GUI application meta-models: a state of the art

Benoît Verhaeghe[1,2], Anne Etien[1],
Nicolas Anquetil[1], Stéphane Ducasse[1]
[1]Université de Lille, CNRS, Inria,
Centrale Lille, UMR 9189 – CRIStAL, France

Abderrahmane Seriai[2], Laurent Deruelle[2],
Mustapha Derras[2]
[2]Berger-Levrault, France

***Abstract*— In this context... When a developer wants to analyse an application, and this application includes a user interface. He could want an abstraction of the application. Often, this abstraction level correspond to models, and their meta-models, of the piece of software. We consider this problem P... There are indeed many elements to represent and to link. P is a problem because... Currently there are many meta-models of GUI application that can be used to represent the interface and some links between the different windows. But none of them express complex behavior, such as loop or condition, nor data structure information. We propose this solution... We defined four meta-models. The first one represent the Graphical User Interface, the second one the layout to apply to the GUI, the third one the data structure implies in the GUI, the last one the behavior associate to an event fired by an element of the GUI. Our solution solves P in such and such way. Our meta-models can express the different elements of a GUI application. So can represent the graphical user interface and the logic of the application.**

***Index Terms*—Graphical User Interfaces, Model-Driven Engineering**

I. Introduction

Contexte use case In the context of the analysis of an application. It happens that the developers want to create tools on top of his analysis. These tools could be useful in cases like: analysis, tests generation, migration, *etc.*

introduction model A way to create those software is the usage of model of the source code of the application. The developers create or use a meta-model of the language of the application source code. Then they instantiate this meta-model from the application to analysed.

Problème intro gui In the case of GUI application, it happens that the abstraction level does not provide enough information. Indeed, the generated model contains the methods, classes, *etc.* of the source application, but no information about how the GUI is shaped. The developers must do another analysis on the model to extract these information and so making his tool.

intro simple gui decomposition A solution to create tool specialized for GUI application is to create GUI model. A GUI application is divided into different elements. The aim of this paper is to define those components and meta-models to represent all the specificities linked to a GUI application.

Know tracks We did not find any other papers which defines a solution to represent graphical application. Nevertheless, the KDM model designed by the OMG proposed a *Resource Layer* which can used to define an GUI application. Their solution is discussed section X.

What is our solution We defined four meta-models to represent the GUI software. The meta-models represent the different main GUI's specificities we extracted from our analysis and other research papers.

Contrib of the paper The main contributions of our work are:

- Description of GUI application structure
- Meta-Models to represent a GUI application
- Discussion about GUI Meta-Models

Paper structure In Section II, we present the different GUI elements Section III exposes our solution. Then, In Section IV, we describe and categorized the solution proposed by others authors. Finally, we conclude in Section VII.

II. GUI application structure

The first step to create the meta-models of a graphic application is to define the elements we need to represent. We divided the UI into the following three parts:

- The user interface
- The business code
- The behavior code

A. User Interface

The user interface is the viewable part. This element represents the interface of the application. It includes the components of the interface. The User Interface does not contain the exact visualisation of a component. But it can precise some feature inherent to the component, like the ability to be clicked, or some properties of the component, like its color or size. More than the components, it also includes the disposition of those components in relation to the others. In the case a application is composed by multiple windows (or web pages for web application), the user interface contains all the windows.

B. Behavior Code

The behavior code is the *executable* part of the application. It corresponds to the logic of the application. It can have two manifestations of the business code. It can be run either by an user action on an interface component (like a click) or by the

system itself. As a programming language, the business code contains control structures (*i.e.* loop and alternative). Linked to the user interface, the business code defined the logic of the user interface. However, the business code does not express the logic of the application. This part is dedicated to the business code.

C. Business Code

The business code defines specific information of an application. It is composed by the general rules of the application (how calculate the taxes?), the distant services link (which server my business code should request), the data of the application (which database? which kind of *object*?). So the business code is not directly linked to the user interface.

