computing@computingonline.net
www.computingonline.net

Print ISSN 1727-6209
On-line ISSN 2312-5381
International Journal of Computing

A MODEL-DRIVEN APPROACH FOR MULTI-PLATFORM EXECUTION OF INTERACTIVE UIS DESIGNED WITH IFML

Sara Gotti, Samir Mbarki, Zineb Gotti, Naziha Laaz

MISC Laboratory, Faculty of science, Ibn Tofail University, Kenitra, Morocco,
gotti.sara1990@gmail.com, mbarkisamir@hotmail.com, zinebgotti01@gmail.com, laaznaziha@gmail.com

Paper history:
Received 28 March 2019
Received in revised form 18 June 2019
Accepted 10 September 2019
Available online 30 September 2019

Keywords:
IFML;
Model Execution;
MDA;
Bytecode;
Virtual Machine;
Model Interpretation;
Model Compilation;
Human Computer Interaction;
GUIs Plasticity;
Computing Everywhere.

Abstract: Quite recently, considerable attention has been paid to the design, implementation and evaluation of graphical user interfaces due to the apparition of the new strategic trend of computing everywhere. Accordingly, it is necessary to adopt an abstract representation of systems front-end in order to ensure this trend. IFML (Interaction Flow Modeling Language) is a user interfaces description language used to describe the content and interaction behavior of applications front-end. It has been conceived with executability aspect that is obtained via model transformations and full code generators into functional codes. however, these code generators are often accompanied by a loss of information. The main goal of this paper is to present a new virtual machine for directly executing GUIs models designed with IFML language in combination with UML domain model; that captures the content dependency. These input models will be then run on different platforms and devices. We adopted a new model driven approach that includes the hybrid approach of interpretive compiler; through a set of transformation rules, for the implementation of the desired virtual machine.

1. INTRODUCTION

Before GUIs systems, users interacted with their systems using the command line interface. Graphical user interfaces have appeared afterwards with their WIMP (windows, icons, menus and pointers) toolset to make it easy for a human to navigate and interact rapidly with interactive systems. Recently, GUIs tools have undergone remarkable evolution and drastic changes in response to the platform requirements and the diversity of devices which is beneficial for users. However, this may encounter some practical problems since it is required to develop multiple GUIs to be run in each device for the same system. This operation is really tedious and time consuming. Actually, it is considerably laborious to build system front-end than to deal with the domain logic. Therefore, the problem that could arise is that it might be very hard for enterprises to cope with this new trend of computing everywhere regarding the time to market. Accordingly, there is a need to plastify the UIs, that is to say to adapt UIs to the context of use while preserving usability [1]. So, users could work everywhere through different devices.

Actually, conceptual models allow to have a complete vision of the business processes of a system. They are conceived with a much longer life than the technologies used to implement the application since they overcome technical constraints. So, for sustaining GUIs plasticity, it is recommended to use conceptual models for describing GUIs at a high level of abstraction without concerning technical issues. Basically, an engineering conceptual model must satisfy these five key characteristics: abstraction, understandability, accuracy, predictiveness and inexpensive [2]. The concept of User Interfaces Description Language (UIDL) could be used in this scope; it represents a formal high-level language for the definition of GUIs [3]. Among the set of UIDLs already exist, we cite the Interaction Flow Modeling Language (IFML); the OMG vision of UIDL.

IFML is a user interfaces description language designed to express the content, user interaction and

control behavior of a system front-end. It is a platform independent description of GUIs that focuses on the representation of the general components, interactions and front-end behavior in which there is no definition of graphics and styles. It has been designed with executability in mind and it is open to extensibility.

It appears clearly that Model Driven Engineering (MDE) and human machine interfaces are two disciplines dedicated to being married. The need for union is even more obvious when we consider the plasticity of the GUIs for which platform switching is dynamic. In fact, IFML was conceived to be executable, that is to say it could be easily transformed to source codes via code generators and model transformations.

In this present work, we propose a new implementation of the Model Driven Architecture (MDA) [4]; MDA is the OMG's particular vision of MDE, for directly executing models designed with IFML. IFML and MDA work here together for the engineering of advanced plastic user interfaces. We have chosen a direct execution of IFML models through an MDA driven process instead of following the code generation option to avoid its drawbacks. The process of execution is based on the building of a new virtual machine under the acronym IFVM (IFML Virtual Machine) for executing GUIs. The process admits the general view elements of an application front-end designed with IFML, plus a second representation describing a domain model such as UML diagrams [5]. UML diagrams have been added to make the binding for extracting information to be shown in the interface, and to ensure any type of navigation, even that which carries data.

The remainder of this paper is structured as follows. In section 2, the related work will be discussed. Section 3 is devoted to introduce the IFML user interfaces description language chosen, and to discuss its executability and the general key elements of content and navigation with IFML. The proposed process of execution for the desired IFVM virtual machine is detailed in section 4. Section 5 shows the experimental results on a running example. The conclusion is reported in section 6.

2. RELATED WORK

After the apparition of IFML language, it has appeared that it came out with several benefits to the development of system's front-end whether it is in web, desktop or mobile. In the literature, there have been some works that have been proposed in the field of adaptive UIs; according to the context of use, which are classified into four categories [6]: 1) Translation Interface, 2) Reverse-engineering and migration Interfaces 3) approaches based on the markup languages and 4) model-based approach.

Besides, other researchers have proposed additional works for a generation of GUIs that is based on several model driven approaches starting with IFML models. Naziha et al. [7] have discussed a number of existing model driven works and IFML modeling tools for the development of GUIs within an in-depth comparative study. Another IFML based solution has been proposed in [8] taking a different way for software modernization according to an architecture-driven modernization (ADM) [9] based approach. The point of convergence between these all previous IFML based solutions, is that they proceed through the code generation step in order to reach the source code of the corresponding GUIs design to be run later.

However, many publications are available in the literature that propose solutions for directly executing models without passing by code generation. Authors in [10] present a comparative study on a subset of these solutions in the field of back-end development based on UML models.

It has been shown that very few or no publications can be found that addresses the issue of direct execution of front-end representation, unlike the back-end one, without passing by the code generation step. Even so, in our earlier work, we proposed different prototypes of IFML virtual machine for directly executing the GUIs representation designed with IFML. A key limitation of the last one in reference [11], is that it doesn't take into account to identify the source of data; from external artefact, to be displayed in the interface, as well as to consider the navigation that carries data to be transmitted from a view element to another one. On the basis of the mentioned lacks, we propose an implementation of the desired virtual machine for directly executing IFML models accompanied with a domain model such as UML diagrams to cover the content dependency within a view element.

3. BACKGROUND

In this section, we will enlarge some of the main concepts that play an important role in understanding the rest of the paper.

