Journal of Software Engineering: Theories and Practices 1 (1): 10-18, 2016
http://sispress.org/journals/josetap
ISSN: 2377-3316
http://dx.doi.org/10.21174/josetap.v1i1.31

Sanford Inter Science Press

# Toward a Timetabling Qt Application Modernization

Yassir Gourram, Samir Mbarki[*], Zineb Gotti and Sara Gotti

*MISC laboratory, Faculty of Science, Ibn Tofail University, BP 133, Kenitra, Morroco*

***Abstract*--** Nowadays, we note that there is a big change in technologies. To track these changes, the evolution of software development practices is required. MDE provides modernization techniques that can quickly follow these changes. We adopt this approach in order to modernize the legacy Qt timetabling application user interfaces. We conducted an automatic reverse engineering of Qt interfaces in order to have as output RIA interfaces. This automatic process reproduces user interfaces with a modern representation and retains data related to graphical components namely properties, position and actions. To illustrate this approach, let us consider FET as our legacy Qt timetabling application.

***Keywords*—** Architecture Driven Modernization (ADM); Knowledge Discovery Model (KDM); Qt Graphical User Interface Meta-model (QGUIM); Concrete Syntax Tree Meta-model (CSTM); Rich Internet Application (RIA).

## I. INTRODUCTION

Once the software world becomes a primary need for companies, new techniques appear under several ways or representations with approaches varying in several areas. To take advantage of the benefits introduced by trendy platforms, the OMG group has introduced several approaches such as MDA in 2000 [1] and ADM [5] in 2007 to build and promote standards that can be applied in reverse engineering process.

The technologies of these approaches based on meta-modeling and transformations of models can help to optimize systems evaluations costs by automating modernization process of systems. Reverse Engineering Technologies can analyze legacy software system, identify its widgets and their interconnection, reproduce it or reproduce anything based on the extracted information, and create a representation at a higher level of abstraction or in another form [12].

In this paper, we applied modernization technologies for migrating from a Qt application to RIA application by implementing a re-engineering process based on three phases: The reverse engineering, the restructuration and the forward engineering. An approaching tool is proposed, in this paper, for modernizing FET application; it allows extracting domain classes according to CSTM meta-model, and semantic graphical information, then analysing extracted information to change them into a higher level of abstraction as a KDM model. This tool provides a modernization process which starts from a timetabling application based on Qt API, as source platform, in order to produce a modern timetabling application based on RIA as a target platform.

It was difficult to define mapping between source platform layout and target platform layout, so we implement an algorithm aiming at calculating the absolute position of each widget.

The rest of this paper is organized as follows: Section 2 is dedicated to the related work. In section 3, we explain the modernization process which contains three phases: Reverse engineering, Restructuring and Forward engineering. Section 4 defines the used technologies. Finally, section 5 concludes the work and presents perspectives.

## II. RELATED WORK

Currently, software modernization approach becomes a necessity for creating new business value from legacy applications, there is a great research effort has been dedicated to the reverse engineering and a number of proposals have been published. The most relevant are [12], [13], [14], [15], [16]. The work proposed by Javier et al [13], focuses on text to model transformation as the essential task in modernization in order to extract models from GPL source code that conforms to a grammar. They proposed a

[*] Corresponding author can be contacted via the journal website.

transformation language whose source domain is the grammar of the source code and whose target domain is the model by manipulating the CST of the source code. R. Pérez-Castillo et al [14] introduce a reverse engineering tool called ANDRIU for migrating Java/swing applications to the android platform. This tool uses two OMG standards; AST for representing data extracted from java swing code in reverse engineering phase and KDM Platform Independent Model. In [15], Roberto Rodriguez-Echeverria et al presented a solution for the modernization of JavaEE applications. The authors have set up a systematic process for WA-to-RIA modernization by applying MDE principles, techniques and tools. The process generates a RIA client from the legacy WA presentation and navigation layers and its corresponding service-oriented connection layer with the underlying business logic at server side. In [16], Yan Liang presented a tool for reverse engineering C++ source code. This tool uses CDT API (to extract artefacts from C++ source code without supporting the user interfaces) and Eclipse EMF (to facilitate the design and realization of the target model). Mbarki, S. et al [12] present a tool for reverse engineering named FlexMigration allowing automatic reverse engineering of Swing GUI to obtain a RIA GUI. This tool is based on three phases; the reverse engineering phase which uses the jdt API for parsing the java Swing code in order to fill in an AST and Graphical User Interface models, the restructuring phase that represents a model transformation for generating an abstract KDM model and the forward phase which includes the elaboration of the target model (FlexM) and a similar Flex Graphical User Interface.

## III. MODERNIZATION PROCESS

The first phase of modernization process is based on Text-To-Model transformation that uses parsing techniques to retrieve information collected from HEADERS, SOURCES and UIS files using algorithms designed for data extraction; graphical components, widget position and related signals and slots used for communication between components. Then, a model transformation is launched to automate the generating process of RIA application model. Our approach includes reverse engineering techniques and OMG standards: ADM, KDM and MDA.

