From Relational ClassSheets to UML+OCL

Jácome Cunha∗
Universidade do Minho
Portugal
jacome@di.uminho.pt

João Paulo Fernandes†
Universidade do Minho &
Universidade do Porto
Portugal
jpaulo@di.uminho.pt

João Saraiva
Universidade do Minho
Portugal
jas@di.uminho.pt

ABSTRACT

Spreadsheets are among the most popular programming languages in the world. Unfortunately, spreadsheet systems were not tailored from scratch with modern programming language features that guarantee, as much as possible, program correctness. As a consequence, spreadsheets are populated with unacceptable amounts of errors.

In other programming language settings, model-based approaches have been proposed to increase productivity and program effectiveness. Within spreadsheets, this approach has also been followed, namely by ClassSheets. In this paper, we propose an extension to ClassSheets to allow the specification of spreadsheets that can be viewed as relational databases. Moreover, we present a transformation from ClassSheet models to UML class diagrams enriched with OCL constraints. This brings to the spreadsheet realm the entire paraphernalia of model validation techniques that are available for UML.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification; D.2.2 [Software Engineering]: Object-oriented design methods; F.1.1 [Models of Computation]: [Relations between models]

General Terms
Design, Languages, Verification

Keywords
Spreadsheets, UML, OCL, ClassSheets

∗Supported by Fundação para a Ciência e a Tecnologia, grant SFRH/BPD/73358/2010.
†Supported by Fundação para a Ciência e a Tecnologia, grant SFRH/BPD/46987/2008.
Work supported by the SSaaPP project, FCT contract PTDC/EIA-CDC/108613/2008.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC’12 March 25-29, 2012, Riva del Garda, Italy.
Copyright 2011 ACM 978-1-4503-0857-1/12/03 ...$10.00.

1. INTRODUCTION

Spreadsheets are widely used by non-professional programmers, the so-called end users, to develop business applications. Spreadsheet systems offer a high level of flexibility, making it easy to start working with them. This freedom, however, comes with a price: spreadsheets are error prone as shown by numerous studies which report that up to 90% of real-world spreadsheets contain errors [12, 13, 14].

In recent years the spreadsheet research community has recognized the need to support end-user model-driven software development, and to provide spreadsheet developers and end users with methodologies, techniques and the necessary tool support to improve their productivity. Along these lines, several techniques have been proposed [1, 6, 9], being ClassSheets [7] the most powerful domain specific model for spreadsheets. ClassSheets provide a powerful paradigm for model-driven software development: a spreadsheet business model is first defined, from which a customized spreadsheet application is generated guaranteeing the consistency of the spreadsheet with the underlying model.

Despite of its huge benefits, the ClassSheets model has two important drawbacks: firstly, it is not powerful enough to capture common end user errors. For example, a great number of spreadsheets represent database tables1, where there is an embedded notion of a table key in one column (as known from database theory). In this context, end users should be warned by the spreadsheet system if, e.g., a duplicated key is introduced in such a column. In fact, in [6] we have proposed techniques for spreadsheet edit assistance that rely on relational database models, and that guide end users in introducing correct data. The ClassSheet model is a very table syntactic oriented formalism and it lacks this notion of relational databases [10].

Secondly, although ClassSheets is a model-based formalism, there is no connection between it and the languages and techniques developed by the modelware community. In fact, the modelware community has done a considerable amount of work on model design, transformation, evolution and co-evolution of models and instances [2], that we are convinced the ClassSheet formalism could reuse and benefit from.

The goal of this paper is three-fold:

• Firstly, we extend ClassSheets with the notion of relational databases. As a result, the spreadsheet data is not only organized in ClassSheets tables, but such tables also have the notion of primary and foreign keys.

1Studies suggest that a huge percentage of spreadsheets are in fact databases, since no formula is used in them! [3].
2. CLASSSHEETS

ClassSheets [7] are a high-level, object-oriented formalism to specify the business logic of spreadsheets. ClassSheets allow users to express business object structures within a spreadsheet using concepts from the Unified Modeling Language (UML). Using the ClassSheets model, it is possible to define spreadsheet tables and give them names, define labels for the table’s columns, specify the types of the values such columns may contain and also the way the tables expand (e.g., horizontally or vertically).