3.1 GUIS EVOLUTION

A graphical user interface (GUI) or a graphical environment is a human-machine dialogue interface, in which the objects to be manipulated are drawn in the form of icons on the screen. So, the user can control, communicate and interact with interactive systems via devices using these objects; icons or widgets, rather than command-line based interfaces

or text-based user interfaces (TUI). Author in [12] offers a brief history of the important research developments in Human-Computer Interaction (HCI) technology.

In fact, GUIs have undergone several changes and evolution during the last five-decade span, this is illustrated in Fig. 1.

Figure 1 – GUIs evolution

First, users interacted by typing orders in form of command lines to call operations to be executed. There have been, after, important incremental refinements to the mode of interactions in steps, starting by introducing menus and textual interfaces in natural languages instead of commands. Query dialogues were used after as an interaction manner by means of questions and answers. Forms were added later as an alternative way for easily deal with inputs. While the WIMP toolset; commonly identified as GUI, came out, afterwards, to replace the earlier computing with a graphical easy to interact system.

3.2 GUIS VS COMPUTING EVERYWHERE

Modern computing has influenced human life on a manner allowing user to work and access information anywhere and anytime using its laptop, tablet, smartphone, and even wearable devices, and that is shown in Fig. 2. Actually, computing everywhere cannot exist without mobility.

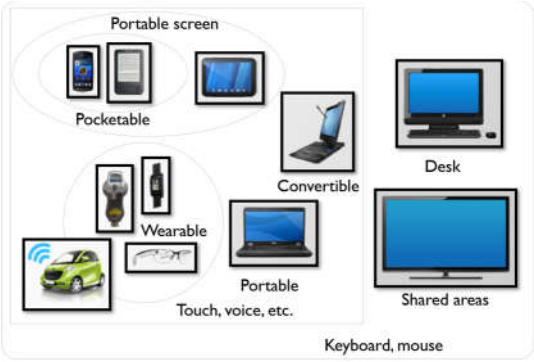

Figure 2 – GUIs vs Computing everywhere

In addition, this modernity has also touched the GUIs representation that should be able to adapt to the context of use in response to the mobility. It helped user by permitting more flexibility, choice and freedom in daily tasks making them smoother which is beneficial for workers. Therefore, computer devices are going to weave themselves into the fabric frequently until it is impossible to distinguish them [13].

Accordingly, user interfaces should be more plastic and adaptive whenever the context changes, in order to make them possible to be run on various computing devices. However, enabling that plasticity is found to be more difficult and tedious in practice, since it requires the development of multiple user interfaces in a separate way for each desired platform or device. The conceptual models have reached a remarkable success to cover all the business processes needs of a system abstractly without considering technical constraints. In spite of that, the use of conceptual models for building system front-end to deal with computing everywhere discipline has not been well-articulated.

In fact, various model-based solutions have been emerged to treat this issue. They provide platform independent conceptual models for the description of UIs within a high-level UI Description Language (UIDL). A UIDL is considered as a common way to describe characteristics of GUIs independently of any target platform. Moreover, it could be then easy to generate the appropriate code of the designed GUIs by means of model-based technics for developing GUIs.

Actually, UIDLs aim at capturing abstractly all the necessary requirements for UIs, what makes UI's definition stable across variety of platforms and devices by applying automatic generation of code. Besides, they help improving UIs reusability to support evolution, extensibility and adaptability of a user interface. Examples of UIDLs are discussed in [14].

Thus, for conceiving and implementing User Interface Management System (UIMS), it is required to choose a UIDL model in order to cleanly separate process or business logic from GUI code.

IFML [15] is a Domain Specific Language (DSL) standard that has been adopted by OMG in 2013. It has been designed to capture content, user interactions and front-end behavior of software front-end, independently from the implementation technology and deployment platform, as well as the binding to the domain model expressing the business logic. IFML is also considered as a UIDL since it permits an abstract description of all GUIs concerns.

3.3 IFML LANGUAGE

Actually, IFML is a PIM standard that brings several benefits to the front-end development process. It organizes the structure design of the interface in terms of a set of gathering elements called ViewContainers that assemble other elements of type ViewContainer or ViewComponent for content display and data entry. IFML offers the possibility to change the interface state from one container to another by means of interaction flows or navigation relations associated with an event that occurs. There are three types of events [16]:

• ViewElementEvent; caused after a user interaction,

• ActionEvent; caused after the execution of an action,

• SystemEvent; caused by the system itself.

The IFML specification is accompanied by four technical artifacts to help understanding the language, we cite: The IFML metamodel, the IFML UML profile, the IFML visual syntax, and the IFML XMI.

In this work, we focus on the IFML definition via the metamodel artifact which describes the semantics and relations between the modeling constructs.

The IFML metamodel is made up of three packages: The Core package, the Extension package and the DataTypes package.

• The Core package: gathers the abstract and general concepts that build up the language infrastructure, such as InteractionFlow-Element, InteractionFlow and Parameter.

• The Extension package: extends the concepts defined by Core package by concrete concepts to manage more complex behaviors.

• The DataTypes package: contains the basic data types defined in the UML metamodel, and specializes a number of UML metaclasses as the basis for IFML metaclasses, and presumes that the IFML DomainModel is represented in UML.

Within a model of type IFML, the general design of interfaces is made up of one top level element called IFMLModel, in which, we incorporate two other metaclasses of type InteractionFlowModel; that offers the general view through ViewElements, action and events, and DomainModel; for the definition of concepts.

As mentioned before, there are three key ingredients supported by an IFML model, see Fig. 3. These ingredients permit to a system's modeler to set the necessary view elements, their relationships and their dependencies with actions and concepts.

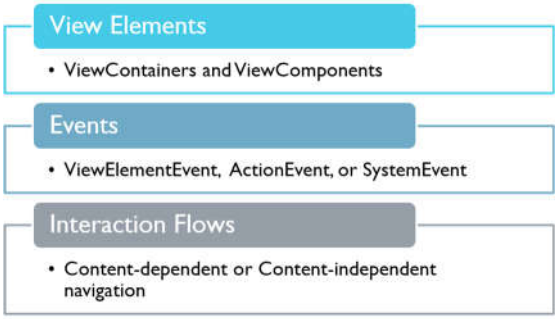

Figure 3 – IFML key ingredients

When an event is occurred, it could cause an interaction flow navigation from a view element to another with/without passing parameters. Therefore, two types of navigations: a content-dependent navigation which carries objects from the source of the navigation to be passed to the target element, and a content-independent navigation which is a simple and independent form of navigation.

3.3.1 CONTENT INDEPENDENT NAVIGATION

It presents a basic form of navigation from a source ViewContainer to another target one; associated with an InteractionFlow, after an event is occurred. It is content independent which means that the user interaction brings a change to the state of the interface by displaying the content of the target ViewContainer without caring about the content of the source one. That is, it is not required to pass parameters from the source of navigation to the target in order to display the content of the target ViewContainer.