III. Proposed meta-models

From the decomposition of the GUI application structure (see Section II), we decided to create four meta-models.

- The GUI model
- The layout model
- The business model
- The behavior model

A. GUI meta-model

In order to represent a GUI, we designed a meta-model presented Figure 1. This GUI model is the first part of the user interface component of a graphic application as defined Section II-A In the following, we present the different entities of the meta-model.

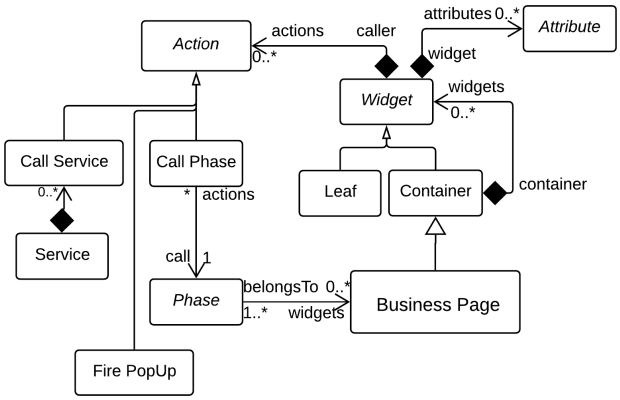

Fig. 1. GUI Application meta-model

The **Phase** represents the main container of a user interface page. It can be a *Window* for Desktop application, a web page or a tab in some specific case. For example, if a web site is composed by only one web page, and this web page has multiple tab. To represent the website as the content of the different tabs, the Phases will represent a tab. A Phase can contains multiple Business Page. It can also be called by any Widget with a *Call Phase Action*. When a Phase is called, the interface changes to display the phase. In desktop application, the current Window interface changes from the previous Phase to the called one or a new Window with the interface owned by the called Phase is opened. In web application, it can be a new web page, the changement of active tab, the transformation of the current web page.

The **Widgets** are the different interface components and layout components. There are two types of Widgets. The **Leaf** is a widget which does not contain another widget such as some text in the interface. The **Containers** can contain other widget. It could be for organization purposed like separating widgets which belongs to a category and widgets which belongs to another category.

The **Business Pages** blabla **Benoit** ►*Très BL specific... Il faudrait supprimer Page Métier*◄ .

The **Attribute** represents informations that belongs to a Widget and can change its visual aspect or its behavior. The common attributes are the height and the width to define precisely the dimension of a widget. There are also attributes to contains attributes such as the text contained by a widget. For example, a widget which represent a button can have an attribute *text* to explicit the text of the button. An attribute can change the behavior, it could be the case of an attribute *enable*. A button with the attribute *enable* positioned at *false* represent a button we can not click.

The **Actions** are own by the Widgets. They are actions that can be run in a Graphic Interface.

Benoit ►*Should disapear and be splitted between Business and behavior model*◄ **Call Service** represents a call to a distant service such Internet. **Fire PopUp** is the action that display a PopUp in the screen. The PopUp can not be considered as a Widget, it is not present in the GUI, it only appears and disappears.

Benoit ►*SHOUDL Disapear and go to the Behavior Model*◄ The **Service** is the reference to the distant feature the application can call from its GUI. In a web context, it can be the server side of the application.

B. Layout meta-model

Benoit ►*Ajouter un layout model*◄

The layout model is the second part of the user interface defined Section II-A. It contains the information concerning the visual disposition of the elements of the user interface. It does not include properties as color or text size of an element. The aim of the layout model is to explicit the layout applied by a *container widget*. It exists some common layout as **HorizontalLayout**, **VerticalLayout** or **GridLayout**. Each widget can only have one layout, but because a *container* can contain another *container*, it is possible to define complex layout composition.

C. Behavior meta-model

The behavior meta-model presented Figure 2 is the owner of the behavior code. There are two main elements, ths statement and the expression.

The **Statement** is the representation of the control structures of a programming language. There are the alternative, the loop, the notion of block and a link to an expression.