In this section we review the embedding of ClassSheets in spreadsheet systems, a technique that we have proposed in [5]. We mimic the well-known embedding of a domain specific language in a general purpose one and inherit all the powerful features of the host language: in our case, the powerful interactive interface offered by the (host) spreadsheet system. In order to present our embedding, we consider a model-based spreadsheet system, adapted from [10], to manage an airline company. The fragment of the ClassSheet model for such a system that deals with the pilots’ salaries is given in Figure 1.

Figure 1: Using ClassSheets to create a Salaries sheet.

From an object-oriented point of view, one can see a summation object, which aggregates a list of objects containing single values. Looking at the layout model, starting from the blue part, we see a class labeled Pilot, consisting of a value for which the default is 1000. In fact, this is not a single value, but a list of values, since the row after it is labeled with ellipses. The summation object, in red, is defined by a label Salaries, a footer labeled Total and an aggregation formula assigned to an attribute named total.

Such an object-oriented extended template is a ClassSheets model since it defines classes together with their attributes and aggregational relationships. ClassSheets consist of a list of attribute definitions grouped by classes and arranged on a two dimensional grid. Additional labels are used to annotate the concrete representation. References to other entries are defined by using attribute names, as shown in the SUM formula in the example. The formal language of ClassSheets is as follows [7]:

\[
\begin{align*}
  f & \in \text{Fml} & ::= & \phi \mid a.a \mid \phi(f, \ldots, f) \\
  b & \in \text{Block} & ::= & \phi \mid a = f \mid b \mid b + b & \text{(formulas)} \\
  l & \in \text{Lab} & ::= & h \mid v \mid n & \text{(classlabels)} \\
  t & \in \text{Hor} & ::= & | n | n & \text{(horizontal)} \\
  v & \in \text{Ver} & ::= & | n | n & \text{(vertical)} \\
  c & \in \text{Class} & ::= & 1 : b \mid b^* : c^+ : c & \text{(classes)} \\
  s & \in \text{Sheet} & ::= & c \mid c^* \mid s \mid s & \text{(sheets)}
\end{align*}
\]

In the next sections, we go through several concrete illustrations of constructions that are common to both spreadsheet systems and ClassSheets.

2.1 Vertically Expandable Tables

Suppose that we want to record the activity of the company’s pilots. A simple way of achieving this is to use a spreadsheet, and a table as the one presented in Figure 2a. This table has a title, Pilots, and a row with labels, one for each of the table’s column: ID represents a unique pilot identifier, Name represents the pilot name and Flight hours represents the total number of hours a pilot has flown. Each of the subsequent rows represents a concrete pilot.

(a) A concrete table.

(b) The visual model.

(c) The formal model.

Figure 2: Pilot activity record.

Tables such as the one presented in Figure 2a are frequently used within spreadsheets, and it is fairly simple to create a model specifying such tables. For the example shown, we can extract the visual ClassSheets model presented in Figure 2b and its corresponding formal definition shown in Figure 2c.

To model the labels we use a textual representation and the exact same names as in the data sheet (Pilots, ID, Name and Flight hours). To model the actual data we abstract concrete column cell values by using a single identifier: we use the one-worded, lower-case equivalent of the corresponding column label (so, id, name and flight_hours). Next, a type is associated with each column: columns k and b hold strings and column c holds integer values (denoted in the model, respectively, by the empty string and 0 following the sign). Notice that the fourth row of the visual model contains vertical ellipses in all columns. This means that it is possible for this column to expand vertically: the tables that conform to this model can have has many rows as needed. The scope of the expansion is between the ellipsis and the black line (between row 2 and 3). In the formal model this expansion possibility is expressed by the sign which affects the same spreadsheet elements.

2.2 Horizontally Expandable Tables
An airline company must also store information on its airplanes. This is the purpose of table Planes in the spreadsheet illustrated in Figure 3a, which is organized as follows: the first column holds labels that identify each row; namely, N-Number, Model and Name; cells in row N-Number (respectively Model and Name) contain the unique identifier of a plane, (respectively the model of the plane and the name of the plane). Each of the subsequent columns contains information about one particular aircraft.

Figure 3: Information on airplanes.