Fig. 4 illustrates a very simple IFML model exemplifying this concept. It shows two ViewContainers; Mails and Contact. The first incorporates a List view element displaying the set of mails, and the second has a List showing the set of contacts. They are associated with a NavigationFlow caused by an event occurrence after a user interaction of type click. It causes a content-independent navigation targeting the display of Contact List.

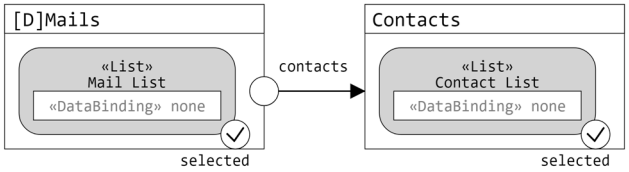

Figure 4 – Simple navigation between Mails view container and Contacts view container

3.3.2 CONTENT DEPENDENT NAVIGATION

It corresponds to an additional behavior offered by IFML representing the second form of navigation which is content dependent. It is similar to content-independent navigation. However, navigation, here, is done by means of ViewComponents and not ViewContainers. It results in changing the content of ViewComponent to display other ones, but this time is dependent on the content of the source ViewComponent. This dependency is ensured by the ParameterBinding concept added to the NavigationFlow. We talk about input/ouput dependency.

As illustrated in Fig. 5, the "Album Details" ViewComponent displays the details of the specific Album selected by the user from the "Album List" ViewComponent. The NavigationFlow is associated with a ParameterBindingGroup that contains the value of the output parameter of the source ViewComponent.

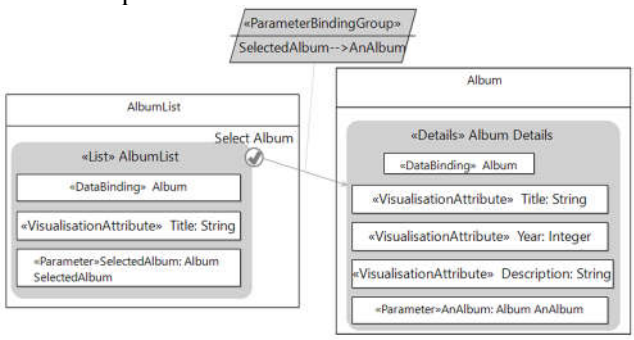

Figure 5 – Navigation after selection between AlbumList and Album Details with parameter passing

3.3.3 CONTENT DEPENDENCY

It should be noted that IFML allows the display of content, within a ViewComponent, that could be derived from a different source. So, the ViewComponent should be accompanied by information about the source of the content to be displayed. Therefore, the DataBinding concept is used to express the source of the content from objects of domain model such as UML class diagram, Entity-Relationship models, ontologies, or other elements.

Fig. 6 shows an example of using the DataBinding concept. The "AlbumList" ViewComponent draws its content from the "Album" Class of the UML domain model (left side). VisualisationAttribute was added to locate the data to be shown in the interface, such as an object attribute.

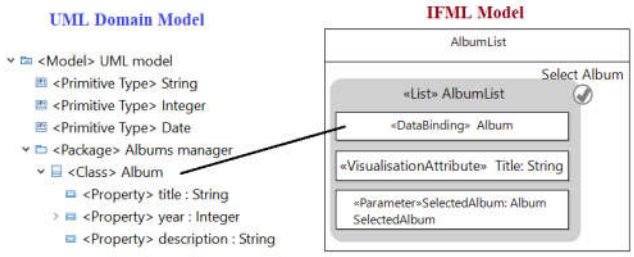

Figure 6 – DataBinding associating the AlbumList view component with UML class diagram

3.3.4 IFML EXECUTABILITY

IFML has been conceived with executability aspect, that is to say, it permits to get easily and automatically the appropriate executable code via model transformations and code generators. It is then recommended to use executable models to ensure the automation.

We mean by executable models, syntactically correct models in term of executability, i.e., they cover the representation of static; structural definition of elements, and dynamic part; behavioral definition [17].

User interaction, within a view, produces events that could affect the state of the views and then execute actions that could signal another event and that are what the execution semantics of IFML.

3.4 MODELS EXECUTION

In the past, developers used to compile their assembly code into machine language. Then, they started to work alike but with high level languages. After the appearance of the model driven development trend, first, they started by working with the automatic code generation from models. Next, developers defined their own approaches for directly implementing models.

In fact, as can be seen from Fig. 7, there are two main approaches were defined for executing models: Code generation and Model direct implementation that has two different forms of execution which are model interpretation and model compilation.

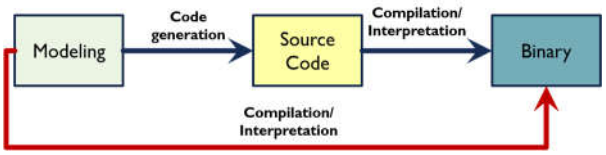

Figure 7 – Two types of models' execution

The code generation approach has firstly allowed the developer to focus just on modeling without worrying about code, since it is automatically generated. However, the code generation is often accompanied by a loss of information what makes

developers forced to modify the generated source code. So, it may create a gap between the model and the source code, which is in contradiction with the benefits of MDE. Accordingly, a direct model implementation could be the solution to prevent any lack of information during the process of the development.

4. IFVM: MDA-BASED PROCESS

As discussed before, GUI's developers should take into account several constraints; heterogeneity of end users, heterogeneity of computing platforms and languages, heterogeneity of working environment and eventually of the context of use. Therefore, to ensure GUIs production, despite these constraints, model-driven UIs development could be systematically used for rapid production of suitable GUIs via alternative designs to permit a good change management. The MDA approach allows rapid development and validation since it is model based, and it enables a set of transformations from abstract representation of concepts to concrete software.

In this paper, we propose a model driven approach that builds on the union of MDA with GUIs abstract description, especially that designed with IFML. It brings a new solution of virtual machine for multiple execution of GUIs in response to the context. So, to reach this objective, we have determined a set of parameterized transformations from abstract UIs models to concrete interfaces.

In the next subsection, we describe the MDA-based process named IFVM, and aim to establish the necessary guidelines to allow the automatic execution of GUIs description designed with IFML.

4.1 PROCESS OVERVIEW

The proposed approach is shown in Fig.8. The IFVM virtual machine was proposed as a model driven process for automatic and direct execution of systems front-end designed at a high level of abstraction. The process allows the developers to abstractly design the interfaces and transform them into concrete software, as well as conducting rapid development and validation according to the context of use.

The process highlights two OMG standards used as input to generate this implementation, that are:

• IFML model: it models the general structure of front-end content, user interactions, the structure of navigations between the view elements, as well as the binding to resources for extracting information.

• UML domain model: it represents the content model that provides the data to display in the interface.

The process has merged the IFML model; describing the GUIs definition, with a domain model of type UML, since the front-end design requires to exploit the knowledge about objects and associations within an application.