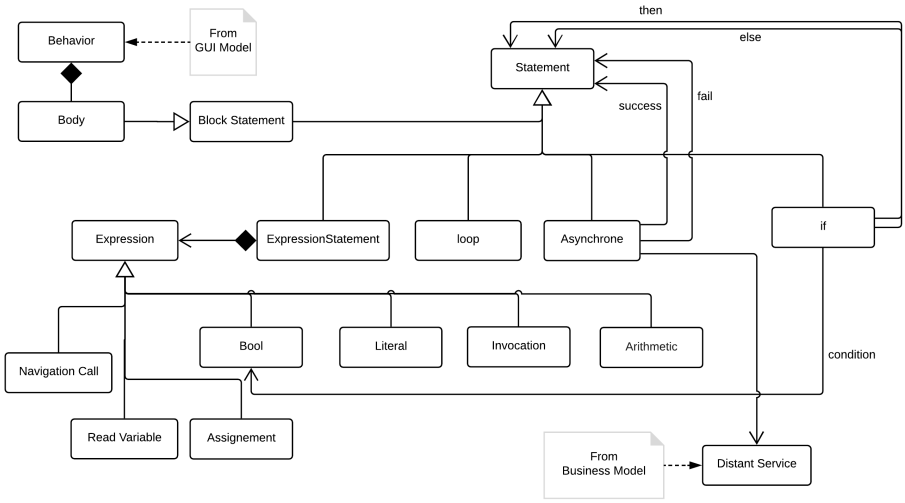

Fig. 2. Behavior meta-model

The **Expression** represents a piece of code that can be valued directly. So it can be an assignment with the value of the assignment, a literal (*i.e.* directly the value of the written element), a boolean expression, an arithmetic expression or an invocation. In the case of an invocation, it is the return value of the called method that is used as the value of the expression.

The **Behavior** elements is the link to the GUI model. It is the container of the logic of an event fired by an action.

Thanks to this model, we are able to represent the logic to execute once an user fire an event from an action on a widget of the GUI model.

D. Business meta-model

Benoit ►*Ajouter un business model*◄ The business model contains two main composites of the application. The data and the information about the distant services.

E. Model-View-Controller

We made a comparaison between our proposed models and the MVC pattern. The Table I present the matching between the models.

TABLE I
LINKED BETWEEN OURS MODEL AND THE MVC PATTERN

MVC	Ours Models
Model	Business
View	Layout
	GUI
Controller	Behavior

The MVC pattern is commonly used for developing user interfaces. It divides the application into three parts.

The Model part represents the data of the application. It also contains the logic of the data, *e.g.* business rules. This kind of data can be internally stored in the application, *i.e.* with local variable, or in a database. In the case of a web application, the logic inherent to the data can be contained by the database with stored procedure and triggers or by the back-end application part. If the logic is contained in the front-end, it is often considered as a security problem. In our proposed meta-models, the Model part of MVC is similar to our Business Model. Both contains data and business code.

The View part of the MVC pattern is the visible part of the application. It also includes the visual organization of the visibles component. It can be a complete web page or window as a unique forms or some buttons. The view used the Model part to get the information to display. In the case of a web application, the View part is own by the HTML file. The View part is similar to our GUI and layout meta-model. The GUI model contains at least the visible element which belongs to the View part and the structuring component. With the properties of the Widget and the layout meta-model, the meta-models represents how the component should be visually organized.

The Controller part defines the possible actions of the user. It is the link between the View part and the Model part through the user. In ours meta-models, we define those behavior in the Behavior meta-model. Like the Controller part, we make a link with the View thanks to the Behavior entity, and with the Model with the Distant Service entity. Our meta-model is also define to represent any kind of executable code.

IV. EXISTING SOLUTIONS

intro expliquand ce que l'on va faire (analyse des modèles)
Many authors have tried to represent a GUI application. They created meta-models that define the entities they need for their analysis. Those models helped us to define meta-models to represent any GUI applications. We analysed and grouped the meta-models of 14 relevant papers in the GUI application representation field.