The Planes table can be modelled by the illustration in Figure 3b, whose formal definition is given in Figure 3c. This model may be constructed following the same strategy as in the previous section, but now swapping columns and rows. The first column contains the label information and the second column the names abstracting concrete data elements in that row (in this example, all cells are to hold strings). The third column has ellipses meaning that rows and the second column the names abstracting concrete data elements in that row (in this example, all cells are to hold strings). The third column has ellipses meaning that rows are horizontally expandable. Notice that the instance table has information about three planes.

2.3 Relationship Tables

So far in the paper, we have modelled tables for pilots and planes; reusing what we built, we can now model, as shown in Figure 4a, a table to store information on concrete flights.

We can see some of the information we had before: in the top left corner we have a concrete spreadsheet where, between rows 8 and 12, we have the pilot information shown in Figure 2a and between rows 15 and 18 the plane information shown in Figure 3a. The lines between rows 1 and 5 represent flight scheduling information. For simplicity, let us for now focus on columns A to E. Column A holds the identifier of the pilot for a concrete flight. Row 2, columns B and E, hold the identifiers of the airplanes assigned to fly from OPO to NAT, two times, and from LIS to AMS, respectively. Origins and destinations of flights are registered in Depart and Destination columns, as well as the date and hour of departure (column Date) and the number of hours the flight will take (column Hours). Notice that we can have as many entries for pilots (planes, respectively) as we need just by adding one row per pilot (and 4 columns per plane). An example of how we read this table is as follows: pilot pl1 flew plane N2342 from OPO to NAT on December 12th, 2010, at 14:00 hours and the flight took 7 hours.

Figure 4: Flights table model.

The spreadsheet has already been edited after the initial generation. The generated spreadsheet contains the labels in bold on the model, the initial formulas and buttons to add new vertical and horizontal blocks of cells. For example, in the Pilots table, there is a button on row 13 which will insert a new row. The values that will appear in the new row are the default values defined in the model and the user can only update them to a value of the same type (string, integer, etc.). A more complex situation would be to add a new flight; this involves a pilot and a plane, and some more information. If the user clicks the button in row 6, the system will add a new row as explained before, but in this case it will also update the necessary formulas; it will update the formulas in cells E7, I7 and K7 to include the new added row. This mechanism prevents the user from editing the spreadsheet without correctly updating its formulas, and therefore from corrupting it. The button in column J works in a similar way, updating the formulas in cells K4 and K5.

3. CLASSSHEETS AND DATABASES

In the previous section we have reviewed ClassSheets as proposed in [7] and we have shown a concrete example of how it can be used in practice. In this section we extend the original ClassSheet formal language with constructions from the relational database realm. In this way, we extend further the amount of validations that are possible for spreadsheets derived from ClassSheets.

We propose the following extensions, marked in red:

\[
\begin{align*}
\varphi \in Fml &::= \varphi \lor n.a | \varphi(f, \ldots, f) \quad \text{(formulas)} \\
\varphi \in Block &::= \varphi | \varphi^\ast | a = f | b \cdot b | b \ast b \quad \text{(blocks)}
\end{align*}
\]

In a model, as a consequence of the syntactic extension \(\varphi^\ast\), a column or row can now be declared as containing unique values. Each of these values is thought of as a primary key in a relational database. This has important semantic consequences. Consider, for example, the pilot activity record presented in Section 2.1. There, we declared that the ID column should hold strings, but the model was not rich enough to ensure that each pilot in the derived spreadsheet (or each row in the corresponding column) has a unique identification. This can cause two different problems/errors: 1) two different pilots could be given the same ID, with obvious implications (one being, for example, a pilot assigned to fly a
(a) Spreadsheet instance and ClassSheet model of an airline company.

<table>
<thead>
<tr>
<th>Flights</th>
<th>Flights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flights</td>
<td>Flights</td>
</tr>
<tr>
<td>PilotsKey</td>
<td>PilotsKey</td>
</tr>
<tr>
<td>PilotsKey</td>
<td>(pilot_key = Pilots.id)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Planeskey</th>
<th>Planeskey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planeskey</td>
<td>plane_key = Planes.n-number</td>
</tr>
<tr>
<td>Flight</td>
<td>(Depart : Destination : Date : Hours)</td>
</tr>
<tr>
<td>Flight</td>
<td>(depart = &quot;&quot; : destination = &quot;&quot; : date = d : hours = 0)</td>
</tr>
<tr>
<td>Planeskey</td>
<td>(pilot = p)</td>
</tr>
<tr>
<td>Flights</td>
<td>Flights</td>
</tr>
<tr>
<td>TotalPilotHours</td>
<td>TotalPilotHours</td>
</tr>
<tr>
<td>Flights</td>
<td>(total = SUM(hours))</td>
</tr>
</tbody>
</table>

(b) The formal model.

