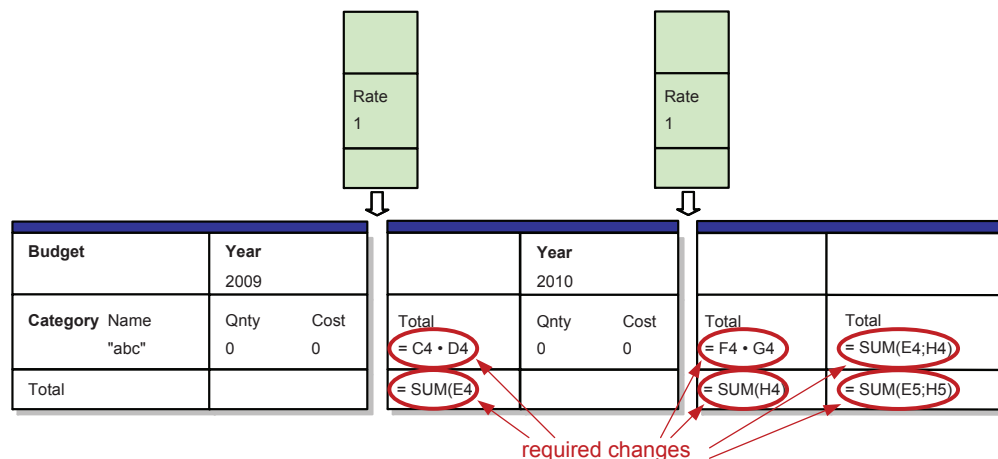
is not only burdensome but also very error-prone.

1.1. Solving the Problem of Instance Evolution

To solve the problem of instance evolution, an approach proposed by Engels and Erwig [9] introduces *ClassSheets* using proved and well-tried techniques from modern Software Engineering like object-orientation and model-driven development to yield Business/IT alignment. ClassSheets are explicit specifications of business models which can be translated to spreadsheets automatically. These explicit specifications support a better understanding of the underlying business model, yield less erroneous spreadsheets due to the distinction of business model and implementation, and insure the correctness of spreadsheets regarding their business model (i. e. ClassSheet) with fully automated tool support. But why are object-orientation and model-driven development a good choice for bridging the gap between business and IT?

Object-orientation, on the one hand, improves the design and development of software by providing a presentation of the domain that imitates the real world in terms of interacting objects and their properties. Programming languages evolved from assembler to high-level languages supporting object orientation and thereby enabled efficient development of complex applications with high quality. Today's requirements put into spreadsheets require a similar evolution in the spreadsheet paradigm. On the other hand, model-driven development evolved into an established development paradigm bridging the gap between business and IT by using models. Abstraction provides an easy understanding leading to less redundant models that are focused on the problem space. Working with a clear comprehensible concept of the business domain speeds up the design and development of software, which is further supported by using partial automation, e. g. generation of software artifacts.

A major benefit of using ClassSheets is that they can be handled by machines and therefore provide spreadsheet applications (e. g. Microsoft Excel) with more information about the spreadsheet in use. Thus, the creation and use of spreadsheets is less prone to errors since the spreadsheet application constrains possible spreadsheet updates according to the underlying ClassSheet. Figure 5 illustrates the current approach of developing spreadsheets on the left and the ClassSheets approach on the right side, where the generated spreadsheet is typed over the corresponding ClassSheet. In contrast to the current approach, the user does not develop low-levelspreadsheets, but creates a ClassSheet (i. e.
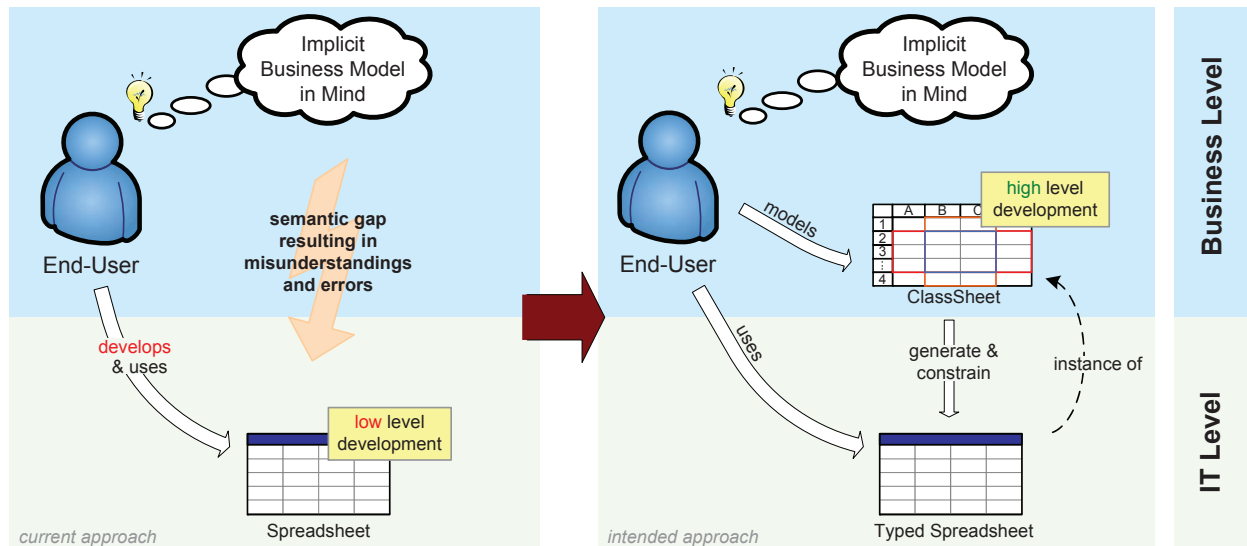
Figure 5: The Current and Intended Approach to Spreadsheet Development

business model) that reflects allnecessary business entities. This information is provided to the spreadsheetapplication to generate spreadsheets and avoid the mentioned errors, such asmissed formula updates.

Let us investigate ClassSheets by example. Figure 6 shows the ClassSheet for the budget calculation spreadsheet. ClassSheets look like spreadsheets with some extra functionallity, e. g. qualified references and specification of repetitions. Bold bolders indicate class boundaries, e. g. the whole ClassSheet consists of one embracing class (**A1:F5**) containing several other classes. Attribute definitions like year = 2010 assign a name and a default value to a cell.

4

	A	B	C	D	E	...	F
1	**Budget**		Year				
2			year = 2010				
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 6: The Budget ClassSheet

Each attribute belongs to one class and may be accessed by its quantified name, e. g. **Year**.year. The budget ClassSheet defines a (blue) association class (cells **C3**:**E4**), associating the two (red and orange) bracketing classes **Category** (rows **3** and **4**) and **Year** (columns **C**, **D**, and **E**). The association class contains entries for quantity, cost, and total per year and category. The two aggregated total values for a summation over years and categories are defined in the corresponding bracketing classes. The overall total is defined in an enclosing (black) bracket class **Budget**. The budget ClassSheet allows the expression of an arbitrary number of categories and years, which is denoted using the three dots in the axis description. The dots refer to all preceding rows/cols that are not divided by bars, i. e. for every year three columns are repeated (**C**, **D**, and **E**) while for every category, only row **4** is repeated. In Section 2.1, we discuss ClassSheets in more detail. Figure 7 shows an instance of the budget ClassSheet indicating its instance evolution, i. e. new data is
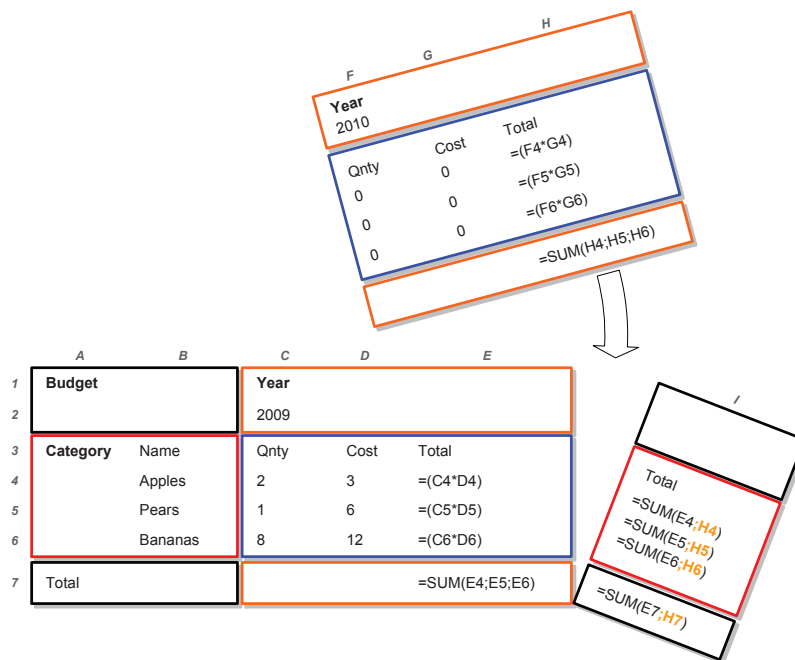
Figure 7: Adding a new Year to the Budget Sheet

inserted while the structure of composed classes remains. In the example the user decides to insert a new year (i. e. an instance of the **Year** class) which is supported by the spreadsheet application by inserting the three corresponding columns and the initial values. The example also demonstrates the automatic update of all involved formulas (marked yellow in the last column). Moving the user's decision (i. e. inserting a bunch of new columns) to a higher level (i. e. inserting a new instance of the class **Year**) does not only simplify working with spreadsheets, but also provide the spreadsheet application with a better understanding of the user's intention and thus yields a lower error rate due to the enabled automation. Therefore, ClassSheets solve the described problem of inserting errors in the process of instance evolution.

Figure 8 shows the life cycle of a spreadsheet using ClassSheets. First, a ClassSheet is created using a supporting

Figure 8: Life cycle of a Spreadsheet using the ClassSheet Approach

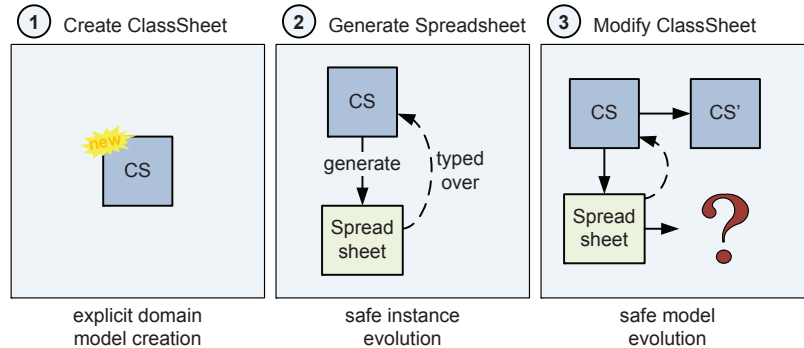tool. The use of ClassSheets at this stage simplifies the design of spreadsheets since well-known techniques from object-orientation can be used. For example, cells can be grouped to classes and formulas can refer to specific cells by using meaningful names. The ClassSheet approach comes with a rule system to identify correct ClassSheets, such that illegal ClassSheets cannot be created [9]. The second step in the life cycle is the generation of an empty ClassSheet instance (i. e. a spreadsheet) and its evolution. The ClassSheet tool suite comes with an Microsoft Excel plug-in that support instance evolution regarding the corresponding ClassSheet. The third step comprises the change of a ClassSheet in the presence of one or more corresponding instances. Those model changes must be replayed on all instances. Currently, there is no support for systematic model evolution in the spreadsheet paradigm.