A. GUI

explication gui The GUI meta-model represents the user interface of the application. It is the core meta-model for the representation of a user interface. It contains at least the visible widgets. The other models extends or add information to this one. It also defines the hierarchy structured of the widget which is often represented thanks to the composite pattern.

TABLE II
GUI ELEMENTS

SubWidgets	[6, 8, 13]
Actions	[2, 4, 5, 6, 7, 8, 11, 12, 15]
Properties	[2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

presenter en quelques lignes tout les modèles proposés Gotti and Mbarki [6] proposed a meta-model inspired by KDM model (see Section IV-E). The model has the main entities we also defined. There is the composite pattern to represent the DOM of a graphic interface. Their widget are called *Component* and as us they have properties. They also implemented an event property for the widget. But they inverse the name *event* and *action* compare to our solution.

Fleurey *et al.* [4] did not show the GUI meta-model directly but we extracted information from their navigation model (see IV-B). They have, at least, two elements in their GUI model that represent a *Window* and a *GraphicElement*. The Window seems to correspond to our phase entity. And because the *GraphicElement* and the *Window* are not linked, we can supposed that the *GraphicElement* is a widget. The *GraphicElement* has an *Event*. We do not have the totality of the GUI meta-model, but we can see that it is next to our.

The GUI meta-model of Sánchez Ramón *et al.* [13] is simple and very similar to our. It has the entities Widget and Window which correspond respectively to our Widget and Phase. Their Widget has their x and y position and their width and height as properties. It is similar to the link between Widget and Attribute in our meta-model. There is also the composite pattern to represent the DOM.

Morgado *et al.* [11] used a GUI meta-model but do not describe it. We only know that the GUI is represented as a tree which is similar to the DOM and can be represented thank to the composite pattern.

The GUI meta-model of Garcés *et al.* [5] differs a lot from our. There are the attributes, the events, the screen (which is like a Phase) but there is not widget. This absent is explained by the difference of source technology. The authors worked on a project that used oracle forms. This technology is used to create simple interface with only textfield or form. The textfield contains data from a database. The disposition of the elements is also really simple in the example provides by the authors because the textfield or form are displayed each below each other. We still can notice that they use an event entity to represent the action of the user with the user interface.

Memon *et al.* [9] represents a GUI user interface with only two entities in their GUI meta-model. A GUI window, which is similar to our Phase, is constituted by a set of widget. Those widgets can have properties and all the properties have a value associate. They do not have a representation of the DOM because it was not in the scope of their work. The authors defined a user interface as the set of widget and their properties, so, if a widget can two different values during the execution of the program, it belongs to two different user interfaces. This point is the major difference with the meta-models we proposed because, in our design if the value of a property change, we still be in the same user interface but a behavioral code has been executed.

Samir *et al.* [12] worked on the migration of Java-Swing application to Ajax Web application. They created a meta-model to represent the UI of the original application. This meta-model is stored in a XUL file and represents the widgets with their properties and the layout. Those widgets belongs to a Window, which is called Phase in our work, and can fire event when a GUI input is executed. In our work, we have the same system for the event but the Input is hidden under the concept of Action.

Shah and Tilevich [14] used a a tree architecture to represent the GUI. The root of the tree is a *Frame*. It corresponds to our Phase. The root contains components with their properties. The component entity is similar to our Widget entity.

Joorabchi and Mesbah [7] represent a user interface with a set of UI element. Those elements are our definition of a widget without the container pattern. For each UI element, the authors' tool are able to handle the detection of multiple attributes and of the event linked to the element.

Memon [8] used a GUI Model to represent the state of an application (see IV-C). As us, they have a widget with properties. The author made the difference between *widget* and *container*, it is similar to the use of the container pattern we use. With the notion of *container*, the author is able to represent a DOM.