FET timetabling application [11] is perfectly good to be used in our case study as input for our approach. Basically, FET is open source free timetabling software for automatically generating schools timetables. It is licensed under the GNU Affero General Public License version 3 or later. FET application is written in C++ using the Qt cross-platform application framework. Each Qt application is based on three main files: The headers, sources and forms files.

The UI (forms) files define a set of graphical components and their properties. As regards source files, they contain all the definitions of the class which declaration is in the headers.

Our approach is based on the analysis of each of these files in order to have an abstract presentation of the original interface and then rebuild it in a rich target platform. Here in figure 1 the diagram detailing our process of C ++ / Qt applications modernization [2]:

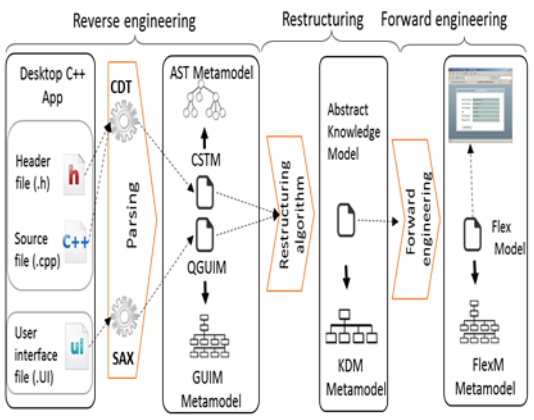


**Figure 1.** Modernization process

### A. *Reverse Engineering Phase*

This section is dedicated to the extraction and representation of information. It defines the first phase of reengineering following ADM process. It is about the parsing technique, consisting in extracting input data information from Qt files and representing them in the form of a model. This is the solution we approached to perform the transformation text to model.

The definitions of Qt classes are distributed in three separate files; a header .h file, .cpp source file and .ui file. In order to extract the necessary information from the three Qt files, we used two open source parsers, an XML parser to parse the .ui file that has the form of xml, and another C++ parser to analyze at the same time header and source files which conform to a grammar definition.

Before we explain the parsing algorithm, it should be noted that two PSM meta-models have been developed. The first focuses on the structural aspect of header and source files (CSTM). The second is particularly devoted to the presentation layer of the application (QGUIM).

As depicted in figure 1, the first parsing relies on UIs component extraction from .ui files, the second one relies on the concrete syntax tree nodes of .cpp and .h files for defining SLOT function definition related to each graphical component.

### 1) *Extracting Graphical Components*

The SAX API is originally specific to the Java programming language that is used to read a portion or all of an XML document. It could then be adopted by most of the current programming languages [18]. This API allows us visiting the input UI file, element by element (tag, comment, text), when it is encountered, in order to get a QGUI model. Our sax parser in reality consists in 2 parsing processes: the first is parsing UI files to get QGUI model that contain all interface granularities. The second is calculating all absolute position of component existing in layouts.

#### *a) First UI Parsing*

This reverse engineering converts all UI file element to get model according to QGUI meta-model (see fig.2). The sax algorithm (see table 1) processes component, container and classifies them in respective position in a temporary QGUI model.

This first level SAX parser is a handler class that can create several types of events. It must be defined as needed for our parsing treatment. The main methods to answer are: EndElement, characters, startDocument, endDocument, startElement. StartElement method is an important method; it is called upon detection of a start component and containers tags. It is called after startDocument method launching; it manages UI object creation when meeting a start tag of a new item. This generated model have an important cons, it generates component without any position information as X and Y, Height or Width. These properties have an important influence on the destination model, that's why we create a second parsing to calculate these informations.

#### *b) Second UI parsing*

The goal of this treatment is to save the traceability of the absolute position calculated from the first source model to the last target model, it will save multiple inconsistency between the different types of layout in various target platforms, for example, a Qt Form layout has no similar in android. Thus, we can easily manipulate our target model with the absolute position in multiple platforms as desktop, web or mobile.

This parsing is a collection of small algorithms that may calculate dimensions of our widgets; these algorithms are based on geometry elements information. Here (see fig.3) we take an example to explain our algorithm.

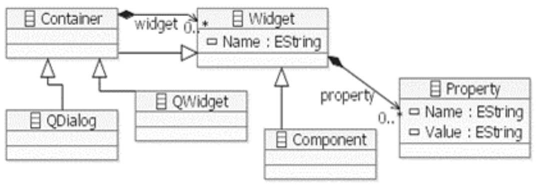


**Figure 2.** QGUI meta-model

**Table 1.** A part of first parsing algorithm

**First Parsing algorithm**

```
If (currentTag = "width") then
Create "<property Name <- "width" Value="
End if
If (CurrentTag = "property" and AttributesNumber
(0) = ("spacing")) then
Create "spacing = ""
currentpropertylayout<- true
End if
If (CurrentTag = "property" and AttributesNumber
(0) = ("leftMargin")) then
Create" leftMargin = ""
currentpropertylayout<- true
End if
If (CurrentTag = "widget") then
If (AttributesNumber (0) = ("QSpinBox")) then
Currentspinbox<-true
Create "<widget xsi:type="guiqt1:QSpinBox"
Name="AttributesNumber(1)"
Else
Create "<widget xsi:type = "guiqt1:
AttributesNumber(0)" Name="AttributesNumber(1)"
End if
End if
If (CurrentTag = "layout") then
Create "<layout xsi:type =
"guiqt1:AttributesNumber(0)"
Name ="AttributesNumber(1) "
layoutopened<- true
End if
```