This article presents a model evolution method that is based on ClassSheets and therefore is coined **ClassSheet evolution**. ClassSheet evolution is a special case of model evolution and comparable to schema evolution known from databases. However, the difficulty of ClassSheet Evolution is that the spreadsheet's spatial layout has to be considered. This is usually not the case for database schema evolution or other types of model evolution.

The core problem of model evolution is the need to replay the changes on existing instances of the model. For instance, changing a XML schema definition requires the change of all XML files that are described by the schema accordingly. The same applies to the paradigm of databases, where a database schema update must be reflected on every database build upon this schema. Many solutions for this problem had been proposed, including model-to-model transformations and graph grammar-based solutions [10, 11, 12]. However, for the spreadsheet paradigm, this problem is yet unsolved. Figure 9 describes the problem for ClassSheets in more detail. Whenever the user updates an existing ClassSheet, the problem of how to propagate this update to existing ClassSheet instances (i. e. spreadsheets) occurs. Manually updating the ClassSheet instances is definitely not advisable since this task is very prone to errors. Thus, an approach to automatically propagate ClassSheet updates to spreadsheets updates has to be developed.

The rest of this article is structured as follows. In Section 2, ClassSheets will be discussed in more detail, providing the reader with a detailed example on ClassSheet evolution in Section 2.2. In Section 3 we describe the implementation of the ClassSheet propagation approach. Therefore, we formally describe the used data structures and propagation rules. After discussing related work in Section 4, we conclude this article in Section 5 giving a short summary and pointing at some interesting future work.

2. ClassSheets

2.1. Basics

ClassSheets are two-dimensional grids that consist of classes, attributes, and labels. Attributes may have a concrete value or describe a formula (cf. Figure 6). The underlying business model of a ClassSheet (i. e. its conceptual information) can be expressed using UML class diagrams or similar representations. Figure 10 shows a UML class

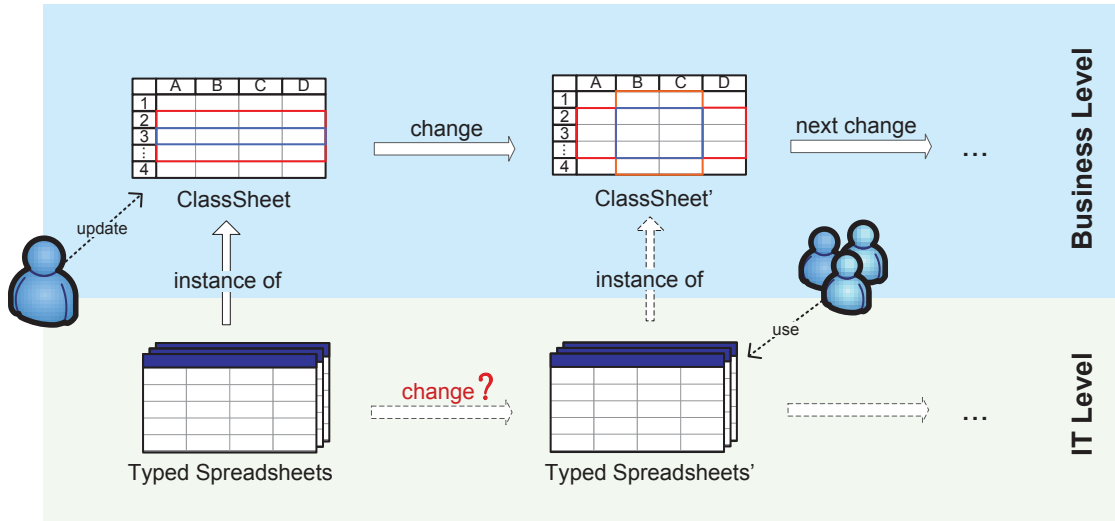Figure 9: Evolution of ClassSheets

diagram that expresses the business model of the budget ClassSheet. We use this representation to show the parallelism of ClassSheets and object-oriented design. Furthermore, we will use common terms that are used together with UML class diagrams, such as *association class*. A UML representation may be derived from every ClassSheet by
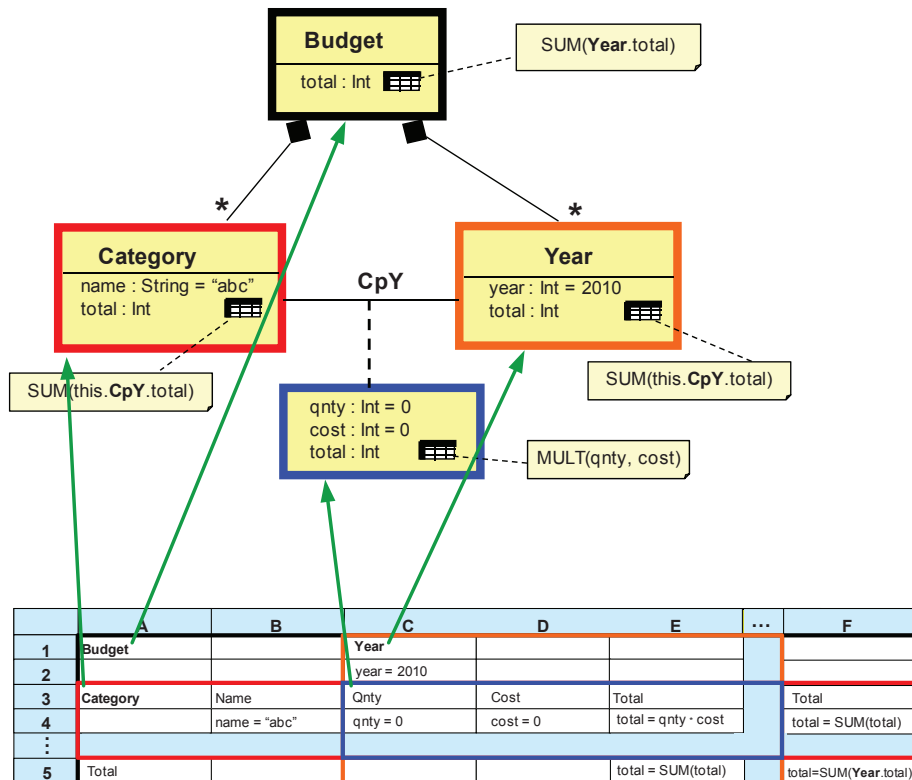

Figure 10: The Budget class Diagram

forgetting all layout information. The figure shows how the tabular structure of this ClassSheet is directly reflected by an association class **CpY** between the two spanning classes **Category** and **Year** that act as components of the root class **Budget**. Attributes are directly transferable to the UML diagram with labels being ignored. 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.

The use of unique names for classes and attributes in ClassSheets makes it possible to reference attributes by simply writing ClassName.AttributeName (see the SUM formula in Figure 10 for an example). Cell references such as SUM(**A1:D6**) known by classical spreadsheet applications are not used anymore. The distinction between relative and absolute cell references (i. e. **A1**,**A1**) is not needed since the correct update of references can be inferred from the relative locations of the referencing and referenced cells in the ClassSheet model [13]. Colored borders depict the different classes within a ClassSheet. Parts of classes may be spread over header and footer entries in spreadsheets, which results in a bracket-like structure indicated by a square-bracket-like notation of (open) class rectangles. For example, in Figure 6, the red class **Year** (columns **C**, **D**, and **E**, rows **1**, **2**, and **5**) is split into a header and a footer part that surround the blue association class. Repeatable rows or columns are denoted using vertical or horizontal dots in the axes-labelings. Finally, labels are used to annotate the representation of data (e. g. name, qnty, cost etc.). The budget ClassSheet (Figure 6) contains two repeatable classes, namely **Category** and **Year**.

The repetition applies to all rows or columns that are located before the dots and that are not separated by a bar. In our example, columns **C**, **D**, and **E** may be repeated horizontally and row **4** may be repeated vertically. Since the two classes **Category** and **Year** are two-dimensionally composed, an association class is necessary (blue framed in Figure 6 and named **CpY** in Figure 10). The association class grows with both the **Category** class and the **Year** class. An instance of this ClassSheet allows the insertion of an arbitrary number of categories, as well as quantity and cost values for each inserted year. The spreadsheet computes three types of totals, namely the total amount of money for each category and all years, for each year and all categories, and for all years and all categories. Figure 11 shows an example instance of this ClassSheet. Note that ClassSheet instances that are represented in a spreadsheet application like Microsoft Excel still make use of cell references. However, these cell references are continously inspected to match the qualified references specified in the ClassSheet.

Budget		Year 2009			Year 2010			
Category	Name	Qnty	Cost	Total	Qnty	Cost	Total	Total
	Apples	0	0	=(C4*D4)	0	0	=(F4*G4)	=SUM(E4;H4)
	Pears	0	0	=(C5*D5)	0	0	=(F5*G5)	=SUM(E5;H5)
	Bananas	0	0	=(C6*D6)	0	0	=(F6*G6)	=SUM(E6;H6)
Total				=SUM(E4;E5;E6)			=SUM(H4;H5;H6)	=SUM(E7;H7)

Figure 11: An instance of the Budget ClassSheet

Class repetitions are called class instances as known from the oo-paradigm. Note that some classes are repeatable while others are not. Non-repeatable classes correspond to singleton classes while repeatable classes may have an arbitrary number of instances. The same applies to attributes. Attributes that are located outside of a repetition group can be compared to static class attributes. ClassSheets do not distinguish between static class attributes and attributes of a singleton class.

ClassSheets can be designed using a tool described in [14]. The tool allows the export of ClassSheets to an intermediate language that is used by an Excel plug-in to constrain the work with spreadsheets. Further information about the tool support is given in Section 3.

2.2. Evolution

Changing requirements and thus adaptation of developed applications is common in the software engineering business. However, in contrast to most other development paradigms, spreadsheets require updates on the data and updates on the application to take place in the same environment—the two-dimensional grid. Using ClassSheets, these two different types of evolution are strictly distinguished. Updates of data, which we call instance evolution, take place inside the spreadsheet application and is solved by the presented ClassSheet approach [15, 9]. Updates of

	A	B	C	D	E	F	...	G
1	**Budget**		Year					
2			year = 2009					
3	**Category**	Name	Qnty	Cost	Rate	Total		Total
4		name = "abc"	qnty = 0	cost = 0	rate = 0	total=qnty·cost*rate		total = SUM(total)
⋮								
5	Total					total = SUM(total)		total = SUM(**Year**.total)

Figure 12: The Budget ClassSheet with Rate added

the conceptual model behind a spreadsheet, called model evolution, have to be performed on the ClassSheet level such that these updates are propagated to existing ClassSheet instances. This is the target of this article.

For example, let us consider the addition of an attribute that contains an exchange rate to the association class from Figure 6. The resulting ClassSheet is shown in Figure 12 with the changes colored yellow. We set a default value 0 for the attribute rate that will initially be set for each association class instance.

To propagate this update to the instance spreadsheet shown in Figure 11, we have to add the new column to each year entry and update all formulas that compute a category total. Figure 13 shows a ClassSheet instance with all necessary changes marked yellow. The parallels to the database paradigm are remarkable. For each association class instance a new data field with a default value is inserted. However, in contrast to ClassSheets, databases do not have to care about the location of fields. Also changing the association class affects the two associated classes as well, which is not the case for databases. In the concrete case of our example, the **Year** class representation had to be extended by an additional column.