Mesbah *et al.* [10] did not present directly the user interface meta-model they used. However, they explain that they use a DOM-tree representation to analyse different web page. They also use the notion of event that can be fired. They use different instances of their UI meta-model to represent the different web page of the application. Those instances can be compared to our Phase entities.

Amalfitano *et al.* [2] designed an Android Application GUI which is similar to ours meta-model. They have a notion of interface which is similar to our phase. An interface contains widgets. The widgets have properties and event right like in our model. Their event are parameterizable, this notion is in our behavior model. They do not use a container pattern to represent the DOM.

Finally, Silva *et al.* [15] used a GUI meta-model but did not present it in them paper. However, they explain that they use a GUI AST to detect the fragment that represents the user interface. We can linked this GUI AST to the container pattern we used in our GUI model.

We saw that the container pattern is often use to represent the hierarchy of widget. However, we detected three kind of

entities linked to the widget entity. The first one is the notion of sub-widget. A sub-widget is an extension of the container or the leaf entity, it could be the definition of a form, a panel, a textfield, a list, *etc*. The second one is the action. An action is linked to a widget. When a widget has an action, a user can performed this action on the widget and it will have an impact on the user interface. The last one is the notion of property. It represents a customisation of a widget.

Table II synthesizes which papers defines those entities. Almost all the GUI model use the properties and the actions but only three use the sub-widget. We also defined Actions and Property entity. We do not have integrated directly the sub-widgets, but it is possible to create an extension to our meta-model to represent themselves.

B. Navigation

explication nav To represent the link between two web pages. It is possible to use a navigation meta-model. This model makes the link between two top-level container of an application. It can connect two containers directly or through a sub-container or sub-element. It is also possible that the model links a element to an event, and this event to a top-level container.

Fleurey *et al.* [4] created a navigation meta-model to represent the transition between the different user interfaces or the forms completed by the user. The model is also linked to an entity called *Operation* but not well described. It still seems to be a way to call specific code to execute. The navigation meta-model proposed by the authors is similar to our behavior model.

Morgado *et al.* [11] designed two meta-models for the navigation. The first one is the windows model. It contains all the window available in the application. For each window, it describes the other window that can be reach from the original one. This describes the navigation between the window in an application. The second one is the navigation model. More than the link between the windows, this meta-model stores the user action to performed in order to execute the navigation. All this information are include in our behavior meta-model.

C. State flow

explication stateflow The state flow can be defined by a meta-model or properties between entities. It is used to represent the link between two states of an application. A state is defined by the visible widget and their properties. If the value of a property change, a new state is generated.

In order to represent the events, Memon [8] used the notion of state. A state represent the state of all the widget. Once an event is performed, if the state of a widget changes, a new state is created. The state of a widget depends to its properties and its visibility. This notion of changement of the user interface is similar to a navigation link. It is include in our behavior meta-model.

Joorabchi and Mesbah [7] used the states to detect the impact of an event on the user interface. They detect two different states of the application by applying an image-comparaison. During the comparaison they ignoring some difference as the text inside a textfield. The usage of the state can here be compared to a navigation meta-model, which are representing in our behavior meta-model.

Mesbah *et al.* [10] used the notion of state to represent the multiple visual an application can have. They execute the different actions possible on the widgets and analyse if the state of the application change. A changement in the application happen if at least the property of one widget change or if a widget is added or removed of the interface. The authors used the state as a representation of a navigation meta-model. The navigation meta-model is completely represented in our behavior meta-model.

Amalfitano *et al.* [2] worked on GUI ripping of Android application. In this case, they represented the user interfaces with a GUI model and the link between them thanks to a state flow graph. Those flow graph contains the information about the different visuals of the application and how to reach a visual from another. This link come from a possible performed action from a user of the analysed application. It is similar to the navigation meta-model. This navigation links are represented in our work in the behavior meta-model.

Silva *et al.* [15] used a directed graph to represent the navigation link between different state of an application. A state is the result of an action on a window when some condition are followed. It is composed by the internal state of the window which means the state of all the component of the window. Once a component property changes the state of the application changes. This behavior is represented in our behavior meta-model.