We implemented this MDA process in a simple way by means of Eclipse-based tools. We started by designing the front-end definition in the form of IFML model that conforms to its metamodel [15]. This later offers the possibility to add a sub model that corresponds to the appropriate domain model. The process runs automatic transformations from models to binary, in order to obtain a front-end execution able to be run on different devices. Therefore, developers design the input models, and then the IFVM virtual machine takes care of generation of the equivalent binary according to the appropriate platform. This approach had allowed several benefits for developers in development time and cost, as well as unnecessary need for development skills, and eventually the consistency of the output in response to the context of use.

As can be seen from Fig.8, the process of the virtual machine incorporates two units: the compilation and the interpretation unit. Actually, there are various ways to implement VMs. The present process is based on the merging of two concepts of implementation which are the compilation and the interpretation to conceive the desired virtual machine. Thus, this hybrid approach takes benefits from their advantages for fast execution [18]. It bridges the gap between the abstract and the concrete representation within a real machine by using an intermediate VM code of instruction set that is the bytecode. This bytecode simple format; result of the compilation, is used as intermediate code for reducing dependence on hardware and facilitating its quick interpretation on several architectures.

Accordingly, we elaborated the IFVM bytecode metamodel; the IFVM instruction set resulting from the compilation first and interpreted by the VM later. So, we can now launch a series of model-based transformations from GUIs design to the binary code generation. Additional details about this intermediate representation is given in the following section.

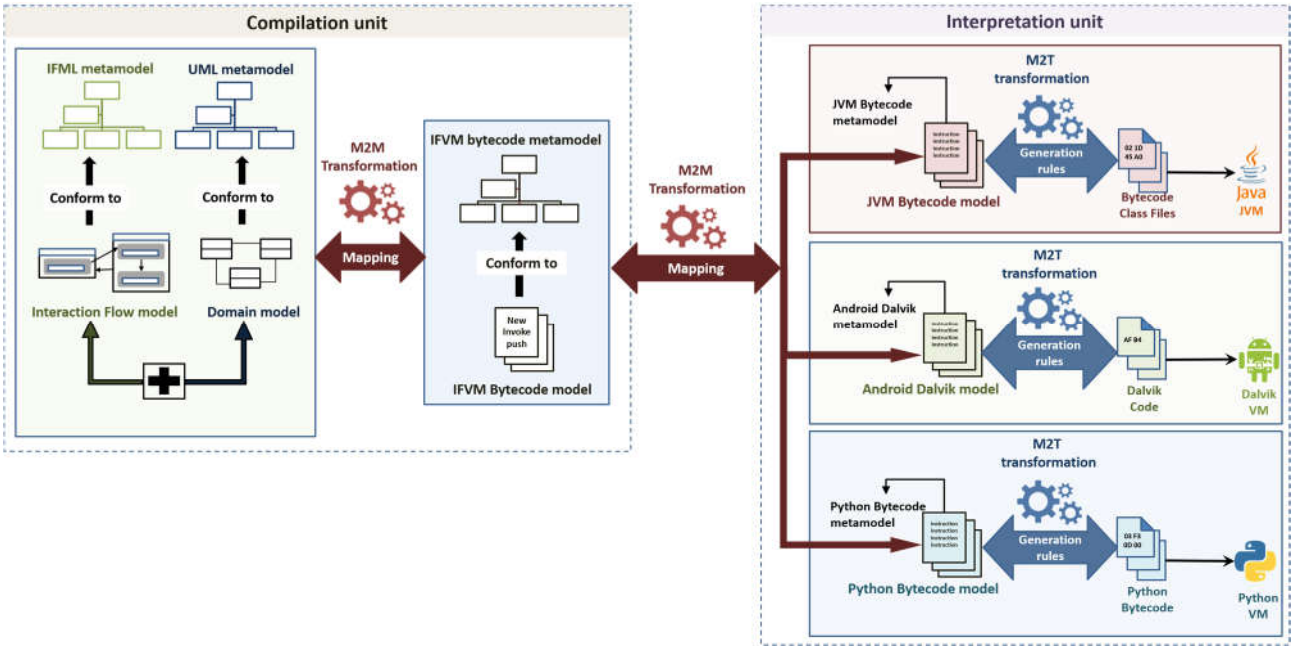

Figure 8 – IFVM Process

4.2 IFVM BYTECODE: IFML VIRTUAL MACHINE INSTRUCTION SET

Actually, there is a need to adapt an intermediate representation during the process of model execution. The bytecode has been chosen as being intermediate representation in order to gain optimization and portability. It is a transitional code between low-level machine instructions and high-level source code, that is not directly executable. This facilitates its interpretation on several architectures using different interpreters, since it is hardware and operating system independent. A bytecode interpreter is a virtual machine that executes the code just like a microprocessor in a portable way. That is to say that the bytecode could be transmitted from one machine to another for which an interpreter exists, and interpreted by different types of hardware architectures. In fact, many interpreted languages are compiled first into bytecode and then executed by an interpreter. Among these languages, we cite: Java, PHP and Python.

Accordingly, we propose a new model-based definition of bytecode for the desired IFVM virtual machine. It is intended to gather a set of instructions. So, we elaborated the IFVM Bytecode metamodel with a set of meta-classes representing bytecode instructions whose syntax was inspired by Java bytecode instruction set.

In fact, there are two types of virtual machine implementation: stack based and register based VM. It depends on the way operands and results are stored. In stack based VM, values are stored onto the stack, however in register based one, values are stored onto registers. To discover the differences between these two ways of implementation, the reader is referred to Table 1 that summarizes each one's properties.

Table 1. Stack based vs Register based virtual machine

Stack based VM	Register based VM
Values onto the stack (push and pop)	Values onto registers
Run on any CPU design with a stack	Each CPU design has its own number of registers
Simple, powerful and portable	Faster (no push and pop)
Hardware and operating system independent	Each process must have its own VM instance

As can be seen from Table 1, each implementation has several assets. However, a stack-based interpreter would be the good implementation for our proposal, since it is hardware and operating system independent. It could be now easy to run the same bytecode on multiple architectures using different interpreters.

By going back to the process schema illustrated in Fig.8, IFML model and its corresponding domain model are compiled first to IFVM Bytecode. So, each element from the input models are mapped to its equivalent instruction in IFVM Bytecode. We have defined a set of instructions of IFVM Bytecode; derived from Java bytecode instruction set, under the form of meta-classes within its appropriate metamodel. Fig. 9 shows an extract of the proposed IFVM Bytecode metamodel.