Even for this small example, this task performed manually is quite annoying and, above all, error-prone. A solution that overcomes this problem is discussed in this article and depicted in Figure 14. At (1) an existing ClassSheet

Budget		**Year** 2009				**Year** 2010					
Category	Name	Qnty	Cost	Rate	Total	Qnty	Cost	Rate	Total		Total
	Apples	0	0	0	=(C4*D4 *E4)	0	0	0	=(G4*H4 *I4)		=SUM(F4;J4)
	Pears	0	0	0	=(C5*D5 *E5)	0	0	0	=(G5*H5 *I5)		=SUM(F5;J5)
	Bananas	0	0	0	=(C6*D6 *E6)	0	0	0	=(G6*H6 *I6)		=SUM(F6;J6)
Total					=SUM(F4;F5;F6)				=SUM(J4;J5;J6)		=SUM(F7;J7)

Figure 13: An instance of the Budget ClassSheet with the column Rate added

constrains a spreadsheet. Regarding instance evolution, ClassSheets ensure that the spreadsheet is a proper instance of the ClassSheet at any time [15]. If a ClassSheet is changed (2) using update commands, these commands are transformed (3) to spreadsheet update commands to reflect the model change on instance level (4). The whole process yields a new spreadsheet version that is a proper instance of the changed ClassSheet while all semantic information (i.e. datasets) is preserved (5). These steps are performed for every single atomic ClassSheet update.

3. Formalization and Implementation

There are at least two approaches for propagating ClassSheet changes to spreadsheets. The first one identifies changes on ClassSheets and reconstructs these changes on the spreadsheets. This approach is difficult to realize since there are countless changes that could be applied to a ClassSheet and especially for a whole sequence of changes it is difficult to reconstruct the atomic updates.[1] Using this approach means to find rules for every possible change on

---

[1]In the following, we distinguish between *update* and *change*. While a change is a passive observation from comparing two ClassSheets, an update is the active action that has been applied to the ClassSheet to achieve a change.
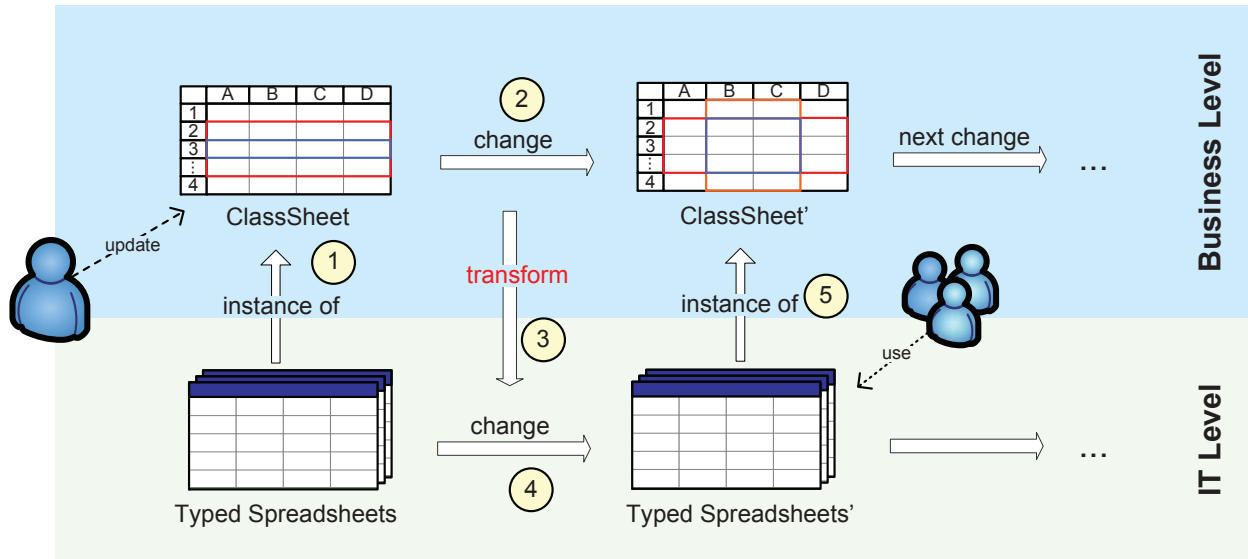
Figure 14: Evolution of ClassSheet - The Approach

a ClassSheet and transform those changes to updates on the corresponding spreadsheets. When demanding semantic-preserving transformations, or reversible transformations respectively, further complications arise. The second approach to tackle the problem of propagating model changes to spreadsheets is by propagating the update commands instead of the resulting change. For this approach, an update command language for ClassSheets has to be defined, along with transformation rules to transform update commands to updates on spreadsheets. In this article we will describe the latter approach of transforming update commands.

To express the propagation of ClassSheet updates to spreadsheets, we introduce a specialized formal graph representation of ClassSheets named ClassSheet graph (CS graph). These CS graphs allow a convenient description and computation of propagation details. Figure 15 shows the different levels classified using the MDA levels PIM, PSM, and code. For each visual language used during spreadsheet development, a formal representation exists. Spreadsheets are formalized using sets of triples consisting of cells coordinates and cell content. The formalization used for ClassSheets are CS graphs. Class diagrams that hold only the conceptual information of ClassSheets may be formalized using class graphs. Technically, CS graphs are class graphs annotated with spatial information. Thus, a ClassSheet can be represented as a class diagram at any time giving a condensed view of the ClassSheet's business model. Changes on class graphs are transformed to changes on CS graphs using defaults for the layout options. Changes on the CS graphs are automatically transformed to changes on spreadsheets. In this article, we will describe the red framed area in Figure 15. The integration of class graphs and diagrams into an MDA-like approach is left to future work.

The propagation of ClassSheet updates is implemented within three main components shown in Figure 16. The rightmost component *Claos* is an editor that allows the visual modelling of ClassSheets [15]. The creation of ClassSheets is enabled by simple operations such as adding a new class before, behind or inside the active class, deleting a class or renaming a class. Each of the classes can be equipped with attributes. Each class has a defined height and width given with the number of spanned cells. Attributes can be arbitrarily placed within these cells. Finally, the editor comprises a rule system that prevents the user from modelling illegal ClassSheets. The Claos editor with the budget ClassSheet opened is shown in Figure 17.

The leftmost component in Figure 16 is given by a VBA plug-in for Microsoft Excel. The VBA plug-in is responsible for representing a ClassSheet instance within a spreadsheet, informing the server about the instance evolution of the spreadsheet, and finally receiving and performing commands from the server that are necessary for model evolution.

The last component in Figure 16, the server, holds the current state of the ClassSheet and its instance and computes instance evolution as well as model evolution commands. The server is an extension to Gencel, a spreadsheet generator
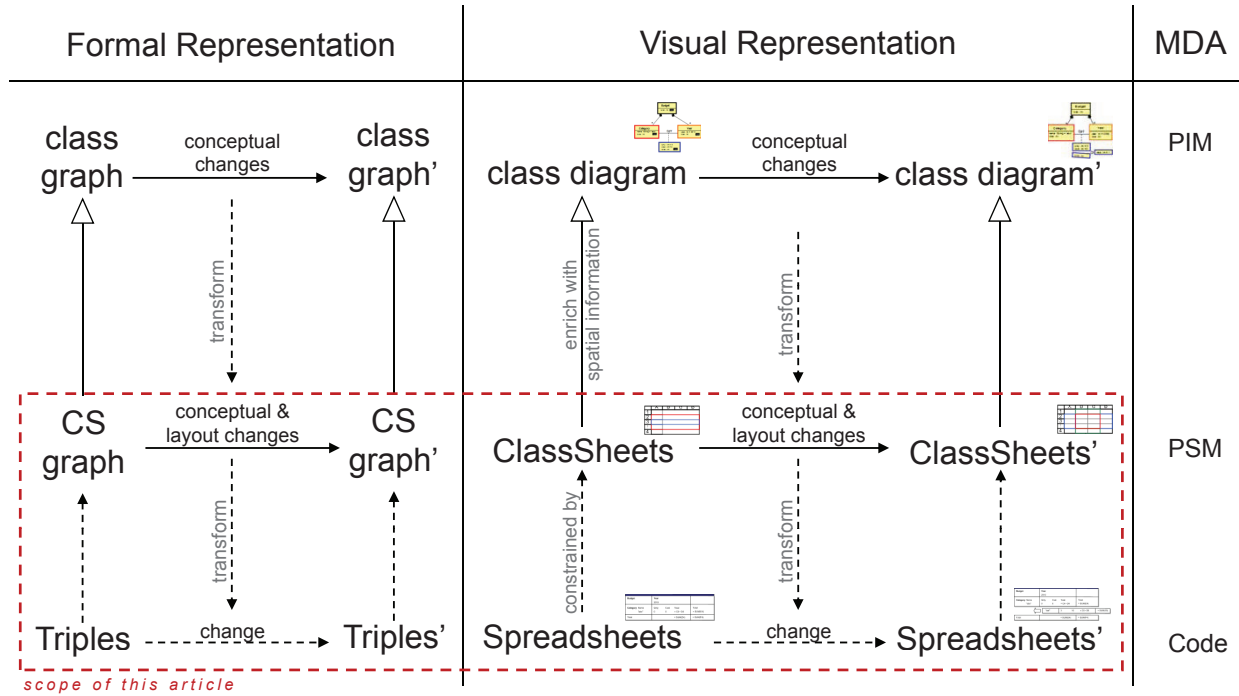
Figure 15: Levels of Spreadsheet Application Development
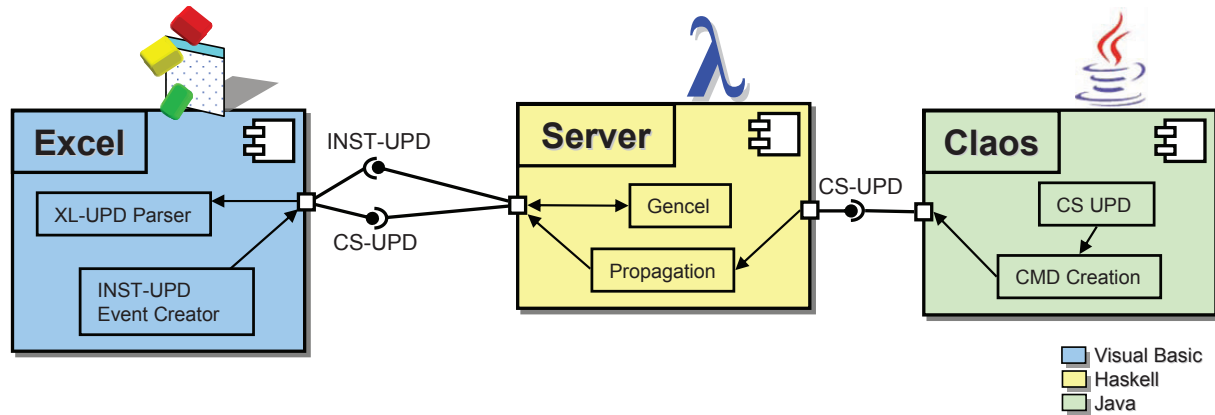
Figure 16: Integrated ClassSheet Tool Suite

that is described in [13]. Gencel employs a visual template language, called VITSL, to specify spreadsheet models. It provides the ability to specify repeatable areas in spreadsheets which can be expanded in Excel automatically. The ClassSheets specific extension of Gencel introduces the concept of object-orientation and the ability of update propagation.