Aho *et al.* [1] worked on extraction GUI of an application. In order to represent the different interface that can be reach by using the piece of software, the authors used a finite-state machine. Each node correspond to a GUI interface and each edge a link between two windows. In this way, they can represent the navigation link inherent to the analysed application. The navigation linked proposed by the authors could be represented by a navigation meta-model. This last is completely included in our behavior meta-model.

D. Layout

explication layout To represent the position of the components of an interface it is possible to use a layout meta-model. The proposed meta-model defines multiple common layouts and describes the usage of a layout by a UI component. In some papers, the layout is integrated as properties of a component.

Gotti and Mbarki [6] used a layout meta-model to represent the position of the element in the user interface. The authors have chosen the meta-model designed by KDM. We described the meta-model Section IV-E.

Sánchez Ramón *et al.* [13] designed three layouts meta-models. The authors made this hight level of decomposition because their case study. The case study was the reverse engineering of a graphic application developed with Rapid

Application Development (RAD). The RAD often implies the non-usage of *common layout*. So, the authors had to find a way to extract the position of the ui element without using *HorizontalLayout*, *VerticalLayout*, *etc.* but finally representing the application with those layout. **Benoit** ►*To recheck*◄ The first model they designed is the Region model. The Region model defines squares in the user interface that match the widgets of the interface and precises the sub-region. This model is a way to recreate the composite pattern of our GUI model. The second model is the Tile model. This model positioned the different regions with others. If a region is inside another, it precises the position of the sub-region (top, middle, bottom, left, right, centre). The tile model groups also the spatially close region and precises the relation between them (horizontally or vertically align). Finally the CUI model generates layout for the UI elements from the Tile model. It exists three kind of layout, the FlowLayout, the StackLayout, the GridLayout and the BorderLayout. The last idea of associating a layout to a ui element is similar to the notion of layout we linked to a widget.

Samir *et al.* [12] did not describe the layout model they designed. But they precise they use the XUL format as they did for the GUI description Section IV-A.

E. KDM

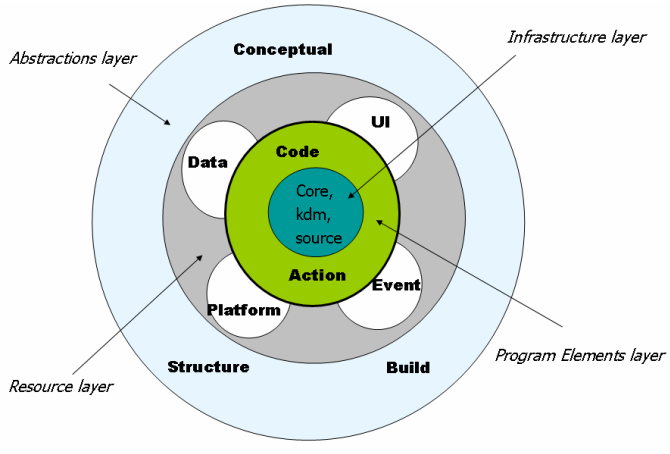

Fig. 3. KDM Architecture

The OMG (https://www.omg.org/spec/KDM/) defined KDM[1] standard to support the evolution of software. The standard is a meta-model to represent a piece of software in a high level of abstraction.

The Figure 3 presents an overview of the KDM architecture. It is divided in twelve packages organized in four layers. The UI package is composed by a set of meta-model to represent the components and behavior of the user interface. The package action defines meta-models to represent the behavior of an application. It can be used to describe the logic of the application, *e.g.* conditions, loop, call. It is similar to our behavior meta-model. The Data package represents the data from different database. It also defines a meta-model for the action of the database,*e.g.* Insert, Update, Delete. Those actions can be linked to a trigger and so automatically executed when an event occurs.