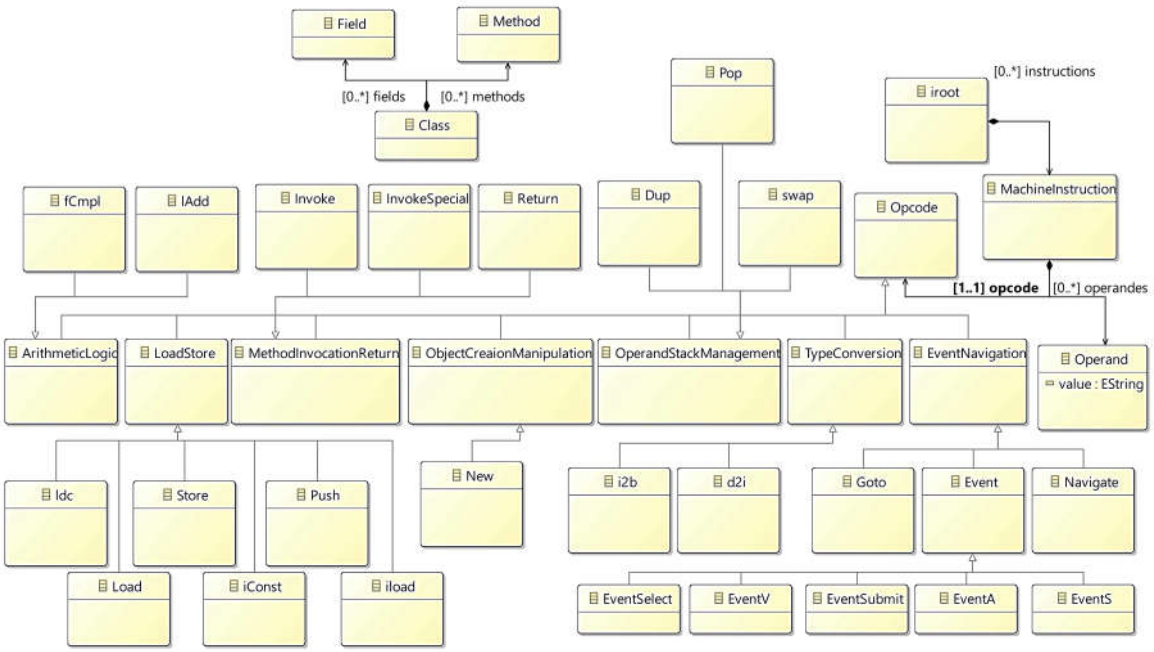

Figure 9 – IFVM Bytecode metamodel

As can be seen from Fig. 9, several opcodes could be found in Java bytecode instruction set. Moreover, additional opcodes have been incorporated for expressing events and navigations after a user interaction, as well as instruction of type invoke that allows calling methods such us create(), addelement(), and eventually setter methods. Additional information about the top used instructions is depicted in Table 2.

Foreach property that accompanies IFML model elements, we have found a solution for representing them in the IFVM bytecode as pushed and popped values onto the stack.

Furthermore, we have assigned foreach type of events, declared in IFML models, its corresponding instruction in the IFVM Bytecode as described in Table 2.

An event could affect a navigation. For that, a navigate instruction is used for expressing the target of the navigation and eventually the parameter binding; when the target view element displays content that relies on the source view element content.

Therefore, we can now cover the general view of a user interface within the IFVM Bytecode format, as well as the behavior through events and navigations. It is now easy to map each element form IFML models to its corresponding instruction in IFVM Bytecode.

The following section presents, in details, the implementation of the proposed process, and specifies the sequence of tasks involved in the whole process.

4.3 IFVM IMPLEMENTATION

This section describes the implementation of the MDA-based process named IFVM. It provides the necessary guidelines to allow an automatic execution of models designed with IFML. It follows a sequence of steps encompassed in two major units: the compilation and the interpretation unit.

4.3.1 COMPILATION UNIT

As previously stated, the first task of the model execution process is to create an IFML PIM model to cover the content and behavior of a system front-end, which must be then compiled to an IFVM Bytecode PIM model. In addition, it is necessary to add a domain model as input to ensure the binding with information to be displayed within the interface.

The information needed to create the IFML model, according to its IFML metamodel, is provided from the specification [15]. the creation of IFML is all around the building of a Core model that includes the description of interaction and domain model.

The interaction flow model is built through a set of view elements. First, we create an element of type *Window*, that represents the *ViewContainer* to support the other *ViewComponents*. We continue by adding, within the window, other *ViewComponents* of type *Form*, *List*, or *Details* with their appropriate fields. And eventually, we could enrich the model by adding other concepts such as the *DataBinding* concept; that refers to the source of content to be displayed, as well as adding the dependencies, in term of *NavigationFlow*, that connects the *ViewComponents* together after an event is triggered.

Table 2. IFVM Instruction set

Opcode	Stack [Before] → [After]	Description
Push	→ property	push a property onto the stack
Pop	property →	discard the top property on the stack
New	→ object	create new graphical object
Invoke	[arg1,arg2,...]→result	invoke a method and put result on the stack
EventS	→EventObject, triggeringExpression	Put on the top of the stack the expression triggering an event of type SystemEvent and an event object
EventV	→ EventObject	Create an event object of type ViewElementEvent. There is no need of ParameterBinding since it corresponds to content independent navigation
EventA	→ EventObject	Put on the top of the stack an object of type ActionEvent triggered. It will be followed by a navigate instruction for expressing the navigation to another view element
EventSelect	→ EventObject	Put on the top of the stack an object of type OnSelectEvent triggered. It will be followed by a navigate instruction for expressing the navigation to another view element
EventSubmit	→ EventObject	Put on the top of the stack an object of type SubmitEvent. It will be followed by a navigate instruction for expressing the navigation to another view element
Navigate	→NavigationObject, [SourceParam, TargetParam]	Put on the top of the stack a set of ParameterBinding to be passed in the navigation flow after an event is triggered, and a navigation object

As for the domain model, it is associated with a UML class diagram that contains all the necessary classes and fields representing the content to be possibly appeared within the interfaces.

The two PIM obtained models are then compiled, i.e. transformed into another PIM model, which is the IFVM Bytecode model, in order to raise the abstraction level independently of platforms and architectures. Compilation is established by using a specific language to define automatic model to model transformation, that is QVT [19].

We have developed a set of rules allowing this transformation. As an illustration, we clarify, in the following algorithm, an extract of the mapping applied to generate the equivalent IFVM Bytecode.

```
Input ifml: IFML
Output ibytecode: IFVMBytecode
begin
map  ifmlmodelToiroot(ifml.InteractionFlow-
Model);
end
      /**********mapping1**********/
mapping
ifmlmodelToiroot(imodel:interaction-
FlowModel):iroot
begin
 for  all  w.isTypeOf(Window)  ϵ  imodel.
interactionFlowModelElements
   => map windowToNew(w)
 end for
end
      /**********mapping2**********/
mapping windowToNew(w:window): new
begin
      foreach p ϵ w.properties
      => add push instruction
      => insert p as operand to push
instruction
      end for
      => add invoke instruction
      => insert create () as operand to
invoke instruction
      //Store w into variable i
      => add store_i instruction
      /*********step2*********/
      for all vc ϵ w.viewComponents
      // form or list or details
        => map elementToNew(vc)
      //Load window w from i
        => add load_i instruction
      //Load ViewComponent vc from k
        => add load_k instruction
      // Binding w with vc
        => add invoke instruction
        => insert addElement() as operand
to invoke instruction
      end for
end
      /**********mapping3**********/
mapping elementToNew(vc:ViewComponent): new
begin
 if vc.isTypeOf(List)
  foreach p ϵ vc.properties
    => add push instruction
    =>  insert  p  as  operand  to  push
instruction
  end for
      /*********step3*********/
  for all e ϵ vc.viewElementEvents
    => add eventV instruction
    => add navigate instruction
    => insert targetNavigation as operand
to navigate instruction
  end for
      /*********step4*********/
  if vc.onSelectEvent is active
    => add eventSelect instruction
    => add navigate instruction
```

```
    => insert targetNavigation operand to
navigate instruction
    => add push instruction
    =>     insert     sourceParameterBinding
operand to push instruction
    => add push instruction
    =>     insert     targetParameterBinding
operand to push instruction
  end if
  => add invoke instruction
  =>  insert  create()  operand  to  invoke
intruction
//Store vc into variable k
  => add store_k instruction
      ....
   end
```