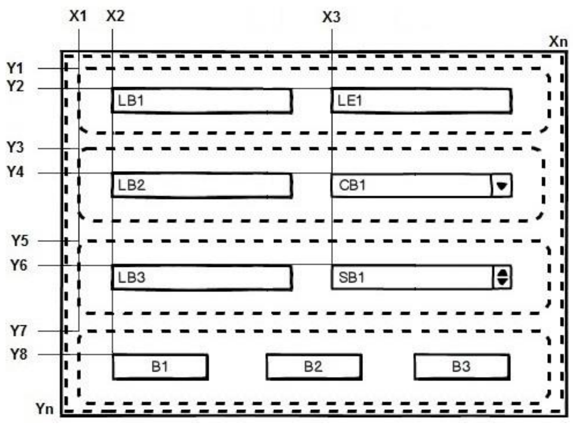


**Figure 3.** GUI case study

This window represents a QDialog that contains a vertical layout container which contains also four small horizontal layouts, each of these layouts has two or three components. The only geometry information here is the QDialog properties that give X, Y, Width and Height. In the first phase of the algorithm, we try to calculate the X,Y of each layout by calculating how many layouts are in to extract the absolute position of all components and containers on them. Thus, the algorithm tries to divide in equal values the height of our layouts and calculate the width.

$$\text{Layoutheight} = (\text{Qdialogheight} - (\text{layoutnumber} + 1) * \text{spacing}) / \text{layoutnumber} \qquad (1)$$

$$\text{Layoutwidth} = \text{QdialogWidth} - 2 * \text{spacing} \qquad (2)$$

With these values we can easily extract layout's X, Y position.

$$X1 = X0 + \text{spacing} \qquad (3)$$

$$Y1 = Y0 + \text{spacing} \qquad (4)$$

$$Yn = Yn\text{-}1 + \text{layoutheight} + \text{spacing} \quad n>1 \qquad (5)$$

The complete equations considerate different various layouts parameters, in our case we choose spacing as example, in fact, for each layout, parameters are not completely similar (see table 2).

After calculating layout position, the program starts with analysing each layout, how many components are in and starts calculating the absolute position of each element. The absolute position equation for LB2 and CB1:

$$\text{Componentwidth} = (\text{layoutwidth}(\text{NumberOfElementInLayout} + 1)* \text{spacing}) / \text{NumberOfElementInLayout} \qquad (6)$$

$$X2 = X1 + \text{spacing} \qquad (7)$$

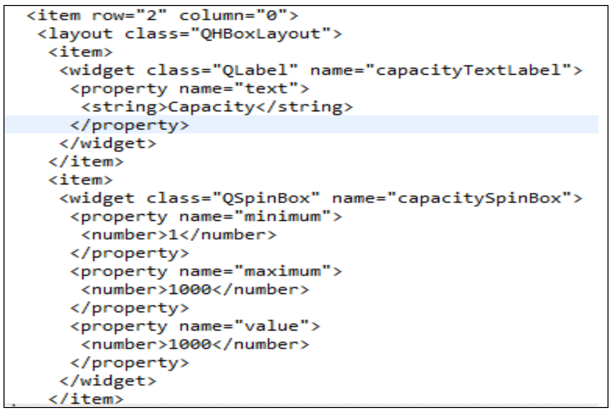
$$Xn = Xn\text{-}1 + \text{Componentwidth} + \text{spacing} \quad X>2 \qquad (8)$$

$$Y4 = \text{layoutheight}/2 - \text{Height LB2}/2 + Y3 \qquad (9)$$

The component height in our case has a default value. In case of layout imbrications, the algorithm above applies recursively. The figure bellow (see fig.4) shows an UI file as entry of our parsing algorithm:

**Table 2.** Differences between layouts properties

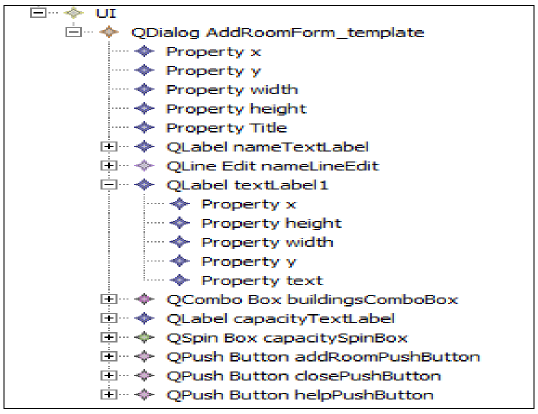
| | **QHBox Layout** | **QVBox Layout** | **QGrid Layout** | **QForm Layout** |
|---|---|---|---|---|
| layoutName | * | * | * | * |
| layoutLeftMargin | * | * | * | * |
| layoutTopMargin | * | * | * | * |
| layoutRightMargin | * | * | * | * |
| layoutBottomMargin | * | * | * | * |
| layoutSpacing | * | * | | |
| layoutStretch | * | * | | |
| layoutSizeConstraint | * | * | * | * |
| layoutRowStretch | | | * | |
| layoutColumnStretch | | | * | |
| layoutHorizontalSpacing | | | * | * |
| layoutVerticalSpacing | | | * | * |
| layoutRowMinimumHeight | | | * | |
| layoutColumnMinimumHeight | | | * | |
| layoutFieldGrowthPolicy | | | | * |
| layoutRowWrapPolicy | | | | * |
| layoutLabelAlignmentHorizontal | | | | * |
| layoutLabelAlignmentVertical | | | | * |
| layoutFormAlignmentHorizontal | | | | * |
| layoutFormAlignmentVertical | | | | * |



```
<item row="2" column="0">
 <layout class="QHBoxLayout">
  <item>
   <widget class="QLabel" name="capacityTextLabel">
    <property name="text">
     <string>Capacity</string>
    </property>
   </widget>
  </item>
  <item>
   <widget class="QSpinBox" name="capacitySpinBox">
    <property name="minimum">
     <number>1</number>
    </property>
    <property name="maximum">
     <number>1000</number>
    </property>
    <property name="value">
     <number>1000</number>
    </property>
   </widget>
  </item>
```

**Figure 4.** UI file entry

The result of the parsing algorithm is explained as follows (see fig.5):



```
UI
  QDialog AddRoomForm_template
    Property x
    Property y
    Property width
    Property height
    Property Title
    QLabel nameTextLabel
    QLine Edit nameLineEdit
    QLabel textLabel1
      Property x
      Property height
      Property width
      Property y
      Property text
    QCombo Box buildingsComboBox
    QLabel capacityTextLabel
    QSpin Box capacitySpinBox
    QPush Button addRoomPushButton
    QPush Button closePushButton
    QPush Button helpPushButton
```

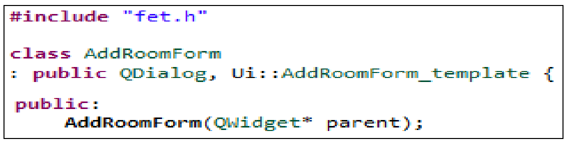
**Figure 5.** Generated model with the second parsing phase

*2) Extracting Concrete Syntax Tree Nodes*

As shown in fig.1, the execution of model to text transformation generates a target model conforming to CSTM meta-model representing the syntax tree of the source code. The based technique for model extraction is to use a parser that provides parsing and model generation. Regarding the choice of C++ open source parser, we opted for the CDT parser, here [3] we found the reasons for choosing CDT parser to extract information from both header and source files in order to represent SLOT function definitions.

The text to model transformation using CDT parser uses node visiting technique, it starts by creating the syntax tree of header file, then visiting node by node for filling in the instance model that conforms to CSTM meta-model. The root node is TranslationUnit, it represents a compilable unit of source code. The TranslationUnit node contains Includedirectives and Declarations, these Declarations contain Declaration-Specifier: an element for representing class type nodes if it is a CompositeTypeSpecifier. They also contain Declarators for representing field type and function type node.

The figures below describe the mapping between a part of AddRoomForm header (see fig.6) code and the model instance result (see fig.7) of the parsing according to a CSTM meta-model (see fig.8).

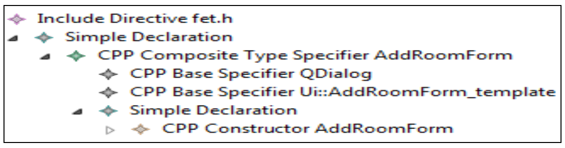


```
#include "fet.h"

class AddRoomForm
: public QDialog, Ui::AddRoomForm_template {

 public:
     AddRoomForm(QWidget* parent);
```

**Figure 6.** Legacy source code



```
Include Directive fet.h
Simple Declaration
  CPP Composite Type Specifier AddRoomForm
    CPP Base Specifier QDialog
    CPP Base Specifier Ui::AddRoomForm_template
    Simple Declaration
      CPP Constructor AddRoomForm
```

**Figure 7.** Model instance

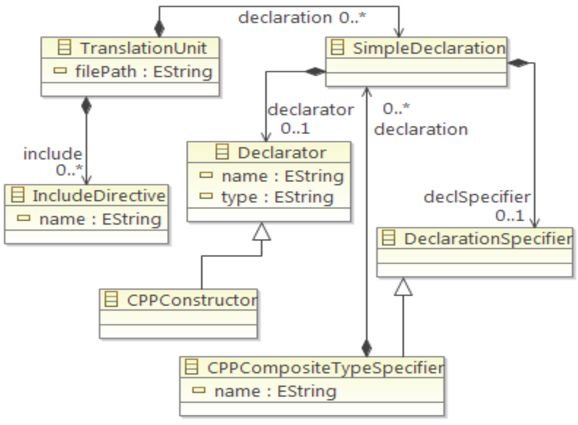


**Figure 8.** CSTM meta-model

Now, for each CPPfunction (SLOT) we should get its definition by parsing the source file (.cpp) and visiting the functionDefinitons. The function definition is located in a block called CompoundStatement which covers inside a set of statements. A statement can be either a FunctionCallExpression if it is a call of function, a BinaryExpression if it is an assignment, or an IfStatement.