Figure 4: Spreadsheet of an airline company and an abstract model representing it.

co-worker’s flight); 2) the same pilot could occur in different rows, with potentially different flight hour records.

This extension is useful for the models that we have seen so far. In fact, it should be used for the fragments storing both the information on pilots and planes of Figures 2 and 3, whose identifiers must be unique:

<table>
<thead>
<tr>
<th>Pilots</th>
<th>Pilots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pilots</td>
<td>IDa</td>
</tr>
<tr>
<td>Pilots</td>
<td>Name</td>
</tr>
<tr>
<td>Pilots</td>
<td>Flight Hours</td>
</tr>
<tr>
<td>Planes</td>
<td>Planes</td>
</tr>
<tr>
<td>Planes</td>
<td>N-Number</td>
</tr>
<tr>
<td>Planes</td>
<td>Model</td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
</tbody>
</table>

Now, and just as an example of the extension at work, if a user tries to insert a new pilot with an identifier already in use, the spreadsheet derived from the pilots model above is now able of producing the warning shown in Figure 5.

Figure 5: Uniqueness value validation.

As for the extension n.a, it is actually not of a syntactic nature. Indeed, you may notice that such a definition already occurred in the original ClassSheet language as a way of referring, in a class, to an attribute of a different class. In our work, however, this has a stronger interpretation. Indeed, we view one such occurrence in the same way a foreign key is viewed in a database model. This means that, when a spreadsheet model is being constructed, we first check whether a class n, and the corresponding attribute a do exist before accepting a cell declaration n.a. Then, we also verify whether a has been declared as unique (i.e., as a primary key) in block n. A final validation of the derived spreadsheet is ensured by construction: it is not possible to have erroneous values in these cells since the user is only allowed to fill them by selecting a value from a list with precisely the values in n.a. Actually, in case the user inserts a value of the appropriate type in a foreign key cell, we do not immediately consider an error the fact that such value is not a primary key of the corresponding table: a situation like that may reveal an intention of introducing a new tuple (key, values) in that table. Therefore, the user is in this case asked either to confirm the erroneous situation or to introduce the values to associate with key, that we correctly insert in the right table. A concrete situation of this kind is illustrated in Figure 6, when the user tries to assign pilot pl4 to a given flight whereas no pilot with that PilotsKey exists in the Pilots table.

These validations were originally not ensured and, at the
4. FROM CLASSSHEETS TO UML+OCL

The Unified Modeling Language (UML) is one of the most frequently used languages to specify and document software systems [15]. In particular, class diagrams are very useful to design business applications. In this section we propose to map ClassSheets into UML class diagrams. With this mapping we enable further transformations from spreadsheets to other paradigms (e.g., the object-oriented).

Recall the Salaries ClassSheets model that we presented in Figure 1. In Figure 7 we show it again (Figure 7a) together with its equivalent class diagram (Figure 7b).

![Figure 6: Foreign key validation.](image)

![Figure 7: Two models for the Salaries sheet.](image)

This example illustrates the similarities between the two representations. The (ClassSheets) Pilot class is represented in UML as a class with the same name and the same attribute, value, of type Integer and initial value 1000. For the Salaries class the transformation is achieved in a similar way, just noticing that total is defined as the sum of the values of the Pilot class. The two classes are connected: Salaries is composed of zero or more Pilot elements.

In the following sections we systematize the transformation of ClassSheets into UML by translating each of the elements that compose the extended ClassSheets language that we have given in Section 3. In fact, we generate specifications within the UML-based Specification Environment (USE); this framework supports UML execution and OCL constraint checking therefore providing analysts, designers and developers with the opportunity to employ model-driven techniques for software production [8].