All three components together form the Integrated ClassSheet Tool Suite (ICTS). While the server holds the both the ClassSheet and its instance, the Claos editor is used as a view to the ClassSheet, while Excel represents the view on one particular ClassSheet instance. The ICTS is created as a client server architecture, where the server acts as the heart of the system. The connection to the Claos editor is one-directional, whereas the connection to the Excel side is bi-directional. None of the provided interfaces is mandatory for the server to run, neither does Claos need the server

11

Figure 17: The Claos Editor

to operate. This means that the Claos editor is usable as a stand-alone application to create and save ClassSheets[2]. Which interfaces have to be served depends on the tasks to be performed. If Claos is not connected, no propagation commands will be created, but instance evolution is still supported. The only mandatory interface from the Excel side is the one named INST-UPD since otherwise Excel would not be able to check the spreadsheet for compliance with the ClassSheet after new data has been inserted.
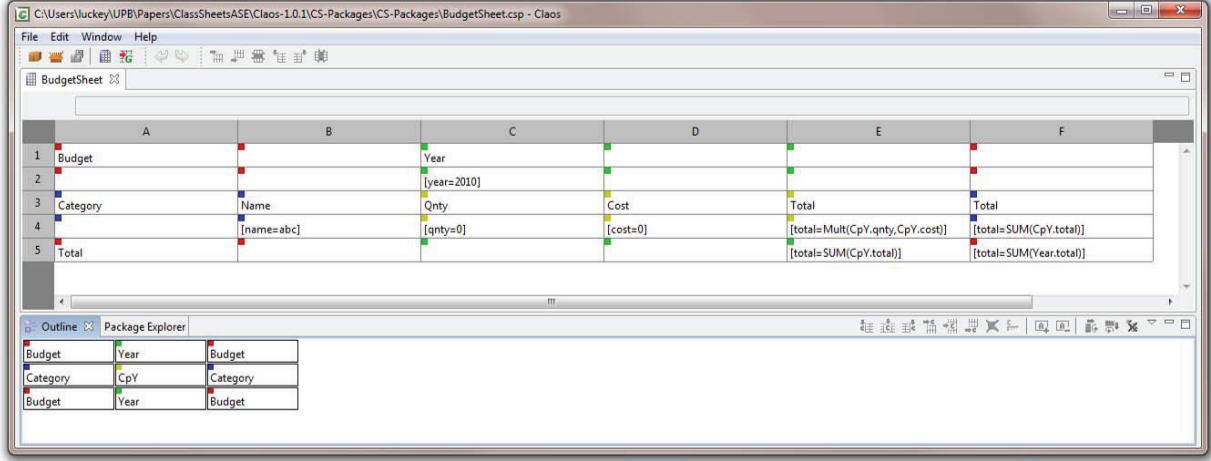
In the following we will describe the formal model, the CS graph, that is created and changed with Claos and hold by the server. After a brief example of instance evolution, which is described in detail in [9, 15], we focus on the computation of propagation details, i. e. how a ClassSheet update is reflected on its instances.

3.1. A formal representation of ClassSheets for Update Propagation

During our research we found out that while the ClassSheet representation is well-chosen concerning readability and thus user acceptance, it helps to formally define them using graph structures when it comes to propagation of updates to spreadsheets. Employing a graph structure allows us to easily represent and cluster the object-orientation means (i.e. classes and contained attributes) within the data structure and in turn helps to define simple propagation rules. Along with the formal representation of ClassSheets, i. e. CS graph, we employed semantics for the update language to formally describe transformation and propagation rules. The CS graphs, used in the server component to describe a ClassSheet, is defined as follows.

$$
\begin{array}{rcl}
S & = & (V, E, L_V) \\
V & \subseteq & \textit{Name} \\
E & \in & V \times V \to L_E
\end{array}
$$

Figure 18: Definition of CS graphs

As shown in Figure 18 a CS graph is a three tuple consisting of a set of nodes V, a set of edges E and a labeling function L_V. The set of nodes V matches the set of classes from the ClassSheet. The edges, given by the mapping E, describe the spatial relations between classes where a label $l_e \in L_E$ gives the type of an edge (cf. Figure 19(a)). The aggregation type **Comp** marks one of the edge's adjacent nodes as being contained in the other. The direction specifies whether the class is inserted horizontally or vertically into its parent class. Figure 19(b) shows the definition

---

[2]Though, this would not be able to propagate changes since they are not yet saved to be propagated in transactions.

of an edge between the two classes **Budget** and **Category** (**Bud** and **Cat** for short), while the left column defines the **Category** class to be inserted horizontally into the **Budget** class, while the right column defines the vertical insertion. The given offset *n* defines the position of one class in the other. *n* ranges over $\mathbb{N}$. For example, in Figure 19(b), the

$$l_e \in \mathsf{L_E} \quad ::= \quad \mathbf{Comp}^{\Rightarrow} \mathsf{O} \mid \mathbf{Comp}^{\Downarrow} \mathsf{O}$$
$$\mid \quad \mathbf{Succ}^{\Rightarrow} \mid \mathbf{Succ}^{\Downarrow}$$
$$\mid \quad \mathbf{ISucc} \mid \mathbf{Assoc} \; \mathsf{V}$$

(a) Grammar

(**Bud**, **Cat**, **Comp**$^{\Rightarrow}$ 2) | (**Bud**, **Cat**, **Comp**$^{\Downarrow}$ 2)

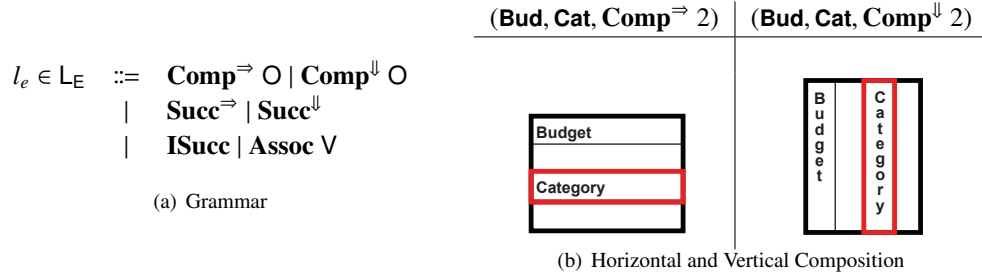(b) Horizontal and Vertical Composition

Figure 19: Class Labels

Category class is located at position 3 inside the **Budget** class. Edges of type **ISucc** (inner successor) specify the order of inner classes. This is useful whenever two inner classes are located at the same offset of their parent class. Classes that are located next to each other have to be connected by the successor edge **Succ**. The direction specifies whether a class is located below or beneath another class. Two classes are connected by an association class $v \in \mathsf{V}$ using the **Assoc** edge.

Finally, the labeling function $\mathsf{L_V}$, shown in Figure 20, assigns to each class its attributes and labels, as well as a size and information about the repetition of classes. We let the variable *ct* range over the set of a class's content, i.e. its attributes and labels.

$$
\begin{aligned}
l_v \in \mathsf{L_V} \quad &\subseteq \quad \mathsf{V} \to 2^{\mathit{Content}} \times \mathit{Type} \\
\mathit{ct} \subseteq \mathit{Content} \quad &= \quad \mathbb{N} \times \mathbb{N} \to (\mathit{Name} \to \mathit{Fml}) \;\cup\; \mathbb{N} \times \mathbb{N} \to \mathit{Label} \\
\mathit{Type} \quad &::= \quad \mathbf{Class}\; \mathit{Size}\; \mathit{Rep} \mid \mathbf{Assoc} \\
\mathit{Size} \quad &= \quad \mathbb{N} \times \mathbb{N} \\
\mathit{Rep} \quad &= \quad \mathbb{N} \times \mathbb{N}
\end{aligned}
$$

Figure 20: Labeling Function of CS graphs

The set *Content* contains named formulas together with labels. Labels are quoted strings. Attributes and labels are assigned to coordinates that must range within the class's size. The type contains the size and information about the repetition. The first component of *Rep* specifies the first row or column that can be repeated. This information allows us to have regions in classes that are not repeatable, i.e. headers. For example, in Figure 6, the upper row of the **Category** class will not be repeated with each category entry. Thus the first row of repetition would be 2. The second component of *Rep* describes how often the class has already been repeated in actual instance spreadsheets. Note that the size and the information about the repetition are not specified for association classes since this information is obtained from the associated classes.

Using the introduced definitions, we can represent the budget ClassSheet from Figure 17 and Figure 6 as $\mathsf{S} \; =$

13

(V, E, L_V) where

$$V = \{\mathbf{Bud}, \mathbf{Cat}, \mathbf{Year}, \mathbf{CpY}\}$$

$$E = \{(\mathbf{Bud}, \mathbf{Cat}, \mathbf{Comp}^{\Rightarrow} 2), (\mathbf{Bud}, \mathbf{Year}, \mathbf{Comp}^{\Downarrow} 2), (\mathbf{Cat}, \mathbf{Year}, \mathbf{Assoc}\ \mathbf{CpY})\}$$

$$L_V = \{(\mathbf{Bud}, \{(1, 1, \texttt{"Budget"}), (1, 3, \texttt{"Total"}), (3, 3, \text{total} = \text{SUM}\ (\mathbf{Year}.\text{total})\}, \mathbf{Class}\ (3, 3)\ (0, 0)),$$
$$(\mathbf{Cat}, \{(1, 1, \texttt{"Category"}), (2, 1, \texttt{"Name"}), (3, 1, \texttt{"Total"}),$$
$$(2, 2, \text{name} = \texttt{"abc"}), (3, 2, \text{total} = \text{SUM}\ (\mathbf{CpY}.\text{total})\}, \mathbf{Class}\ (3, 2)\ (2, 0)),$$
$$(\mathbf{Year}, \{(1, 1, \texttt{"Year"}), (1, 2, \text{year} = 2007), (3, 3, \text{total} = \text{SUM}\ (\text{total}))\}, \mathbf{Class}\ (3, 3)\ (1, 0)),$$
$$(\mathbf{CpY}, \{(1, 1, \texttt{"Qnty"}), (2, 1, \texttt{"Cost"}), (3, 1, \texttt{"Total"}),$$
$$(1, 2, \text{qnty} = 0), (2, 2, \text{cost} = 0), (3, 2, \text{total} = \text{qnty} \times \text{cost})\}, \mathbf{Assoc})\}$$

The rough structure of our CS graph can be extracted from the set of nodes V and the set of edges E. The annotation function L_V contains all attributes and labels for the classes, or nodes respectively, in V. Leaving out all spatial information, i.e. coordinates, labels, orientations and offsets ($\mathbf{Comp}^{\Rightarrow} 2$), the graph structure describes the corresponding class graph. The ICTS server component uses this formal ClassSheet representation to store the ClassSheet and the current state of its instance.