We define a parsing recursive function algorithm that analyzes the source file and gets statements (see table 3).

The first part of the figure below (see fig.9) shows the body definition code of addRoom SLOT and the second part represents the result of the relative function definition node parsing.

**Table 3.** CDT parsing algorithm

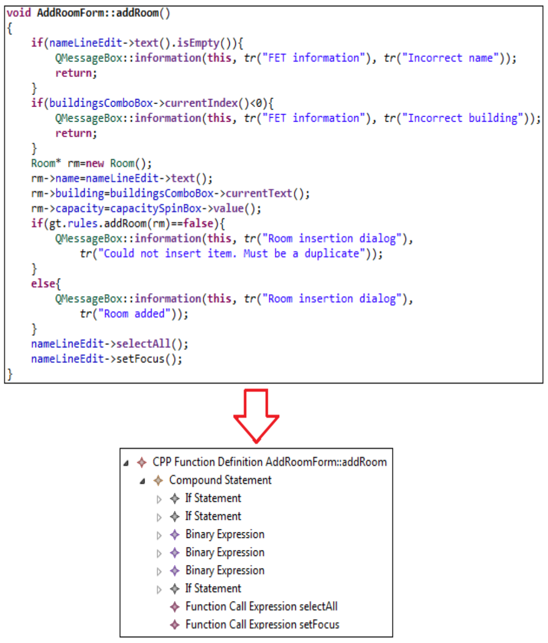
**Visiting blockstatements(Body) function Algorithm**

```
Begin
for all statement inbody do
if(statement isFunctionCallExpression) then
FunctionCallExpression.Create()
add(FunctionCallExpression, body)
end if
if(statement isBinaryExpression) then
BinaryExpression.Create()
add(BinaryExpression, body)
end if
if(statement isIfStatement) then
IfStatement.Create()
forall condition instatement.getConditionExpression() do
if(condition isFunctionCallExpression) then
functionCallExpressionCondition.create()
end if
if(condition isBinaryExpression) then
BinaryExpressionCondition.create()
end if
end for
body ←statement.getThenStatetment()
Visiting blockstatements(body)
Body ←statement.getElseStatetment()
Visiting blockstatements(body)
add(IfStatement, body)
end if
end for
end
```

In order to get dependencies between Qt windows, our parser has used a navigability algorithm that searches in each SLOT function if there is a FunctionCallExpression whose name is show() or exec() that can launch a window from another one.

## B. *Restructuring Phase*

This is the important phase of the modernization process. It is aiming at deriving an enriched conceptual technology-independent specification of the legacy system in a knowledge model KDM from the information stored inside the models generated on the previous phase.



```
void AddRoomForm::addRoom()
{
    if(nameLineEdit->text().isEmpty()){
        QMessageBox::information(this, tr("FET information"), tr("Incorrect name"));
        return;
    }
    if(buildingsComboBox->currentIndex()<0){
        QMessageBox::information(this, tr("FET information"), tr("Incorrect building"));
        return;
    }
    Room* rm=new Room();
    rm->name=nameLineEdit->text();
    rm->building=buildingsComboBox->currentText();
    rm->capacity=capacitySpinBox->value();
    if(gt.rules.addRoom(rm)==false){
        QMessageBox::information(this, tr("Room insertion dialog"),
            tr("Could not insert item. Must be a duplicate"));
    }
    else{
        QMessageBox::information(this, tr("Room insertion dialog"),
            tr("Room added"));
    }
    nameLineEdit->selectAll();
    nameLineEdit->setFocus();
}
```

```
CPP Function Definition AddRoomForm::addRoom
  Compound Statement
    If Statement
    If Statement
    Binary Expression
    Binary Expression
    Binary Expression
    If Statement
    Function Call Expression selectAll
    Function Call Expression setFocus
```

**Figure 9.** CSTM model

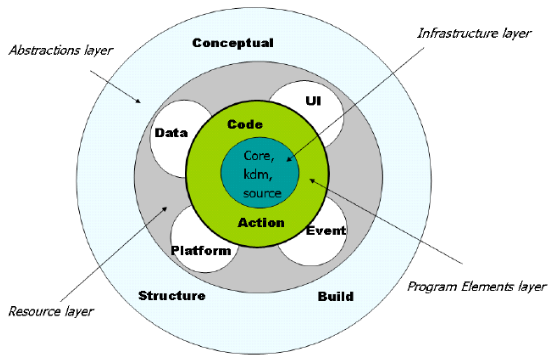


**Figure 10.** KDM architecture [6]

KDM is an OMG standard for representing information retrieved by reverse engineering phase [7]. The KDM architecture is divided into four abstraction layers: Infrastructure layer, Program elements layer, Resource layer and Abstractions layer (see fig.10). Each layer is dedicated to a particular application view point. In this paper, we are interested in only two layers: infrastructure and resource. These layers allow representing user interfaces, data and source code.