4.1 Mapping ClassSheets Into UML Classes

The mapping from ClassSheets to UML is divided into two phases: firstly, UML classes are generated; secondly, the associations between the generated classes are inferred. In this section we present the first transformation encoded in function T, which uses some auxiliary functions. The first of these functions is getType: it receives a formula \( f \), which is used to define the value of an attribute \( a \) such that \( a = f \), and returns a pair whose first component is the type of \( a \) and whose second component is its (textual) definition.

\[
\begin{align*}
Type &= String \\
Definition &= String \\
AttType &= Type \times Definition \\
getType \varphi &= (getType_1 \varphi, \varphi) \\
getType (n, a) &= getType (getTypeDef (n, a)) \\
getType (\varphi (f, \ldots, f)) &= (getType_2 (\varphi (f, \ldots, f)), (\varphi (f, \ldots, f)))
\end{align*}
\]

If an attribute is defined by a default value \( \varphi \), \( getType \) returns the type of \( \varphi \) as a string (given by \( getType_1 \)) together with \( \varphi \) itself. As an example, if we have an attribute definition \( a = \varphi \) such as value = 0, then the result of \( getType \varphi \) is \((\text{Integer, 0})\).

In case an attribute is defined as a reference to another, the function \( getAttDef \) is used to get such attribute definition. The function \( getType \) is then recursively applied to retrieve the attribute’s type and definition.

Finally, if an attribute is defined as a formula, \( getType_2 \) is used to retrieve the formula’s type and the formula definition is itself used to complete the result.

Given the simplicity of \( getType_1 \), \( getAttDef \) and \( getType_2 \), their definition is not given here, but it can be seen in the implementation.

Next, we introduce function \( getAtts \), that receives a block \( b \) and returns the set of attribute definitions that occur in \( b \).

\[
\begin{align*}
Attribute &= \text{AttName} \times \text{AttType} \\
getAtts : \text{Block} &\rightarrow \{ \text{Attributes} \} \\
getAtts \varphi &= \emptyset \\
getAtts (a = f) &= \{ a \times (getType f) \} \\
getAtts (b_1 \cup b_2) &= \text{getAtts} b_1 \cup \text{getAtts} b_2 \\
getAtts (b^n) &= \text{getAtts} b
\end{align*}
\]

Whenever an attribute definition \( a = f \) is found, \( getAtts \) creates the singleton set with the name of the attribute being defined, \( a \), paired with the result of applying \( getType \) to the definition of \( a \), i.e., to \( f \). In the remaining cases, \( getAtts \) simply recurses over the structure of blocks. This function will be used to calculate the attributes of a new UML class.

The next function, called \( getName \), receives the label of a class and returns the string in that label:

\[
\begin{align*}
\text{UMLClassName} &= \text{String} \\
\text{getName} : \text{Lab} &\rightarrow \text{UMLClassName} \\
\text{getName} (n) &= n \\
\text{getName} (c \in s) &= n
\end{align*}
\]

The last auxiliary function, \( addClass \), receives a UML class \( c \) (represented by its name and the set of its attributes) and a set \( s \) of the classes that have already been mapped from ClassSheets to UML and inserts \( c \) in \( s \) according to the rule:

\[
\text{addClass} : \text{UMLClass} \times \{ \text{UMLClass} \} \rightarrow \{ \text{UMLClass} \} \\
\text{addClass} (c, s) &= \text{addClass} (c, s \cup \{ c \})
\]

3That is available in the HaExcel folder of http://haskell.di.uminho.pt/vebmr/
if there is already in $s$ a class with the same name as $c$, then the attributes of $c$ are added to the existing class; otherwise, the new class is added to $s$.

$$UMLClass = UMLClassName \times \text{Attributes}$$

$\text{addClass} : UMLClass \rightarrow \{ \{ \text{UMLClass} \} \rightarrow \{ \text{UMLClass} \} \}$

$\text{addClass} (c) \equiv \{ c \}$

$\text{addClass} (l \times s) \equiv \{ \{ (l' \times s') \cup t) = \}$

$\begin{cases} l' \times (s \cup t) \cup \text{t} \text{ if } l \equiv \text{t} \\ l' \times s' \cup \text{t} \text{ else } \end{cases}$

We are now in position to present the transformation function $T$. It receives a sheet and a set of its initial classes and returns the set of its mapped classes.