3.1.1. Update Language

During the creation of a ClassSheet using Claos, the user makes use of several update commands. For instance, constructing the budget ClassSheet, the user starts with an empty ClassSheet and initially uses a command to add a class named **Budget**. Next the user would probably add an inner class **Category**, and so on. Formally, these update commands take an existing CS graph and return a new CS graph that was changed accordingly. Figure 21 illustrates an update command within Claos. In the following, we define the set of update commands, which are necessary to
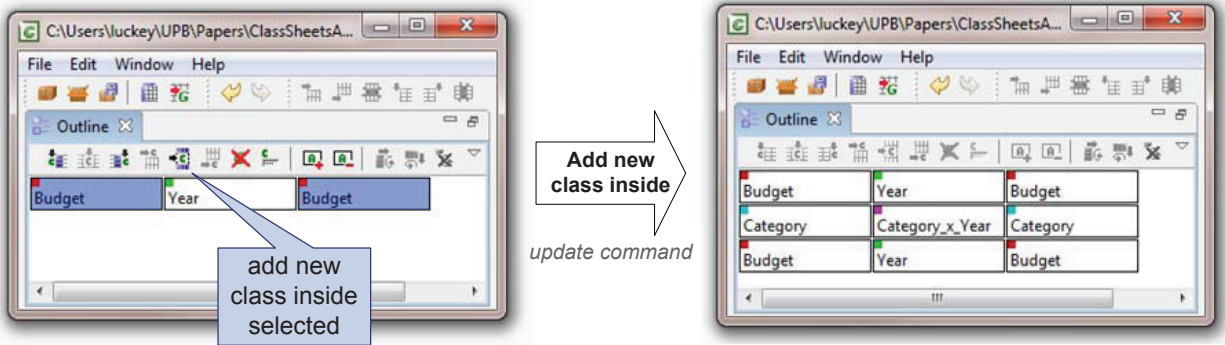

Figure 21: Update Command in Claos

create and update a ClassSheet.

In Figure 12, we altered spatial as well as conceptual information of our ClassSheet: adding a new column changed the spatial layout and adding or changing attributes influenced the conceptual information. While the class diagram contains only conceptual information, ClassSheets mix spatial and conceptual information as known from common spreadsheet applications (e.g. Microsoft Excel). Therefore, we can classify ClassSheet changes into conceptual and layout changes. Conceptual changes are all those changes that affect conceptual information captured in the class diagram. In contrast, layout changes never affect a spreadsheet's conceptual information, such as computations, but rearrange a spreadsheet's spatial layout, such as the location of formulas in the spreadsheet grid.

Structural changes on ClassSheets can be expressed by the update language shown in Figure 22. All these operations reflect common updates on class diagrams. We use the meta variable f to range over formulas. Formulas are defined over (qualified) attributes and standard calculation operations supported by the respective spreadsheet application.

$$
\begin{aligned}
u \in \textit{Upd} \quad &::= \quad \textbf{add } v \\
&| \quad \textbf{add } v.a = f \\
&| \quad \textbf{add } v \Diamond{\rightarrow}_k v \\
&| \quad \textbf{add } v{\hookleftarrow}v{\hookrightarrow}v \\
&| \quad \textbf{del } v \\
&| \quad \textbf{del } v.a \\
&| \quad \textbf{del } v\Diamond{\rightarrow}v \\
&| \quad \textbf{del } v{\leftarrow}{\rightarrow}v \qquad (\textit{updates}) \\
k \in \textit{Card} \quad &::= \quad (i,i) \mid \star \qquad\quad (\textit{cardinality})
\end{aligned}
$$

Figure 22: CS graph Update Language (structural updates).

The operation **add** v adds a new class with given name to the CS graph. The second **add** operation adds the given attribute consisting of a name a and a formula f to the class v. There are two kinds of edges in our CS graph representation that can be added with two different commands. The **add** $c_1 \Diamond{\rightarrow}_k c_2$ command adds a compositional relation between two given classes. This means that the class v_1 will be parent of class v_2. The additional argument k allows the specification of a cardinality that describes the possible number of repetitions of information (rows or columns) in class v_2. In a spreadsheet instance this repetition could be limited to a few rows or columns of the class, i.e. not the whole class has to be repeated. This will be shown in detail in Section 3.1. On the other hand, the command **add** $c_1 {\hookleftarrow}c{\hookrightarrow}c_2$ sets the class v to be the association class of the classes v_1 and v_2. The last four operations delete classes (**del** c), attributes (**del** $c.a$), compositions (**del** $c_1 \Diamond{\rightarrow}c_2$) and associations (**del** $c_1 {\leftarrow}{\rightarrow}c_2$). A class is given by its name, the attribute is given by its qualified name (class and attribute name), compositions are given by their adjacent classes and associations are given by the names of the associated classes.

Of course, the same update operations can be applied to class graphs in order to synchronize the models on the different levels. However, since these operations applied to CS graphs have to define spatial aspects of classes and attributes, they make some initial assumptions, such as where to place a new attribute in the tabular grid. To adjust these assumptions and to maintain the grid, further update operations shown in Figure 23 can be applied to CS graphs. To change a CS graph's orientation, the **toggleOri** operation can be used, which toggles the whole graph's orientation[3],

$$
\begin{aligned}
u \in \textit{Upd} \quad &::= \quad \textbf{toggleOri} \mid \textbf{move } v\ d \mid \textbf{move } v\ (i,i)\ d \\
&| \quad \textbf{add } v\ (i,i)\ l \mid \textbf{del } v\ (i,i) \\
&| \quad \textbf{enlarge } v\ d \mid \textbf{shrink } v\ d \mid \textbf{rep}^{\uparrow} v \mid \textbf{rep}^{\downarrow} v \quad (\textit{updates}) \\
d \in \textit{Direction} \quad &::= \quad \{\rightarrow, \downarrow, \leftarrow, \uparrow\} \qquad\qquad\qquad\qquad (\textit{direction})
\end{aligned}
$$

Figure 23: CS graph Update Language (layout updates).

for example to change a horizontal to a vertical orientation. The operations **move** $v\ d$ and **move** $v\ (i,i)\ d$ move classes and cell content, respectively, into the given direction d. Cell content is referred to by coordinates related to the class's upper left cell. The operations **add** $v\ (i,i)\ l$ and **del** $v\ (i,i)$ allow the addition and deletion of labels, while the operations **add** $v.a$ and **del** $v.a$ allow the addition and deletion of attributes in a CS graph. Enlargement and shrinking of a class v can be achieved by applying the **enlarge** $v\ d$ and **shrink** $v\ d$ operations, where d is the direction in which to change the size. Finally, the operations **rep**$^{\uparrow} v$ and **rep**$^{\downarrow} v$ increase and decrease the first row or column of repetition in a repeatable class v. For example, increasing the first row of repetition to 2 does not repeat the first row on the class for every single class instances. The first row might be a header. If the first row or column of repetition is set to 0, the class may not be repeated at all (i.e. the class is a singleton class). Since a class cannot be repeatable in vertical and horizontal direction at the same time, one number suffices to represent the repetition. In the example from Figure 6 this number is set to 2 for the **Category** class and to 1 for the **Year** class. The **Budget** class cannot be repeated at all,

---

[3]mirroring at the diagonal axis

thus the first row or column of repetition is set to 0.

3.1.2. Update Semantics

In the last section we have shown a formal data structure for ClassSheets (i.e. CS graphs) and an update language to change ClassSheets. Next we show how the update operations are implemented, by giving some insights into their semantic function. Since describing the semantics for all update commands would be out of proportion, we will concentrate on the semantics of the four operations used to perform the update from the initial example in Section 1. These operations include enlarging a class, moving cells and adding an attribute and a label. For the sake of readability we use Haskell style for function definitions and applications. In Haskell, arguments are not given in brackets and comma separated but are written directly behind the function name separated by white spaces.

First of all, we address the problem of enlarging a class. Enlarging of one class may involve other classes. For example, enlarging a class that is part of a horizontal composition at its bottom edge requires all other classes to be enlarged at their bottom edge since the resulting ClassSheet must be of rectangular shape. Thus, we introduce a function `sibl` that computes the set of all siblings of a class in a ClassSheet.

$$\texttt{sibl} :: \mathcal{S} \rightarrow 2^{\mathsf{V}} \rightarrow \{\leftarrow, \rightarrow, \uparrow, \downarrow\} \rightarrow 2^{\mathsf{V}}$$

The function takes the CS graph as first argument and a singleton set containing the class we are searching the siblings for as second argument. The passed class is returned in a singleton set, if it has an association class since then there are no siblings per definition (see Tilings in [15]). The third argument specifies the direction for the search.[4] For example, if we want to find all siblings that are horizontally composed with the given class, we pass $\rightarrow$ as an argument.

$$
\begin{aligned}
[\![\textbf{enlarge}\ c\ d]\!]^{U}\ (\mathsf{V}, \mathsf{E}, \mathsf{L_V}) \quad = \quad & (\mathsf{V}, \mathsf{E}, (\mathsf{L_V} \setminus \{\mathsf{L_V}(c') \mid c' \in s\}) \\
& \cup \{\texttt{enlarge}\ d\ \mathsf{L_V}(c') \mid c' \in s\}) \\
& \textbf{where}\ s = \texttt{sibl}\ (\mathsf{V}, \mathsf{E}, \mathsf{L_V})\ \{c\}\ d
\end{aligned}
$$

Basically, the semantic definition calls the `enlarge` function for each of the passed class's siblings. The real work is performed by the `enlarge` function, defined as follows.

$$
\begin{aligned}
\texttt{enlarge}\ d\ (c, ct, \textbf{Class}\ si\ r) \quad = \quad & (c, ct', \textbf{Class}\ si'\ r) \\
& \textbf{where}\ si' = si + |d| \\
& \quad ct' = \begin{cases} \texttt{shiftCont}\ |d|\ ct & \textbf{if}\ d \in \{\uparrow, \leftarrow\} \\ ct & \textbf{otherwise} \end{cases}
\end{aligned}
$$

Since enlargement of classes is allowed at all four edges (left, right, up, and down), we have to shift content in some cases. For example, enlarging a class at its top edge is done by enlarging it at its bottom edge and shifting its content down. To shift content into a given direction, the function `shiftCont` is used.

$$\texttt{shiftCont}\ (i, j)\ ct = \{((x + i, y + j), a) \mid ((x, y), a) \in ct\}$$

Let us consider as a simple example the following CS graph definition.

$$
\begin{aligned}
\mathsf{S} \quad &= \quad (\mathsf{V}, \mathsf{E}, \mathsf{L_V}) \\
\mathsf{V} \quad &= \quad \{\textbf{Bud}, \textbf{Cat}\} \\
\mathsf{E} \quad &= \quad \{(\textbf{Bud}, \textbf{Cat}, \textbf{Comp}^{\Rightarrow}\ 2)\} \\
\mathsf{L_V} \quad &= \quad \{(\textbf{Bud}, \{(1, 1, \texttt{"Budget"}), (1, 3, \texttt{"Total"})\}, \textbf{Class}\ (3, 3)\ (0, 0)), \\
& \qquad (\textbf{Cat}, \{(1, 1, \texttt{"Category"}), (2, 1, \texttt{"Name"}), (3, 1, \texttt{"Total"}), \textbf{Class}\ (3, 3)\ (2, 0))\}
\end{aligned}
$$

---

[4] The defined function distinguishes between horizontal and vertical search. Thus the arguments $\rightarrow$ and $\leftarrow$ are handled as synonyms. Same applies to vertical directions.