We analysed the KDM OMG standard and it's meta-model to select the packages needed in this phase. We select ui package and kdm package. The ui package is used to represent the user interfaces structure and interactions among them. The kdm package is the infrastructure for the ui package. In order to link the graphical element represented in QGUI model with there graphical properties, we incorporated the Attribute (tag-value) element in the kdm package. As shown in figure 1, after obtaining CST and QGUI models from Qt files, we define a set of mapping rules that transform these models to a kdm model. This mapping distinguishes two concerns: ui elements mapping and SLOT mapping. The first mapping allows transforming graphical containers and components into KDMUI element.

The containers (e.g. QDialog) are mapped to Screens. Concerning AbstructButton element (QpushButton, QRadioButton, QCheckBox) are transformed into UIRessource element. The DisplayWidget elements (QLabel) are transformed into UIField element. The InputWidget elements (QTextEdit, QSpinBox, QComboBox, QLineEdit) are transformed into UIField.

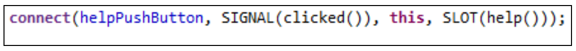
The QGUI model contains graphical component descriptions (such as text, size, position …); these descriptions are stored in a property element which will be converted to Attribute.

The second mapping starts when a QGUI element is used to interact with another tier of system. These interactions are handled by SIGNALs and SLOTs. SIGNALs are mapped into UIEvent and SLOTs are mapped into UIAction. A SLOT represents a CST FunctionDefinition that is connected to a graphical component if a SIGNAL is emitted. Each CST statement defined inside a SLOT function is mapped as follows:

FunctionCallExpression to CallableUnit

IFStatetment to ActionElement

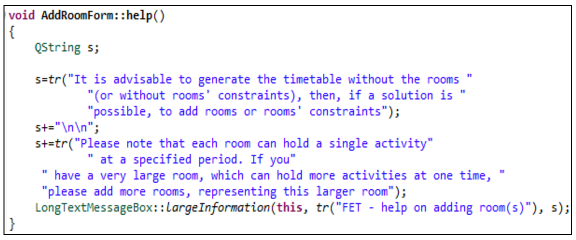
BinaryExpression to ActionElement



```
connect(helpPushButton, SIGNAL(clicked()), this, SLOT(help()));
```

**Figure 11.** Connecting a SIGNAL to a SLOT

Figure 11 represents how to connect a SIGNAL emitted from a graphical component to a SLOT function. Figure 12 shows the body definition of help SLOT inside AddRoomForm class.

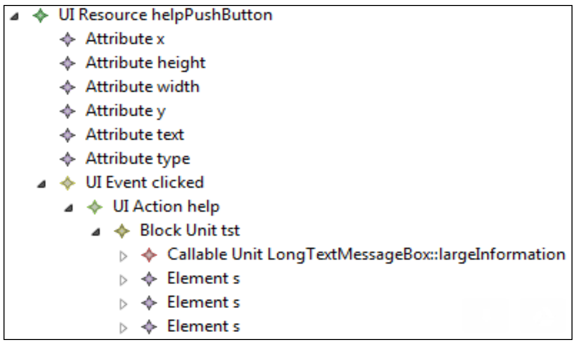


```
void AddRoomForm::help()
{
    QString s;

    s=tr("It is advisable to generate the timetable without the rooms "
            "(or without rooms' constraints), then, if a solution is "
            "possible, to add rooms or rooms' constraints");
    s+="\n\n";
    s+=tr("Please note that each room can hold a single activity"
            " at a specified period. If you"
     " have a very large room, which can hold more activities at one time, "
     "please add more rooms, representing this larger room");
    LongTextMessageBox::largeInformation(this, tr("FET - help on adding room(s)"), s);
}
```

**Figure 12.** SLOT body definition

The two mappings generate as a result a KDM model. Figure 13 shows a part of transformation execution result.



```
UI Resource helpPushButton
    Attribute x
    Attribute height
    Attribute width
    Attribute y
    Attribute text
    Attribute type
    UI Event clicked
        UI Action help
            Block Unit tst
                Callable Unit LongTextMessageBox::largeInformation
                Element s
                Element s
                Element s
```