$T : \text{Sheet} \rightarrow \{ \text{UMLClass} \} \rightarrow \{ \text{UMLClass} \}$

$T (l, n, \emptyset) = cs$

$T (l, b) = \text{addClass } \text{getName } \text{getAtts } b \times cs$

$T (c_1, c_2) \equiv T c_2 (T c_1 cs)$

$T (c) \equiv T c cs$

$T (s_1 \times s_2) = T s_2 (T s_1 cs)$

Cell classes, i.e., classes labelled with $n$ are not converted to UML classes because they represent relationships between other classes, and thus they will be converted into associations instead (see Section 4.2). This is expressed by the first statement. From each class with a label other than $n$ a new class is generated according to $T$’s second statement. Such class is constructed using the name in the label $l$ and the attributes in the definition of the block $b$; the constructed class is then added to the already computed ones.

When the sheet is built using two classes $c_1$ and $c_2$, the third statement first applies $T$ to $c_1$ and then $T$ again to the result of the first application. A similar situation is implemented in the fourth statement for sheets that consist of the composition of two other sheets $s_1$ and $s_2$.

The remaining case occurs when a sheet is composed by an expandable class (fourth case in $T$). In such case, the function recurses on the expandable class.

As we explained before, $T$ transforms a ClassSheet into a set of UML class diagrams represented as UMLClass values. To make the result available to other tools, such as USE, we need to export it to simple text. The function $ppUMLClass$ receives a UML class diagram and computes a string:

$\text{ppUMLClass} : \text{UMLClass} \rightarrow \text{String}$

$\text{ppUMLClass} \text{(UMLClass } s \text{)} = "$class " \text{start } " s + " \text{\textbackslash \textbackslash attributes } n \text{" } + \text{ppAtts } l + " \text{\textbackslash \textbackslash end } n \"$

Function $\text{ppAtts}$ generates a string for the classes’ attributes:

$\text{ppAtts} : \{ \text{Attribute} \} \rightarrow \text{String}$

$\text{ppAtts} \{ \} = "$$

$\text{ppAtts} \{ \text{(a } \times \text{ (t } \times \text{ d)) } \cup \text{ r) } = \text{a } + " \text{\textbackslash n } \text{t } + " \text{\textbackslash n } \text{d } + \text{ppAtts } r$$

The generated string can be import and manipulated by tools like USE. For our running example, the generated classes are shown next:

class Pilots attributes
id : String = ""
name : String = ""
flight_hours : Integer = 0
end

class PilotsKey attributes
total : Integer = SUM(hours)
pilot_key : String = Pilots.id
end

class Flights attributes
total : Integer = SUM(PlanesKey.total) end

In the next section we will explain how to generate the associations between the UML classes representing the connection between the ClassSheet classes.

4.2 Mapping ClassSheets Into Associations

In the previous section we have presented a systematic transformation from ClassSheet diagrams to UML classes. In this section we introduce a mapping from ClassSheets to UML associations that link the generated classes. These associations together with the generated classes constitute a complete UML model. As in the previous section, we start by presenting some auxiliary functions and definitions.

We view an association as a tuple with the following information: the name of the association, the two classes being linked as well as the cardinalities of the association in each end, and the attributes of the association.

$\text{AssocName} = \text{String}$

$\text{Association} = \text{AssocName} \times (\text{UMLClassName} \times \text{Cardinality}) \times (\text{UMLClassName} \times \text{Cardinality}) \times \{ \text{Attribute} \}$

The first function we introduce, $\text{getName}$, receives a class and returns its name. In case a class is composed of two classes, it returns the name of the first that is returned.

$\text{getName} : \text{Class} \rightarrow \text{UMLClassName}$

$\text{getName} (l) = \text{getName } l$

$\text{getName} (c) = \text{getName } c$

The next function, $\text{getParent}$, returns what we call the parent of a class. The parent of a class with name $c$ is a class $c_1$ if there exists an association between $c_1$ and another class $c_2$ such that the name of $c_2$ matches $c$.