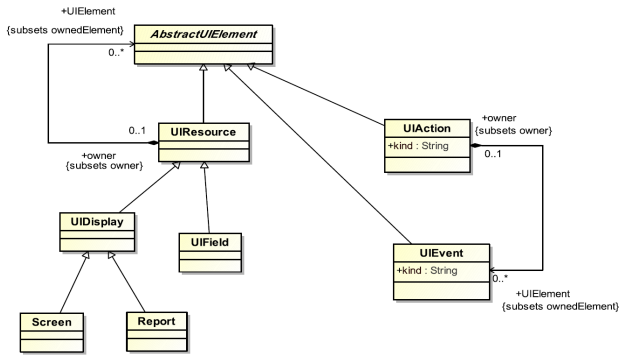

Fig. 4. KDM UIResources Class Diagram

The Figure 4 is the UIResources Class Diagram. It defines many entities for the abstraction of a user interface.

UIResource can be defined as UIDisplay, UIField or UIEvent. A UIField is used to represent forms, text fields, panel, *etc.* It is similar to the entity Widget in our UI model. We can also detect the pattern composite with UIResource and AbstractUIElement. This pattern makes the representation of the user interface architecture possible.

UIDisplay can be a Screen or a Report. The Screen represents a window for desktop software or a web page. The Report represents an element that will be printed. This difference between Report and Screen is not represented in our meta-models.

Every AbstractUIElement can have an UIAction. A UIAction is used to represent the sequence of display in a user interface. It includes the logic to go from a UI to another. It is not a navigation model nor a state flow model and it is similar to our behavior model.

A UIAction can performed multiple UIEvent. The UIEvent represents the events that belongs to a user interface. It can be the callback notion or the idea of navigation (*i.e.* going from one web page to another).

The Figure 5 is the UIRelations Class Diagram. It represents the disposition of the UIElements with the others and the flow from UIDisplay to another.

The flow between two Display is not represented in ours meta-models. It defines the behavior of the user interface as the sequential flow from one instance of the display to another.

The UILayout defines the layout of an AbstractUIElement for a specific Display. This notion is similar to our Layout meta-model.

F. IFML

Brambilla *et al.* [3] defined the Interaction Flow Modeling Language (IFML) to represent a front-end application. The

[1]KDM: Knowledge Discovery Metamodel

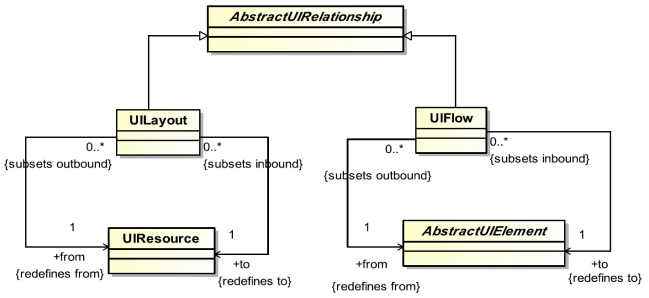

Fig. 5. KDM UIRelations Class Diagram

work is defined with the OMG (http://www.ifml.org/). The aim of IFML is to provide a system to describe the view part of an application, with the components and the containers, the state of the components, the logic of the application and the binding between data and user interface.

With IFML, a piece of software can be modeled with one or multiple top-containers. The containers represents a main window for a desktop application or one web page for a Web application. Then, each containers has sub-containers or can contain a components.

A component is the abstract level of a visible widget. This element can have input or output parameters. An input parameters from a data ressource means the widget displays the value of the data.

The container and the component can be associate with events. An element linked to an event support users' interaction, *e.g.* click, drag-and-drop, *etc.* Once the action is performed, the effect is represented by an interaction flow connection which connect the event and the elements affected by the event.