The first mapping aims at corresponding the two root elements of IFML and IFVM Bytecode metamodels, which are *interationFlowModel* and *iroot*. Such mapping represents the main operation through which we could appeal to the rest of mapping making the correspondence. It looks over all the contained view elements of type *Window* to be mapped by calling a second mapping.

The extract of mapping 2 transforms each *window* into a *New* machine instruction and captures all the window properties to be mapped into instructions of type *Push*. Then, it follows by adding *Invoke* instruction for calling the creation method of the window, and eventually the *Store* instruction to store the created object. Step2 of the current mapping, is dedicated to map the inside *ViewComponents* by calling the third mapping, and make the binding after using *Load* and *Invoke* instruction as detailed in the algorithm.

As for mapping 3, this operation is dedicated to make correspondence between view elements of type ViewComponent; *Form*, *List*, and *Details*, to their equivalent instructions in IFVM Bytecode in the same way we did with *Windows*, but this time, each *ViewComponent* will be mapped in a separate IFVM Bytecode model. Properties such as *VisualizationAttribut* and *DataBinding* are mapped in the form of values to be pushed with *Push* instruction.

Step3 manages the mapping if an event of type *ViewElementEvent* is triggered. It is then transformed into instructions of type *EventV* and *Navigate*, with specification of the target of the navigation by adding the *TargetNavigation* operand to *Navigate* instruction.

Step4 is devoted to deal with another type of event which is *OnSelectEvent*. It is then mapped to a set of instructions starting with *EventSelect*, and *Navigate* with a *TargetNavigation* operand, followed by *Push* instructions for pushing *ParameterBinding* values to be passed from the source to the target *ViewComponent*.

Regarding the domain model, it is then compiled by easily storing the classes, their attributes ,and eventually the constants.

4.3.2 INTERPRETATION UNIT

The principle objective of this second unit is to interpret the obtained IFVM Bytecode from the previous unit and generate the equivalent concrete representation which is the binary, to be run in the adequate platform. As can be seen from Fig. 8, the interpretation of the IFVM Bytecode is implemented by following these two stages:

• Model to model transformation:

IFVM Bytecode → Java Bytecode, Dalvik Bytecode, and Python Bytecode,

• Model to text transformation, to get the equivalent bytecode format of each bytecode model.

IFVM Bytecode instruction set has been designed with abstraction to gain portability. Indeed, to ensure this portability in implementation, we decided to work with implementations that already exist. We talk about Java, Android Dalvik, and Python implementations.

The first stage has a single task, which carries out the model to model transformation, in which the obtained IFVM Bytecode model is evolved to three other types of bytecode models. The bytecode models are represented according to their three metamodels, which are the Java Bytecode metamodel, the Dalvik Bytecode metamodel and the Python Bytecode metamodel. The three metamodels of the bytecode forms have been elaborated in accordance with their specifications in [20, 21, 22].

The model to model transformation can be formally established by QVT language [19]. Table 3 outlines the mapping between elements from IFVM Bytecode into their equivalent elements in Java Bytecode syntax.

Accordingly, we proceed in the same way to build the mapping with the other two forms of bytecode.

Once we get the equivalent bytecode format; JVM, Dalvik and Python, we could now pass to the second stage of the interpretation unit which is launching a model to text transformation to get the real bytecode text, to be run on real or existing VMs. This transformation is formalized by means of the open source Acceleo [23] language; an Eclipse implementation of the OMG MOF2Text Transformation Language that maps model elements into text instructions. However, it could be difficult to fill in the template with binary code, while the computed text from elements provided by bytecode models is not written in binary. Therefore, the filling, within the template, must be performed by means of bytecode editing libraries. ASM library

[24] is one of the existing libraries for Java Bytecode manipulation and analysis. So, we simply need to fill in the Acceleo template by the ASM-based program that generates dynamically and directly the Java Bytecode class files. We act alike for the other bytecode types, that is to say that we use libraries for building the binary form of bytecodes within the Acceleo template.

Once we generate the bytecodes numeric codes, they will be passed to the VMs to be executed and produce finally the binary code that will subsequently be run in platforms.

Table 3. Mapping IFVM Bytecode to Java Bytecode

IFVM Bytecode	Java Bytecode
push	bipush
pop	pop
new	new
invoke	invokeSpecial (e.g. invoking constructor method) invokeVirtual
eventS	invokeVirtual (the action listener method)
eventA	
eventV	
eventSelect	
eventSubmit	
navigate	new (instantiate the target interface)
store_n	astore_n