$\text{getParent} : \text{UMLClassName} \rightarrow \{ \text{Association} \} \rightarrow \text{UMLClassName}$

$\text{getParent} c \{ \} = c$

$\text{getParent} c ((\ell x (c_1) x (c_2) x \emptyset)) = \text{if } (c \equiv c_1) \text{ then } c_1 \text{ else } \text{getParent } c \text{ r}$

We are now in conditions to define the function that generates UML association classes from a ClassSheet model. The following code implements such transformation:

$C : \text{Sheet} \rightarrow \text{UMLClassName} \rightarrow \{ \text{Association} \} \rightarrow \{ \text{Association} \}$

$C (c_1, c_2) = (\{ (c_1 \times \ell p, b) \equiv r \} \text{ parent as } = \text{let } n_{c_2} = \text{getName } c_2 \text{ n}_{c_1} = \text{getName } c_1 \text{ assoc } = \text{p } \times \{ n_{c_2} \times (\ast * \ast) \times (n_{c_1} \times (\ast * \ast) \times (\text{getAtts } b) \text{ sh } = (c_1 \times c_2) \equiv (c_1 \text{ EmptyClass } \rightarrow \text{ r }) \text{ in } \{ \text{ assoc } \} \cup \{ \text{ sh } \text{ parent as } \} \text{ C (l : (c_1) \equiv \text{ parent as } = \text{let } n_{l} = \text{getName } l \text{ in } \text{if } \text{ parent } \equiv n_{l} \text{ then } \text{C (getParent } n_{l} \text{ as } \text{ as } = \text{else let } a = n_{l} \times (\text{parent } \times (\ast * \ast) \times (n_{l} \times (\ast * \ast) \times \{ ) \text{ in } \text{C (c_1, c_2) parent as } = \text{C parent (c_1, c_2 parent as ) \text{ (c_1) parent as } = \text{C parent c parent as } \text{C (c_1, c_2) parent as } = \text{C parent c parent as } \text{assoc } c \text{ as } = \text{as }$
The name of this association is the label of the cell class. This association is added to the existing ones and the function recurses over the same input sheet with one difference: the cell class is substituted by an “empty class” so the remaining associations can be found.

The second case occurs when two classes are composed. Here, if parent is equal to the label of the first class, then the function recurses over the second class, but with parent being the parent of the first class. Otherwise, an association between parent and the first class is created. The name of this association is the label of the argument class. The function recurses again over the second class.

In the remaining cases the function recurses over the argument ClassSheets.

As in the previous section, from the output computed by C, we can generate a representation of the associations that can be read by the USE framework. The following function receives an association and returns a string representing it.

```plaintext
ppAssociation :: Association -> String
ppAssociation (n × (c1 × c2)) × (c2 × c3) × \{}\) = ppAssociationAux "association" "n c1 c2 c3 aatts = let patts = "attributes\n" + ppAttributes atts + "\nend"
    in ppAssociationAux
    "association class" "n c1 c2 c3 aatts = patts
ppAssociationAux ao n c1 c2 c3 aatts = ao + n + " between\n" + "vt" + c1 + "[t + cd1 + ]\n" + "vt" + c2 + "[t + cd2 + ]\n"
```

The function has two cases: the first one, with no attributes in the association, generates a regular UML association. For the second case, with attributes, an association class is generated so the attributes can be inserted in the model.

For our example, the associations generated are as follows:

```plaintext
association PilotsKey association PlanesKey
    between Flights[1] between Flights[1]
    PilotsKey[*] end PlanesKey[*] end
associationclass FlightDetails between PilotsKey[*]
    PlanesKey[*] attributes
depart : String = ""
destination : String = ""
date : String = "d"
hours : Integer = 0 end
```

The complete UML class diagram generated by our technique and drawn by the USE tool is shown in Figure 8:

![UML class diagram for the flights example.](image)

Figure 8: UML class diagram for the flights example.

### 4.3 Generating OCL

In the previous sections we introduced functions to generate UML class diagrams from ClassSheets. But we also need to generate OCL code to guarantee that both representations ensure the same properties. In particular, we need to generate certain restrictions on classes that hold primary and foreign keys. In the case of primary keys, we need to ensure that no value used as such key is repeated. In the case of foreign keys, we need to make sure that all these values correctly point to an existing primary key.