$\downarrow$	$(0, 1)$	$\uparrow$	$(0, -1)$
$\rightarrow$	$(1, 0)$	$\leftarrow$	$(-1, 0)$
$|d|$	positive direction	$-|d|$	negative direction
$i + d$	adds direction d to tuple i: $(i_1 + d_1, i_2 + d_2)$		

Basically, a direction is a tuple, where the first value describes the horizontal direction and the second value describes the vertical direction. This notation is highly used in mathematics, where these tuples are called vectors. In particular, the + operation is the vector addition.

Table 1: Directions as Tuples

We want to enlarge the **Cat** class at its top edge. The semantic rules set the class's size to $(3, 4)$ and add 1 to the vertical coordinates of all attributes by calling `shiftCont` $(0, 1)$ ct.

Moving content into the specified direction basically changes the content with the neighbor cell. The `swapCells` function takes the content and a direction to move the content into and returns the updated content. In the following definition, directions are implicitly converted to tuples as described in Table 1. Now we define the `swapCells` function as follows[5]:

$$\text{swapCells } ct\ (i, j)\ d \quad = \quad ct \setminus \{ct(i, j), ct((i, j) + d)\}$$
$$\cup \{(i, j, ct((i, j) + d)), ((i, j) + d, ct(i, j))\}$$

Finally, the following definitions show the semantics of adding an attribute and a label while `nextFreeCell` returns the coordinates of a class content's next empty cell.

$$[\![\textbf{add } c\ (i, j)\ l]\!]^U\ (\mathsf{V}, \mathsf{E}, \mathsf{L_V}) \quad = \quad (\mathsf{V}, \mathsf{E}, (\mathsf{L_V} \setminus \{\mathsf{L_V}(c)\}) \cup \{(c, (ct \setminus \{ct(i, j)\}) \cup \{((i, j), l)\}, t)\})$$
$$\textbf{where } (c, ct, t) = \mathsf{L_V}(c)$$

$$[\![\textbf{add } c.a = f]\!]^U\ (\mathsf{V}, \mathsf{E}, \mathsf{L_V}) \quad = \quad (\mathsf{V}, \mathsf{E}, (\mathsf{L_V} \setminus \{\mathsf{L_V}(c)\}) \cup \{(c, ct \cup \{(pos, a, f)\}, t))\})$$
$$\textbf{where } (c, ct, t) = \mathsf{L_V}(c)$$
$$pos = \texttt{nextFreeCell } ct$$

Due to the chosen data structure and the denotational style of defining the semantic function's domain, other update commands are similar straightforward to define. In the following section, we will give a short example of the CS graph data structure and the defined semantics for update commands.

To experience how the semantic function for the update commands operates, let us enlarge the year class at its right edge. The denotational semantics updating our CS graph are given as follows where $\mathsf{L_V}|_c$ denotes the tuple from set $\mathsf{L_V}$ with the first component matching c.

$$[\![\textbf{enlarge Year} \rightarrow]\!]^U\ (\mathsf{V}, \mathsf{E}, \mathsf{L_V}) \quad = \quad (\mathsf{V}, \mathsf{E}, (\mathsf{L_V} \setminus \{\mathsf{L_V}|_{c'} \mid c' \in \texttt{sibl S } \{\textbf{Year}\}\ (\rightarrow)\})$$
$$\cup\ \{\texttt{enlarge} \rightarrow \mathsf{L_V}|_{c'} \mid c' \in \texttt{sibl S } \{\textbf{Year}\}\ (\rightarrow)\})$$

Since **Year** has an associated class, the siblings function `sibl` returns {**Year**} per definition. The `enlarge` function adds 1 to the class's width.

$$[\![\textbf{enlarge Year} \rightarrow]\!]^U\ (\mathsf{V}, \mathsf{E}, \mathsf{L_V}) \quad = \quad (\mathsf{V}, \mathsf{E}, (\mathsf{L_V} \setminus \{(\textbf{Year}, ct, \textbf{Class } (3, 3)\ (2, 0))\})$$
$$\cup\ \{(\textbf{Year}, ct, \textbf{Class } (\textbf{4}, 3)\ (1, 0))\})$$

---

[5]We handle undefined function results ($\bot$) as values, which allows us to exchange content with an empty cell.

As shown in this example, we have all means to formally represent ClassSheets and express update operations applied to them. We employed Claos to enable the user to create and maintain ClassSheets and showed how the ClassSheets are formally represented and updated in the server. In the following two sections we will give a formal representation for spreadsheets and a corresponding update language to describe the transformation of update operations on the ClassSheet level to update operations on the spreadsheet level.

3.2. Formalizing Spreadsheet Updates

In our tool suite, there are two different scenarios in which a spreadsheet has to be changed. First, the user decides to perform an instance evolution, e. g. adding a new year or a new category. To this end, the Excel add-in makes some buttons available that enable the addition or deletion of class repetitions. For example, the user may select a cell from year 2010 and use a button to add another year righthand. This would result in adding three new columns with default values for every existing category and updating all affected formulas. The second scenario in which the spreadsheet might change is model evolution. Take our example from Section 1, where we added an attribute to the class **Year**. This update would result in adding a new column for each year and one default value for each existing category. Both scenarios require an update language for spreadsheets, which is briefly introduced in the following.

Spreadsheets are represented by a set of triples containing the horizontal and vertical coordinates and the formula: $T = \mathbb{N} \times \mathbb{N} \times F$. Note that the formulas are different from the ones used in CS graphs since they do not contain qualified references to attributes but coordinates.

In order to propagate model updates to spreadsheets, we need a simple spreadsheet update language that can be executed by the spreadsheet application. We call it "T-update" language where T stands for table. T-updates are used to change actual spreadsheets in an Excel-like application. The corresponding language is defined in Figure 24.

$$
\begin{array}{llll}
u \in \textit{T-Upd} & ::= & \mathbf{add}^R \ i \mid \mathbf{add}^C \ i \mid \mathbf{del}^R \ i \mid \mathbf{del}^C \ i & \\
& \mid & \mathbf{move}^R \ i^+ \ d \mid \mathbf{move}^C \ i^+ \ d & \\
& \mid & \mathbf{swap} \ ((i,i),(i,i)) \mid \mathbf{upd} \ (i,i) \ f & (\textit{updates}) \\
d \in \textit{Direction} & ::= & \{\rightarrow, \downarrow, \leftarrow, \uparrow\} & (\textit{direction})
\end{array}
$$

Figure 24: T Update Language.

The $\mathbf{add}^R$ and $\mathbf{add}^C$ commands insert rows and columns at the given row or column index. Existing rows or columns are shifted down or to the right. The operations $\mathbf{del}^R$ and $\mathbf{del}^C$ delete the given row or column. The produced hole is filled with the rows from below, respectively with the columns from the right. The commands $\mathbf{move}^R$ and $\mathbf{move}^C$ move the given list of columns or rows into the given direction. $\mathbf{swap}$ exchanges the given two cells' content, and $\mathbf{upd}$ sets the given cell to the given formula.

The update language's semantics are trivial since they correspond to the behavior known from common spreadsheet applications. For example, the semantics for the $\mathbf{add}^C \ i$ command are defined as follows.

$$
\begin{aligned}
[\![\mathbf{add}^C \ i]\!]^{XL} \ T \ = \ & \{(x, y, f) \mid (x, y, f) \in T, x < i\} \cup \\
& \{(i, y, \sqcup) \mid y = 1 \ .. \ \textit{LastUsedRow}\} \cup \\
& \{(x + 1, y, f) \mid (x, y, f) \in T, x \geq i\}
\end{aligned}
$$

$$
[\![\mathbf{upd} \ (i, j) \ f]\!]^{XL} \ T \ = \ \{(x, y, f) \mid (x, y, f) \in T, (x, y) \neq (i, j)\} \cup \{(i, j, f)\}
$$

Adding a new column into a spreadsheet means shifting all columns behind column $i - 1$ to the right and inserting new empty cells into the new column i. Consider the small spreadsheet shown in Figure 25. The spreadsheet is represented by the following formula.

$$
T \ = \ \{(1, 1, \texttt{"Test"}), (2, 1, \texttt{"Desc"}), (1, 2, 3), (2, 2, \texttt{"abc..."})\}
$$

Adding a column at position 2 with the command $\mathbf{add}^C$ 2 updates the spreadsheet as follows.

$$
\begin{aligned}
[\![\mathbf{add}^C \ 2]\!]^{XL} \ T \ = \ & \{(1, 1, \texttt{"Test"}), (1, 2, 3)\} \cup \{(2, 1, \sqcup), (2, 2, \sqcup)\} \\
& \{(3, 1, \texttt{"Desc"}), (3, 2, \texttt{"abc..."})\}
\end{aligned}
$$

18

	A	B
1	Test	Desc
2	3	abc…
3		
4		

Figure 25: Example Spreadsheet

3.3. Implementing Instance Evolution - A small example

Whenever the user has loaded a ClassSheet in Excel, an additional toolbar is made available. The buttons allow the insertion or deletion of class repetitions in the neighborhood of the selected cell. For instance, see Figure 26 where a new income item will be added below the selected cell. Figure 27 shows the corresponding sequence diagram. The
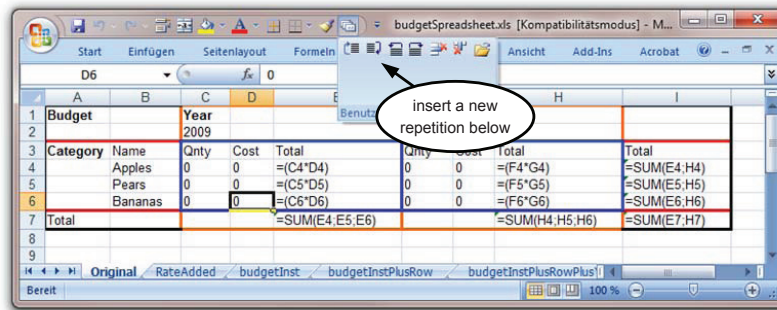

Figure 26: Inserting a new repetition of the **Category** class

initial event to insert a row is triggered by the button click. VBA translates this event into two commands. The first one updates the cursor position in the spreadsheet. The second one requests the insertion of one row below the actual cursor position. The server updates the ClassSheet and computes a command to insert the row into the spreadsheet. Also, commands to update several cells are created. The first inserts a value into the new added row which corresponds to the default value specified in the ClassSheet (see Figure 17), the remainder updates all cells that contain references since these may have changed due to inserting a new row.

3.4. Propagation of ClassSheet Changes to Spreadsheets

In the last sections, we showed how ClassSheets are created and persisted in the server. Further, we described a small update language for spreadsheets and showed how instance evolution is realized in our tool suite. Now the user shall be able to change the ClassSheet in use while the server shall propagate those changes to the ClassSheet instance. As mentioned before, this is achieved by translating the update commands for ClassSheets to update commands for spreadsheets. As an example consider an update on the budget ClassSheet, where we add a new row below row 4. The command is caught in Claos and sent to the server (see the sequence diagram in Figure 28). The server computes all propagation details and sends suitable update commands to Excel. The VBA add-in in Excel parses these update commands and adds several new rows to the ClassSheet instance (one for each existing category). In this section we will illustrate how an update on a ClassSheet propagates to a spreadsheet by enlarging the category class of our budget sheet, adding a formula and updating another one. To this end, we will concentrate on a few update commands. Other commands are handled analogously. Full details, including semantic definitions for the evaluation of formulas, can be found in [16].