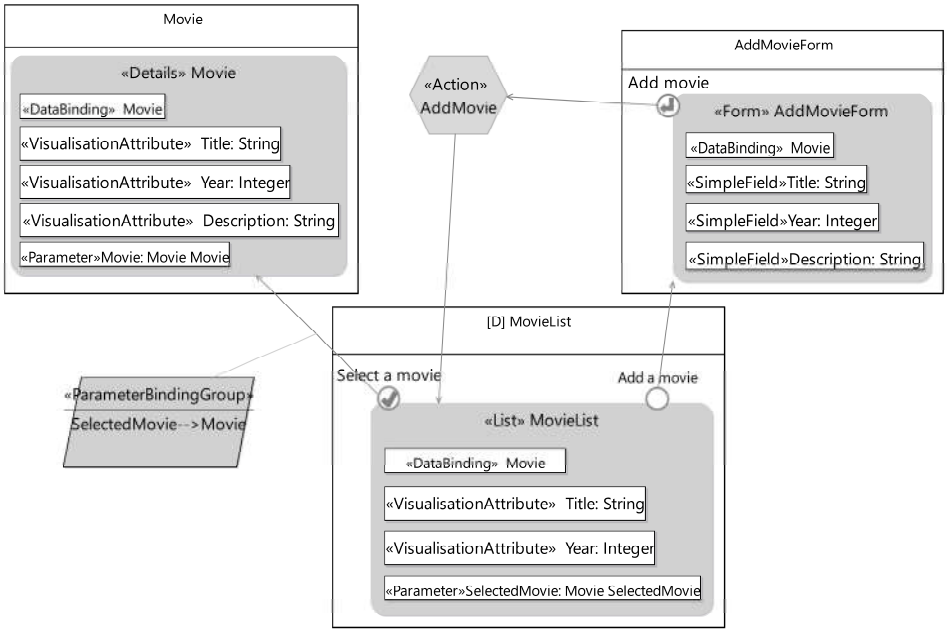

Figure 10 – Movie manager system designed with IFML

5. ILLUSTRATING EXAMPLE

This section presents a case study to demonstrate the feasibility of IFVM, the proposed MDA-based process. This project addresses the execution of GUIs abstract description designed with IFML. The case study relates to a movie manager system. Within this system, a user could add movie, list movies and eventually display the details of a selected movie from the list. Fig. 10 shows the general views and possible navigations of the chosen system, designed with eclipse IFML editor.

The view contains three ViewContainers of type Window: MovieList, Movie and AddMovieForm windows. The Movie Window incorporates a ViewComponent of type Details to display the descriptions of a selected movie. The MovieList Window, in its turn, contains a ViewComponent of type List that permits showing the set of existing movies. As for the third Window, it includes a ViewComponent of type Form that allows a user to add a movie to the existing list. The following paragraphs present the details of the main stages defined during the process.

The first unit of implementation consists in transforming an XML file instance of IFML metamodel (see Fig. 11). It is obtained from the graphical design with eclipse IFML editor [25].

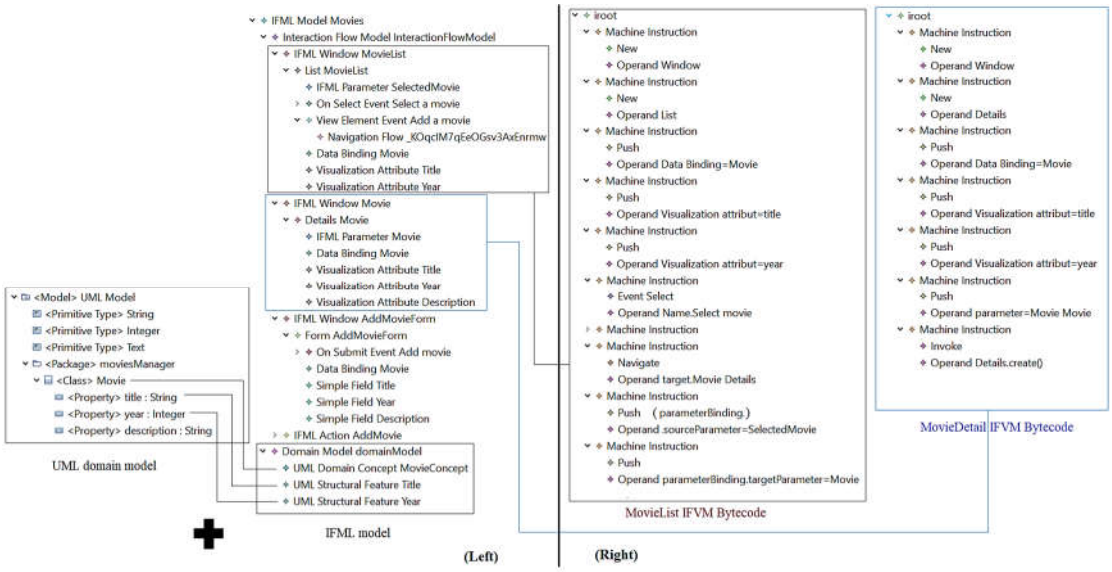

Figure 11 – Input of compilation unit (left). Output of compilation unit (right)

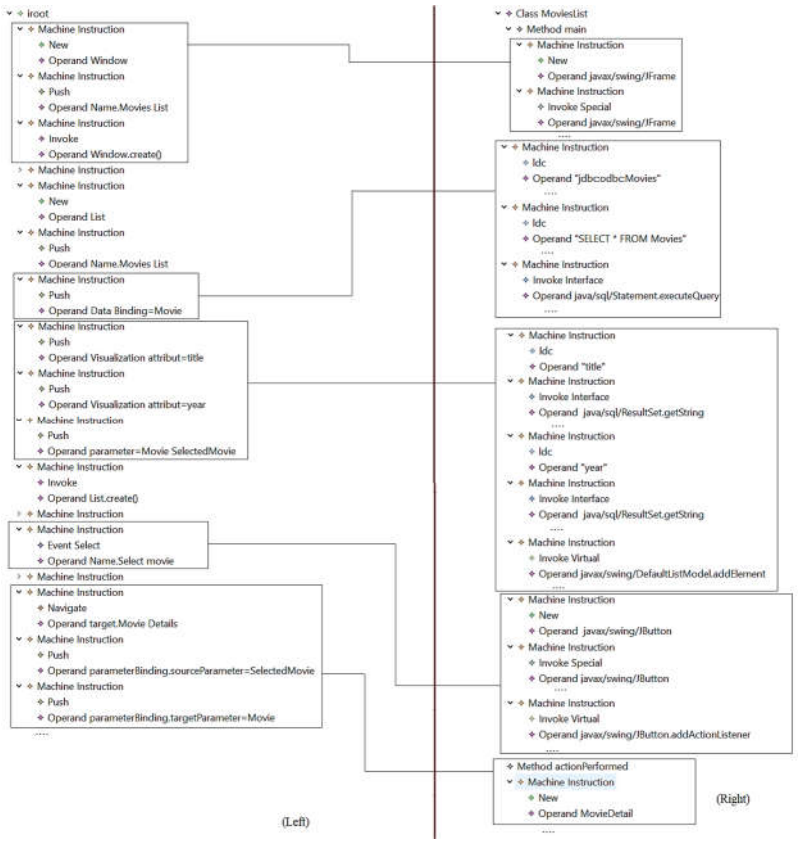

Figure 12 – Input of interpretation unit (left). Output of interpretation unit (right)

The IFML model and its corresponding domain model are used as input (left of Fig. 11) on which the transformation rules are applied.

The result of transformation is a set of XML files that correspond to the IFVM Bytecode instructions foreach Window. Right of Fig. 11 exposes the generated IFVM Bytecode models of MovieDetail and MovieList Windows.

IFVM Bytecode XML files, of each window, will be passed for a second mapping, within the interpretation unit, to produce models of existing bytecodes forms, that are Java Bytecode model, Dalvik model and Python Bytecode model.

Right of Fig. 12 shows an extract of Java Bytecode model of a window of type List, for displaying the list of movies, corresponding to its IFVM Bytecode model shown in left of Fig. 12.

Once we get the bytecode models, we established a model to text transformation, in which, the text to be generated is the program of the library permitting

the bytecode editing.