To generate OCL code, we first define a function that generates a string containing the attributes that must be inserted in the model. For a class with two primary keys, we define the following function to generate the OCL code for the primary key:

```plaintext
pk : Sheet -> {Lab × {Block}}
```

The auxiliary function `ppPKs` generates a string containing the attributes that implement it. In essence, we define a function that produces lists of pairs, each one containing the label of the class which the invariant is associated with and a set of blocks that represent the primary key.

```plaintext
ppPKs : {Lab × {Block}} -> String
ppPKs \emptyset = ""
ppPKs (\{v | v \∈ \emptyset\}) ∪ r = ppPKs r
ppPKs (\{v | v \∈ \emptyset\}) ∪ lbs = let n = name l
    in "context " + n + "\n" + n + "." + allInstances->" + "forall(a1, a2 | a1 <> a2 implies " + ppPKs' 1 bs + " <> "+ ppPKs' 2 bs + "*) + ppPKs lbs
```

The auxiliary function `ppPKs'` generates a string containing all the blocks. It receives an integer so that it can be reused to generate code for both sides of the inequation on `ppPKs`. For a class with two primary keys "ID" and "name", for example, `ppPKs'` would generate a string such as "a1.ID a1.name".

```plaintext
ppPKs' : Integer -> {Block} -> String
ppPKs'' \{\} = ""
ppPKs' n (b ∪ bs) = "a*n + n + "+ b + " + ppPKs' n bs
```

For our running example, the OCL code generated for the `Pilots` class, and that guarantees the safe use of a primary key value on the pilots identification, is shown next:

```plaintext
context Pilots
    inv pkPilots : Pilots.allInstances->forall(a1, a2 | a1 <> a2 implies a1.id <> a2.id)
```

The functions to generate code for foreign key validation are in all similar to the ones shown above. For this reason, we refrain from showing the complete definitions of the functions that implement it. In essence, we define a function that produces lists of pairs, each one containing the label of the class being affected by the constraint and a triple of strings given by an auxiliary function `pk`. The definition of `pk` for the remaining cases follows the definition of `pk`.
Each generated element contains the name of the attributes being defined as a foreign key, k, the name of the class being referenced, n, and the attribute in that class, a.

The generated OCL code that implements the foreign plane key validation in the PlanesKey class is as follows.

```
context PlanesKey
inv fkPlanesKey : Planes.allInstances->
exists(a | a.n-number = planesKey.planes_key)
```

5. RELATED WORK

The work presented in this paper has strong connections with [7]. Indeed, in that paper the authors have proposed the ClassSheet modelling language for spreadsheets and used UML to witness the class structure of ClassSheets. Our studies also involve ClassSheets and UML, but several aspects distinguish our paper and [7]. Firstly, we have taken the original ClassSheet language of [7] and extended it with both syntactic constructions and deeper semantics. Namely, we have focused on spreadsheets that can be viewed as relational databases and, for such spreadsheets, our extended modelling language offers more correctness guarantees on the derived spreadsheets than the original one. Secondly, the UML diagrams presented in [7] are given, on concrete examples, just as illustrations of ClassSheet properties. The diagrams shown are quite simple and one can easily see the relation to particular ClassSheets. However, the purpose of [7] was not to obtain them automatically from ClassSheets models. On the contrary, in this paper we have formalised a transformation from the extended ClassSheet language to UML. The generation of OCL to model properties consists of a final distinction between our work and [7].

The derivation of UML class diagrams with OCL constraints from ClassSheets opens a wide range of validation possibilities for spreadsheets. Concrete validation techniques are outside the scope of this paper, but we should point as interesting the combination of our work with some others: [8] offers an animation based approach for the validation of UML models and OCL constraints; [4] provides a validation environment for UML that checks consistency, completeness and dependability requirements. We believe these and other techniques can cooperate to further reduce the still alarming number of errors in spreadsheets.

6. CONCLUSIONS

In this paper we have extended ClassSheets with characteristics from relational databases. This means that we are now able of specifying database oriented spreadsheets where it is possible to define primary and foreign keys.

We have also shown how to systematically transform an extended ClassSheet into a UML class diagram (classes, associations and OCL restrictions). UML class diagrams are generated under the notation of the USE framework, and thus we allow for immediate verification of the spreadsheet model. This framework allows one to create instances of the given UML model and verify the OCL constraints on those objects. Moreover, it is also possible to define pre and post-conditions on operations. From this, it must also be possible to generate spreadsheet macros/formulas to ensure the constraints. This is an aspect that we have not fully explored yet, but that we plan to do in the future.

We have implemented our transformation in the Haskell programming language under the HAExcel framework. The tests we have run show, for example, that we generate the same UML models shown in [7].

7. REFERENCES