**Figure 13.** KDMUI model

**Table 4.** KDM to FlexM transformation algorithm

```
KDM to Flex transformation algorithm
    inputkdm : KDMui
    output flex : FLEXM
    begin
        for all e in kdm.UiResource
        map Uiresource2FelxModel (e)
         end for
    end
    mapping Uiresource2FelxModel (res:
UiResource):FlexModel
    begin
            for all e.typein res.UiElement
              if e.typeis Screen
              map screen2DataGroup(e)
              end if
            end for
    end
    mapping screen2DataGroup(s:Screen): BorderContainer
    begin
        if s.typeis Frame
            forall e in s.UiElement
            if e.typeis UiResource
            if e.typeis Qcombobox
            map UiResource2combobox (e)
            end if
            if e.typeis QSpinbox
            map UiResource2combobox (e)
            end if
            if e.typeis QPushButton
            map UiResource2Button (e)
            end if
            end if
            if e.typeis Uifield
            if e.typeis QLAbel
            map Uifield2label (e)
            end if
            if e.typeis QlineEdit
            map Uifield2TextInput(e)
            end if
            end if
            end for
        end if
    end
    mapping atrribute2Properties(A:Attribute): Property
    begin
        name □A.tag
        value □A.value
    end
```

Concerning the dependency mapping between two windows, the model transformation has defined a navigability mapping that use UIFlow elements added to UIAction to define the relationship between the two windows.

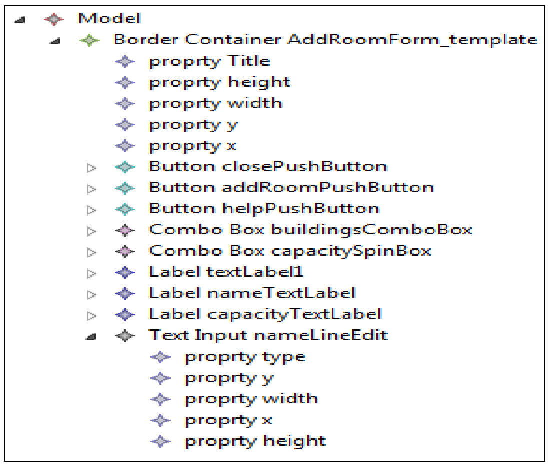
C. Forward Phase

Forward engineering is a process of moving from high-level abstractions [10]; it involves using transformational techniques to automatically obtain source code according to the new platform or programming language.

Once the information retrieved from the user interface is integrated into the KDM repository, it can be used for migrating the KDM UI model to RIA model with Flex technology.

In this step we focused on the mapping of graphical components and their properties (Text, width, Position ...) according to mapping rules. In the transformation algorithm (see table 4), we have collected all the properties related to each graphical component into Flex model.

The figure 14 depicts a part of the model obtained from KDM to Flex model transformation.



```
Model
  Border Container AddRoomForm_template
    proprty Title
    proprty height
    proprty width
    proprty y
    proprty x
    Button closePushButton
    Button addRoomPushButton
    Button helpPushButton
    Combo Box buildingsComboBox
    Combo Box capacitySpinBox
    Label textLabel1
    Label nameTextLabel
    Label capacityTextLabel
    Text Input nameLineEdit
      proprty type
      proprty y
      proprty width
      proprty x
      proprty height
```

**Figure 14.** Flex model result

To keep the same appearance of the legacy timetabling interfaces, we made a model to text transformation that take as input the generated Flex model. The figures below show two graphical interfaces. Figure 15 represents the Qt timetabling interface for adding rooms that will be transformed into a similar Flex user interface as shown in figure 16.

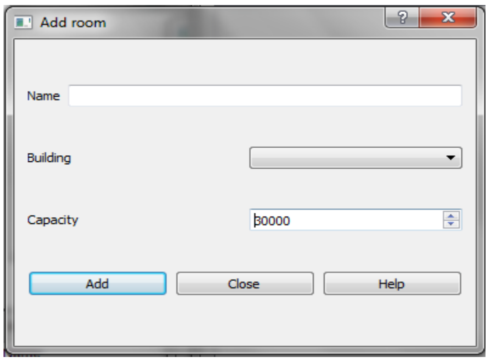


**Figure 15.** Legacy AddRoom interface

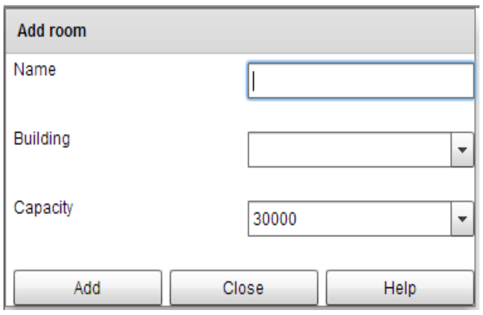


**Figure 16.** Modern Addroom Flex interface

## IV. TECHNOLOGIES USED

This tool has been developed for existing timetabling application based on Qt / C++. It is based on five key technologies. The first technology is the CDT [4] parser which provides techniques of how to inspect a concrete syntax tree and how to walk its nodes. The second one is developed using SAX API, it works by iterating over the XML file for extracting graphical components, their properties and their interconnection.

The CDT and SAX parser are used for developing the first phase of modernization process.

The third technology is EMF (Eclipse Modeling Framework), "It represents a modeling framework and code generation facility for building tools and applications based on other structured data models" [9]. It can build meta-models according to the meta-model ECORE. Then, from these meta-models, we build models that conform to their syntax.