Figure 29 shows the definitions for the two commands **enlarge** and **add**. The first case of definition (1) in Figure 29 returns a list of $\mathbf{add}^c\ i$ commands, one for each column to be added to enlarge class c at its right edge. Definition (2) returns a list containing update commands that add the given formula to each instance of class c. For our example, we assume the instance spreadsheet of our budget sheet (Figure 11) that has three category instances and two year
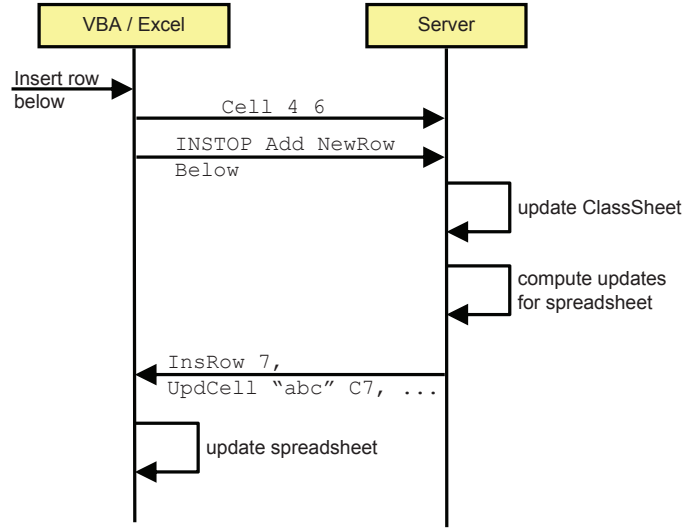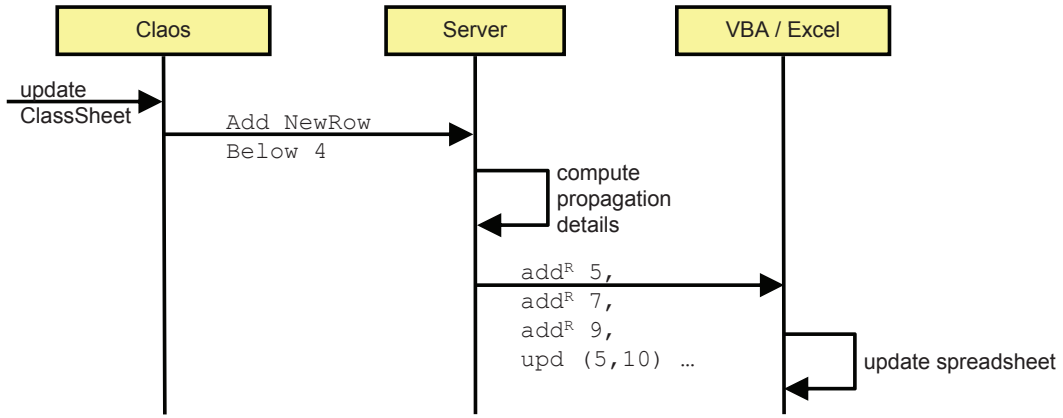
19

Figure 27: The Control Flow for Inserting a Row

Figure 28: The Control Flow for Inserting a Row in the ClassSheet

$$
[\![\, \textbf{enlarge } c\ d \,]\!]^{P}\ \mathsf{S} \quad = \quad
\begin{cases}
\Big[\textbf{add}^{\mathsf{C}}\ (x+1) \mid x \in \texttt{getColI}\ \overleftrightarrow{\mathsf{c}}\ c\ \mathsf{S}'\Big] & \textbf{if } d = \rightarrow \\
\ldots & \ldots
\end{cases}
\tag{1}
$$
$$
\texttt{where } \mathsf{S}' = [\![\, \textbf{enlarge } c\ d \,]\!]^{U}\ \mathsf{S}
$$

$$
[\![\, \textbf{add } c.a = f \,]\!]^{P}\ \mathsf{S} \quad = \quad [(\textbf{upd }\ x\ f) \mid x \in \texttt{getCellI}\ (i, j)\ c\ \mathsf{S}']
\tag{2}
$$
$$
\texttt{where } \mathsf{S}' = [\![\, \textbf{add } c.a = f \,]\!]^{U}\ \mathsf{S}
$$
$$
(i, j) = (\mathsf{S}'.\mathsf{L_V}(c).fst)^{-1}(a, f)
$$

Figure 29:

instances. Visually, the enlargement of our category class is expressed by a new column in our CS graph. Since the

20

CpY class's size depends on its associated classes, it is enlarged as well. The formal definition of the CS graph without the enlargement is defined as follows.[6]

$$
\begin{aligned}
S \quad = \quad &(\{\textbf{Bud}, \textbf{Cat}, \textbf{Year}, \textbf{CpY}\}, \\
&\{(\textbf{Bud}, \textbf{Cat}, \textbf{Comp}^{\Rightarrow}\, 2), (\textbf{Bud}, \textbf{Year}, \textbf{Comp}^{\Downarrow}\, 2), (\textbf{Cat}, \textbf{Year}, \textbf{Assoc}\,\textbf{CpY})\}, \\
&\{(\textbf{Bud}, \dots, \textbf{Class}\,(3,3)\,(0,0)), (\textbf{Cat}, \dots, \textbf{Class}\,(3,2)\,(2,2)), \\
&(\textbf{Year}, \dots, \textbf{Class}\,(3,3)\,(1,1)), (\textbf{CpY}, \dots, \textbf{Assoc})\})
\end{aligned}
$$

Of most interest are the classes' sizes and the instance information. The instance information used to perform some computations hold the number of instances for each class in the particular instance spreadsheet. The type stored in the labeling function had been defined previously as follows where the second value of the repetition part describes the number of instances.

$$
\begin{aligned}
\textit{Type} \quad &::= \quad \textbf{Class}\,\textit{Size Rep}\ |\ \textbf{Assoc} \\
\textit{Rep} \quad &= \quad \mathbb{N} \times \mathbb{N}
\end{aligned}
$$

Since our **Year** class has two instances the data structure stores the value 1 to mark the one repetition: **Class** $(3,3)\,(1,\textbf{1})$. Now let us enlarge the **Year** class at its right edge. The corresponding command is **enlarge Year** $\rightarrow$. The semantics for this command are defined as follows (cf. Definition (1) in Figure 29).

$$
[\![\,\textbf{enlarge Year} \rightarrow]\!]^{P}\, S \quad = \quad \left[\,\textbf{add}^{C}\,(x+1)\ |\ x \in \texttt{getColI}\, \overleftrightarrow{\textbf{Year}}\, \textbf{Year}\, S'\,\right]
$$
$$
\texttt{where}\ S' = [\![\,\textbf{enlarge Year} \rightarrow]\!]^{U}\, S
$$

Let us approach the result step-wise. First of all, the updated CS graph S' is computed in the **where** clause. This was shown in Section 3.1.2. The $\texttt{getColI}$ function returns a sequence of all columns that match the given column $\overrightarrow{\textbf{Year}} = 3$ in the given class **Year** in the instance S'. In our example, we have two instances of the **Year** class and thus—due to our spatial layout—six columns. The returned sequence is [8, 5]. This sequence is returned to the semantics definition, and we obtain the following result.

$$
\left[\,\textbf{add}^{C}\,(x+1)\ |\ x \in [8,5]\,\right] \quad = \quad \left[\,\textbf{add}^{C}\,9, \textbf{add}^{C}\,6\,\right]
$$

The result means the spreadsheet application adds columns behind columns 8 and 5 in exactly this order.[7]

Now we assume that the operations of moving the column and adding the label "Rate" have already been performed and take a look at the more interesting case of adding the formula. Definition (2) in Figure 29 is instantiated as follows.

$$
[\![\,\textbf{add CpY}.\text{rate} = 0\,]\!]^{P}\, S \quad = \quad [\,\textbf{upd}\,x\,0\ |\ x \in \texttt{getCellI}\,(i,j)\,\textbf{CpY}\,S'\,]
$$
$$
\texttt{where}\ S' = [\![\,\textbf{add CpY}.\text{rate} = 0\,]\!]^{U}\, S
$$
$$
(i,j) = (S'.\textsf{L}_{\textsf{V}}(\textbf{CpY}).\textit{fst})^{-1}(\text{rate}, 0)
$$

In the where clause the update is performed on the CS graph S to retrieve the changed graph S'. The second line in the where clause retrieves the cell location of the added formula by simple function application. The function *fst* returns a tuple's first component. Using the function $\texttt{getCellI}$ these coordinates are converted into the corresponding list of coordinates in the ClassSheet instance. In our example, the equation evaluates as follows.

$$
\begin{aligned}
[\![\,\textbf{add CpY}.\text{rate} = 0\,]\!]^{P}\, S \quad &= \quad [\,\textbf{upd}\,x\,0\ |\ x \in [(5,4),(5,5),(5,6),(9,4),(9,5),(9,6)]\,] \\
&= \quad [\,\textbf{upd}\,(5,4)\,0, \textbf{upd}\,(5,5)\,0...\,]
\end{aligned}
$$

Finally, the resulting list of update commands is sent to the VBA add-in in Microsoft Excel, which parses and executes the commands to complete the model evolution.

---

[6] Attributes have been left out.

[7] This order is mandatory since the insertion of rows might change the overall row numbering.

4. Related Work

The problem of erroneous spreadsheets has been addressed from many perspectives. Adopting software engineering techniques, such as testing [17] and debugging [18], for spreadsheets has been successful in supporting end users in the effective localization of errors. But even though these approaches cost less time and are an improvement over the more traditional auditing and code inspection approaches [19, 20, 21], a challenge is still how to motivate the users to spend time and effort on the error finding process.

Automatic error detection approaches, such as type checking [22, 23, 24, 25, 26], can avoid this problem, but they can generally only find a subset of errors.

Therefore, approaches that try to avoid errors from getting into spreadsheets in the first place seem to be another promising alternative. The adoption of good spreadsheet design practices has been repeatedly promoted [27, 28, 29], but these approaches cannot guarantee any level of correctness. Introducing errors into spreadsheets during maintenance can be prevented using an ontology-driven approach to generate a help system as proposed in [30, 31]. Although, this approach seems to be very powerful, it appears to be not as user friedly as necessary for most spreadsheet developers. A more high-level spreadsheet model that can provably guarantee the absence of a large class of referencing and range errors has been proposed in [13, 32]. The ClassSheet model presented in [9] and discussed in this paper is a generalization of that model. ClassSheets allow the modeling of spreadsheets in a model-driven approach using the well-known and user friendly visual Unified Modeling Language (UML).

Model-driven engineering [33] is well-accepted and well-established in science and industy [34]. Current efforts are spent into developing solid solutions to spread MDE into more and more domains [35, 33]. One of the best known initiatives of MDE is MDA [36, 37, 38], launched by OMG [39] in 2001. Tool support for a complete MDE process was provided by Fujaba [40, 41] since 13999, allowing a complete round-trip engineering process [42] with UML and Java. The UML supports the usage of an MDE process in specific domains by allowing the profiling of their language [43, 44]. Several approaches use UML extension mechanisms to generate web applications [45, 46, 44] and embedded systems [47, 48]. MDE approaches either require the user to assemble the generated code manually or are limited to a small domain where the visual expressiveness suffices to describe software while still keeping clarity. The approach presented in this article uses the MDA development approach. MDA separates design from architecture where design maps to class graphs and architecture maps to CS graphs [16]. The use of the UML and the MDA enables utilizing of well-known design techniques from software engineering such as design patterns and the like. This approach differs from other spreadsheet development environments such as Quantrix [49] that rely on cell-oriented spreadsheet programming instead of using explicit object-oriented modeling.