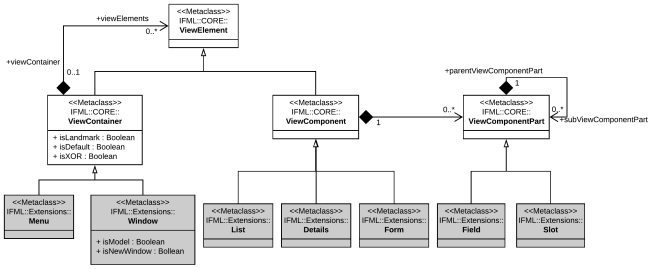

Fig. 6. IFML View Elements

Figure 6 presents the meta-model *View Elements* proposed by IFML. This meta-model has the same aim as the GUI model we designed, *i.e.* representing the visible part of the user interface. We notice that they are slightly different. Both use a container pattern and so have the notion of container and component, respectively container and leaf in our meta-model. The IFML meta-model introduced the notion of ComponentPart. This entity is necessary to represent all the possible component because the authors decided to set elements like a List or a Form as a Component. In our case, a List is represented as a Container, so we do not need a ComponentPart because we can use container pattern to represent the sub-element of a component.

V. Reverse Engineering Strategies

A. Static

B. Dynamic

C. Mix

VI. Discussion

Critères d'évaluation

- Certains auteurs n'ont pas présenté le schema mais l'on seulement décrit

VII. Conclusion

In this paper, we defined four meta-models to represent a GUI applications.

The **GUI model** represents the different *pages* of the applications and their contents. It includes the properties of the widgets and the link from the widgets and their events.

The **layout model** expresses the positioning relation between the widgets of the GUI models.

The **behavior model** defines the behavior to execute when an event is fired. The events are fired by using an action on a widget of the GUI model or automatically by the application.

The **business model** includes information about the data manipulated by the GUI application. Those data can be used by the behavior model and transmit to the GUI model.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

References

[1] Pekka Aho, Matias Suarez, Teemu Kanstrén, and Atif M Memon. Industrial adoption of automatically extracted gui models for testing. In *EESSMOD MoDELS*, pages 49–54, 2013.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[3] Marco Brambilla, Piero Fraternali, *et al.* The interaction flow modeling language (ifml). 2014.

[4] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. Model-driven engineering for software migration in a large industrial context. In *International Conference on Model Driven Engineering Languages and Systems*, pages 482–497. Springer, 2007.

[5] Kelly Garcés, Rubby Casallas, Camilo Álvarez, Edgar Sandoval, Alejandro Salamanca, Fredy Viera, Fabián Melo, and Juan Manuel Soto. White-box modernization of legacy applications: The oracle forms case study. *Computer Standards & Interfaces*, 2017.

[6] Zineb Gotti and Samir Mbarki. Java swing modernization approach: Complete abstract representation based on static and dynamic analysis. In *ICSOFT-EA*, pages 210–219. SCITEPRESS - Science and Technology Publications, 2016.

[7] Mona Erfani Joorabchi and Ali Mesbah. Reverse engineering ios mobile applications. In *Reverse engineering (wcre), 2012 19th working conference on*, pages 177–186. IEEE, 2012.

[8] Atif M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, September 2007.

[9] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, November 2003.

[10] Ali Mesbah, Arie Van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3, 2012.

[11] I Coimbra Morgado, Ana Paiva, and J Pascoal Faria. Reverse engineering of graphical user interfaces. In *The Sixth International Conference on Software Engineering Advances, Barcelona*, pages 293–298, 2011.

[12] Hani Samir, Eleni Stroulia, and Amr Kamel. Swing2script: Migration of java-swing applications to ajax web applications. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 179–188. IEEE, 2007.

[13] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 147–186. ACM, 2014.

[14] Eeshan Shah and Eli Tilevich. Reverse-engineering user interfaces to facilitateporting to and across mobile devices and platforms. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pages 255–260. ACM, 2011.

[15] João Carlos Silva, Carlos Silva, Rui D Gonçalo, João Saraiva, and José Creissac Campos. The guisurfer tool: towards a language independent approach to reverse engineering gui code. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 181–186. ACM, 2010.