And as a fourth technology used for mapping between the models, we used the QVTo framework [8].

Finally the fifth technology is based on Acceleo "Acceleo is a pragmatic implementation of the Object Management Group (OMG) MOF Model to Text Language (MTL) standard" [17], in order to generate the code with riche interface.

We worked on 60% of application classes, among classes that we have not processed, are the treatment classes as "timeconstraint", "generate" ... In the other side the SAX algorithm has limitation when a graphical component or a container within the layout has a specific height and width e.g. “QtabWidget, QListWidget …”

## V. Conclusion and Future Work

Ultimately, the final result of our approach shows that the designed solution can be used to reproduce modern and rich interfaces from Qt timetabling applications, while keeping the arrangement of different graphical components and related signals and slots associated to each component. To achieve this, we relied on MDE principles. Our contribution bases on a set of transformations, text-to-model transformation (parsing), model-to-model transformation and model-to-text transformation. This involves a series of meta-models (data structures) and transformations (algorithms).To switch from one representation to another, Meta-models are considered to be a useful formalism to represent the gained knowledge in the reverse engineering process. QGUIM and CSTM as well as KDM and FlexM meta-models are given as the key of our migration approach.

However, the transformation algorithms do not treat all cases of layouts. The advantage we have is that our tool is extensible; eventually we can modernize a solution for the migration of different graphical components and different layouts. Also we can modernize the entire Qt application, not only GUIS but also the business rules. The advantage is that it is designed to be extended to accept other target platforms for migration.

## References

[1] Blanc, Xavier, and Olivier Salvatori. *MDA en action: Ingénierie logicielle guidée par les modèles*. Editions Eyrolles, 2011.

[2] Chikofsky, Elliot J., and James H. Cross. "Reverse engineering and design recovery: A taxonomy." *Software, IEEE* 7, no. 1 (1990): 13-17. DOI= http://dx.doi.org/10.1109/52.43044

[3] Piatov, Danila, Andrea Janes, Alberto Sillitti, and Giancarlo Succi. "Using the Eclipse C/C++ development tooling as a robust, fully functional, actively maintained, open source C++ parser." In *Open Source Systems: Long-Term Sustainability*, pp. 399-399. Springer Berlin Heidelberg, 2012. DOI= http://dx.doi.org/10.1007/978-3-642-33442-9_45

[4] CDT, Eclipse C/C++ development tools, viewed December 2014. https://eclipse.org/cdt.

[5] OMG, Architecture-Driven Modernization, viewed January 2014. http://adm.omg.org.

[6] OMG, Architecture-Driven Modernization: Knowledge Discovery Meta-Model, v1.1, viewed February 2014. http://www.omg.org/spec/KDM/1.1/PDF/2009.

[7] OMG, Architecture-Driven Modernization Standards Roadmap, viewed March 2014. http://adm.omg.org/ADMTF%20Roadmap.pdf.

[8] OMG, QVT. Meta Object Facility 2.0, Query/View/Transformation Specification, viewed June 2014. http://www.omg.org/spec/QVT/1.0/PDF/.

[9] EMF, Eclipse Modeling Framework, viewed September 2014. http://eclipse.org/modeling/emf/.

[10] Wagner, Christian. Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems. Springer Science & Business Media, 2014. DOI= http://dx.doi.org/10.1007/978-3-658-05270-6

[11] FET, Free Timetabling Software, viewed September 2014. http://lalescu.ro/liviu/fet/.

[12] Mbarki, S., Laaz, N., Gotti, S., Gotti, Z., “ADM-Based Migration from JAVA Swing to RIA Applications”, In *5th International Conference on Information Systems and Technologies (ICIST)*, Istanbul, Turkey, march 21 - 23, 2015.

[13] Izquierdo, Javier Luis Cánovas, and Jesus Garcia Molina. "Extracting models from source code in software modernization." *Software & Systems Modeling* 13, no. 2 (2014): 713-734. DOI= http://dx.doi.org/10.1007/s10270-012-0270-z

[14] Pérez-Castillo, Ricardo, De Guzmán, I.G.R., Gómez-Cornejo, R., Fernandez-Ropero, M., Piattini, M., “ANDRIU. A Technique for Migrating Graphical User Interfaces to Android”, In: *Twenty-Fifth International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 516-519, Boston. 2013.

[15] Rodriguez-Echverria, R., Conejero, J.M., Clemente, P.J., Preciado, J.C., Sánchez-Figueroa, F., “Modernization of Legacy Web Applications into Rich Internet Applications”, In: *Andreas Harth and Nora Koch, editors, Current Trends in Web Engineering - 11th International Conference on Web Engineering - ICWE 2011 Workshops*, Cyprus, 2011.

[16] Liang, Yan. "On the Exploration of Lightweight Reverse Engineering Tool Development for C++ Programs." In *Proc. International Conference on Software Engineering Research and Practice*. 2011.

[17] Acceleo, viewed September 2014. https://eclipse.org/acceleo/.

[18] SAX 2.0: The Simple API for XML, viewed September 2014. http://www.saxproject.org/.