5. Conclusion and Future Work

ClassSheets present an effective way of designing spreadsheet models using several popular paradigms from actual computer science research. ClassSheets define object-oriented models that are used as templates and allow the use of an MDE style development process for spreadsheets. To deal with the problem of maintainability, we propose a formally defined graph data structure for ClassSheets to enable the easy definition of propagation rules that allow the semantics-preserving propagation of updates on ClassSheets to ClassSheet instances, i.e. spreadsheets. To express updates on ClassSheets we introduced an update language and described its semantics using denotational style. We have developed a prototype system that uses the ClassSheet editor Claos to propagate model updates to spreadsheets.

This work does not cover the availability of a ClassSheet's different versions. That means, once a model has been modified, its old versions are no longer available for users or applications. This could be addressed by introducing a concept that resembles the database view concept. Furthermore, we did not go into detail describing a data structure that allows a transaction-based propagation of updates. Transaction-based propagation would provide the ability to optimize update sequences. Related to this approach is an undo/redo concept, which is not fully supported in the prototype yet. The reverse engineering approach of inferring ClassSheets from spreadsheets was not discussed either. This would enable a convenient way of development, proposed by the ARE paradigm. Similar to the approach discussed in [50], ClassSheets would have to be inferred from existing spreadsheets. However, this is less trivial since ClassSheets carry more information than spreadsheets. In particular, if these inferred ClassSheets were compared to existing, eventually older versions of the underlying ClassSheet, a concept of wildcards would have to be introduced to match existing class names to automatically inferred class names. The ClassSheet inference combined with, for

example, a ClassSheet difference algorithm would yield great benefits in handling ClassSheets and spreadsheets. For example, updates on spreadsheets could be inferred as updates in the ClassSheet model. The user could then be asked whether this was his purpose. This would greatly contribute to the idea of spreadsheet safety.

References

[1] R. R. Panko, Spreadsheet errors: What we know. what we think we can do., In Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG), 2000.

[2] B. Boehm, V. R. Basili, Software defect reduction top 10 list, Computer 34 (1) (2001) 135–137. `doi:http://doi.ieeecomputersociety.org/10.1109/2.962984`.

[3] R. R. Panko, The spreadsheet research website, `http://panko.shidler.hawaii.edu/SSR` (2000).

[4] L. Pryor, Spreadsheet research website, `http://www.louisepryor.com` (2007).

[5] C. Weekly, Celebrations halted, `http://www.microscope.co.uk/articles/article.asp?liArticleID=139213` (2006).

[6] L. P. Johnson, M. A. Sides, The Sarbanes-Oxley Act and Fiduciary Duties, SSRN eLibrary.

[7] MOF, Unified Modeling Language (UML), `http://www.omg.org/spec/UML/2.3/` (2010).

[8] P. Chen, The entity-relationship model—toward a unified view of data, ACM Trans. Database Syst. 1 (1) (1976) 9–36. `doi:http://doi.acm.org/10.1145/320434.320440`.

[9] G. Engels, M. Erwig, ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications, in: 20th IEEE/ACM Int. Conf. on Automated Software Engineering, 2005, pp. 124–133.

[10] J. Bzivin, G. Dup, F. Jouault, G. Pitette, J. E. Rougui, First experiments with the atl model transformation language: Transforming xslt into xquery, in: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture, 2003.

[11] MOF, Query / Views / Transformations (QVT), `http://www.omg.org/spec/QVT/1.1/Beta2/` (2009).

[12] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, A. Schürr, Metamodel-based tool integration with moflon, in: ICSE '08: Proceedings of the 30th international conference on Software engineering, ACM, New York, NY, USA, 2008, pp. 807–810. `doi:http://doi.acm.org/10.1145/1368088.1368206`.

[13] M. Erwig, R. Abraham, S. Kollmansberger, I. Cooperstein, Gencel — A Program Generator for Correct Spreadsheets, Journal of Functional Programming 16 (3) (2006) 293–325.

[14] G. E. J.-C. Bals, F. Christ, M. Erwig, Classsheets - model based, object oriented design of spreadsheet applications, Journal of Object Technology 6 (9).
URL `http://www.jot.fm/issues/issue_2007_10/paper19/`

[15] J. Bals, F. Christ, ClassSheets - modellbasierter, werkzeuggestuetzter Entwurf von Spreadsheet-Anwendungen, diploma Thesis (2006).

[16] M. Luckey, Automatic Propagation of Model Updates in the Spreadsheet Paradigm, bachelor Thesis, University of Paderborn, Oregon State University (2007).

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

[18] R. Abraham, M. Erwig, GoalDebug: A Spreadsheet Debugger for End Users, in: 29th IEEE Int. Conf. on Software Engineering, 2007, pp. 251–260.

[19] R. R. Panko, Applying Code Inspection to Spreadsheet Testing, Journal of Management Information Systems 16 (2) (1999) 159–176.
URL `ApplyingCodeInspectiontoSpreadsheetTesting.pdf`

[20] J. Sajaniemi, Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization, Journal of Visual Languages and Computing 11 (2000) 49–82. `doi:10.1006/jvlc.1999.0142`.
URL `http://www.idealibrary.com`

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

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

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

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

[25] R. Abraham, M. Erwig, Type Inference for Spreadsheets, in: ACM Int. Symp. on Principles and Practice of Declarative Programming, 2006, pp. 73–84.

[26] R. Abraham, M. Erwig, UCheck: A Spreadsheet Unit Checker for End Users, Journal of Visual Languages and Computing 18 (1) (2007) 71–95.

[27] T. Isakowitz, S. Schocken, H. C. Lucas, Jr., Toward a Logical/Physical Theory of Spreadsheet Modelling, ACM Transactions on Information Systems 13 (1) (1995) 1–37.

[28] K. Rajalingham, D. Chadwick, B. Knight, D. Edwards, Quality Control in Spreadsheets: A Software Engineering-Based Approach to Spreadsheet Development, in: 33rd Hawaii Int. Conf. on System Sciences, 2000, pp. 1–9.

[29] A. G. Yoder, D. L. Cohn, Real Spreadsheets for Real Programmers, in: Int. Conf. on Computer Languages, 1994, pp. 20–30.

[30] A. Kohlhase, M. Kohlhase, Compensating the computational bias of spreadsheets with mkm techniques, in: Calculemus '09/MKM '09: Proceedings of the 16th Symposium, 8th International Conference. Held as Part of CICM '09 on Intelligent Computer Mathematics, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 357–372. `doi:http://dx.doi.org/10.1007/978-3-642-02614-0_29`.

[31] A. Kohlhase, M. Kohlhase, Spreadsheet interaction with frames: Exploring a mathematical practice, in: Calculemus '09/MKM '09: Proceedings of the 16th Symposium, 8th International Conference. Held as Part of CICM '09 on Intelligent Computer Mathematics, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 341–356. `doi:http://dx.doi.org/10.1007/978-3-642-02614-0_28`.

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

[33] D. C. Schmidt, Guest editor's introduction: Model-driven engineering, IEEE Computer 39 (2) (2006) 25–31.

[34] R. B. France, D. Ghosh, T. Dinh-Trong, A. Solberg, Model-Driven Development Using UML 2.0: Promises and Pitfalls, Computer 39 (2) (2006) 59–66. `doi:http://doi.ieeecomputersociety.org/10.1109/MC.2006.65`.

[35] R. France, B. Rumpe, Model-driven development of complex software: A research roadmap, in: FOSE '07: 2007 Future of Software Engineering, IEEE Computer Society, Washington, DC, USA, 2007, pp. 37–54. `doi:http://dx.doi.org/10.1109/FOSE.2007.14`.

[36] OMG, Model Driven Architecture, `http://www.omg.org/mda/` (2001).

[37] D. DSouza, Model-driven architecture and integration: Opportunities and challenges, `http://www.catalysis.org/publications/papers/2001-mda-reqs-desmond-6.pdf` (2001).

[38] J. D. Poole, Model-driven architecture: Vision, standards and emerging technologies, in: ECOOP: Workshop on Metamodeling and Adaptive Object Models, 2001, `http://www.omg.org/mda/mda_files/Model-Driven_Architecture.pdf`.

[39] OMG, Object Management Group, `http://www.omg.org/` (2010).

[40] U. Nickel, J. Niere, A. Zündorf, The FUJABA environment, in: ICSE '00: Proceedings of the 22nd international conference on Software engineering, ACM, New York, NY, USA, 2000, pp. 742–745. `doi:http://doi.acm.org/10.1145/337180.337620`.

[41] U. A. Nickel, J. Niere, J. P. Wadsack, A. Zündorf, Roundtrip engineering with FUJABA (extended abstract) (2000).
URL `citeseer.ist.psu.edu/383005.html`

[42] A. Henriksson, H. Larsson, A definition of round-trip engineering, `http://www.ida.liu.se/~henla/papers/roundtrip-engineering.pdf` (January 2003).

[43] L. Fuentes-Fernndez, A. Vallecillo-Moreno, An introduction to uml profiles, UPGRADE 5 (2).

[44] N. Moreno, P. Fraternalli, A. Vallecillo, A UML 2.0 profile for WebML modeling, in: ICWE '06: Workshop proceedings of the sixth international conference on Web engineering, ACM, New York, NY, USA, 2006, p. 4. `doi:http://doi.acm.org/10.1145/1149993.1149998`.

[45] N. Koch, Transformation techniques in the model-driven development process of uwe, in: ICWE '06: Workshop proceedings of the sixth international conference on Web engineering, ACM, New York, NY, USA, 2006, p. 3. `doi:http://doi.acm.org/10.1145/1149993.1149997`.

[46] A. Kraus, A. Knapp, N. Koch, Model-Driven Generation of Web Applications in UWE, in: N. Koch, A. Vallecillo, G. Houben (Eds.), Proc. 3rd Int. Wsh. Model-Driven Web Engineering (MDWE'07), Vol. 261 of CEUR-WS, 2007, pp. 1–16.
URL `http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-261/paper03.pdf`

[47] S. Burmester, H. Giese, M. Hirsch, D. Schilling, M. Tichy, The fujaba real-time tool suite: model-driven development of safety-critical, real-time systems, in: ICSE '05: Proceedings of the 27th international conference on Software engineering, ACM, New York, NY, USA, 2005, pp. 670–671. `doi:http://doi.acm.org/10.1145/1062455.1062601`.

[48] B. Schtz, A. Pretschner, F. Huber, J. Philipps, Model-based development of embedded systems, in: In Advances in Object-Oriented Information Systems, Lecture, Springer-Verlag, 2002, pp. 298–312.

[49] Quantrix, Quantrix Modeler, `http://www.quantrix.com/` (2010).

[50] R. Abraham, M. Erwig, Inferring templates from spreadsheets, in: ICSE '06: Proceeding of the 28th international conference on Software Engineering, ACM Press, New York, NY, USA, 2006, pp. 182–191. `doi:http://doi.acm.org/10.1145/1134285.1134312`.