Fig. 13 shows the execution result after running the obtained Java Bytecode class files, that have been generated using the ASM library.

The first window is the MovieList Window, through which, a click on see details button makes the display of the MovieDetail Window, we talk about content dependent navigation. And eventually, a click on Add movie button permits the display of the AddMovie Form without considering the content, we talk about content independent navigation.

6. CONCLUSION

This research work explored the feasibility to use a model-driven approach to automatically execute user interfaces and interactions designed with IFML with content provided by UML domain model.

The execution process is based on the usage of a set of metamodels to represent the models involved in each unit of the process. For example, the IFML, UML, and IFVM Bytecode metamodels have been used during the compilation unit. Moreover, the Java Bytecode, Dalvik Bytecode, and Python Bytecode metamodels have been utilized throughout the interpretation unit.

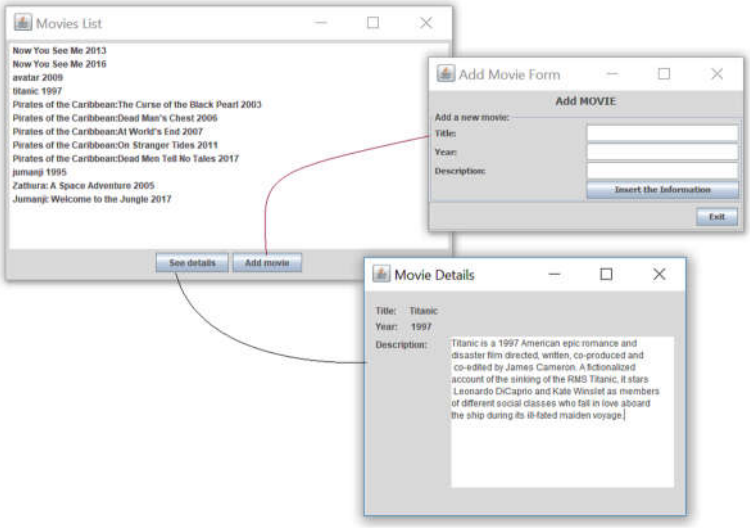

Figure 13 – Execution result

Our contribution offers an easy way of building and maintaining the software front-end from abstract representation. It relies on the use of transformations and model tools without passing by an intermediate code generation phase. It excludes all the errors that could arise during code generation, and helps increasing portability of GUIs execution.

The next step in this research initiative is to develop a framework that combines front-end representation with the back-end that captures the business operations to make a fully automatic executions according to model-driven approaches.

7. REFERENCES

[1] J.-S. Sottet, G. Calvary, and J.-M. Favre, "Models at run-time for sustaining user interface plasticity," *Proceedings of the 2006 International Conference on Models Run Time Workshop Conjunction Model*, 2006, pp. 1-4.

[2] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19-25, Sep. 2003.

[3] O. Shaer, R.J.K. Jacob, M. Green, K. Luyten, "User interface description languages for next generation user interfaces," *Proceedings of the International Conference on Human Factors in Computing Systems CHI'08*, New York, NY, USA, 2008, pp. 3949–3952.

[4] J. Miller and J. Mukerji, *MDA Guide Version 1.0.1*, 2003.

[5] OMG, "About the Unified Modeling Language Specification Version 2.5.1," 2017. [Online]. Available at: https://www.omg.org/spec/UML/.

[6] W. Bouchelligua, A. Mahfoudhi, L. Benammar, S. Rebai, and M. Abed, "An MDE approach for user interface adaptation to the context of use," *Proceedings of the International Conference on Human-Centred Software Engineering*, 2010, pp. 62–78.

[7] N. Laaz, K. Wakil, S. Mbarki, and D. N. A. Jawawi, "Comparative analysis of interaction flow modeling language tools," *J. Comput. Sci.*, vol. 14, no. 9, pp. 1267–1278, Oct. 2018.

[8] Z. Gotti and S. Mbarki, "Java swing modernization approach – Complete abstract representation based on static and dynamic analysis," *Proceedings of the International Conference on ICSOFT-EA*, 2016.

[9] OMG, *ADM Platform Task Force*, Object Management Group, 2003. [Online]. Available at: https://www.omg.org/adm/.
[10] S. Gotti and S. Mbarki, "UML executable: A comparative study of UML compilers and interpreters," *Proceedings of the 2016 International Conference on Information Technology for Organizations Development (IT4OD)*, 2016, pp. 1–5.
[11] S. Gotti and S. Mbarki, "IFVM bridge: A model driven IFML execution," *Int. J. Online Biomed. Eng. IJOE*, vol. 15, no. 4, pp. 111–126, Feb. 2019.
[12] B. A. Myers, "A brief history of human-computer interaction technology," *Interactions*, vol. 5, no. 2, pp. 44–54, Mar. 1998.
[13] A. Schuster, Ed., *Intelligent Computing Everywhere*, London: Springer-Verlag, 2007.
[14] J. Guerrero-Garcia, J. M. Gonzalez-Calleros, J. Vanderdonckt, and J. Munoz-Arteaga, "A theoretical survey of user interface description languages: Preliminary results," *Proceedings of the 2009 Latin American Web Congress*, 2009, pp. 36–43.
[15] IFML: The Interaction Flow Modeling Language, The OMG standard for front-end design, [Online]. Available at: https://www.ifml.org/
[16] M. Brambilla, *The IFML book – OMG's Interaction Flow Modeling Language explained*, 1st ed., Morgan Kaufmann, 2014.
[17] E. Cariou, C. Ballagny, A. Feugas, and F. Barbier, "Contracts for Model Execution Verification," *Proceedings of the European Conference on Modelling Foundations and Applications ECMFA'2011*, 2011, pp. 3–18.
[18] A. Aggarwal, D. S. K. Singh, and S. Jain, "A Hybrid Approach of Compiler and Interpreter," *International Journal of Scientific & Engineering Research*, vol. 5, no. 6, p. 4, 2014.
[19] OMG, "About the MOF Query/View/Transformation Specification Version 1.3," 2016. [Online]. Available at: https://www.omg.org/spec/QVT/About-QVT.
[20] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java® Virtual Machine Specification*, Oracle Help Center, 2013, 604 p.
[21] D. Ehringer, "The Dalvik Virtual Machine," 2010. [Online]. Available at: http://www.davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf.
[22] A. Rigo and S. Pedroni, "PyPy's approach to virtual machine construction," *Proceedings of the Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'06*, 2006, pp. 944-953.
[23] Acceleo. [Online]. Available at: https://www.eclipse.org/acceleo/documentation/.
[24] E. Bruneton, "ASM 4.0 A Java bytecode engineering library," 154 p.
[25] Eclipse, "Open source IFML editor." [Online]. Available at: http://ifml.github.io/.

***Sara Gotti**, PhD Student. She got her Master Degree in software quality in 2013. She is a researcher on studying the execution of conceptual models at MISC laboratory in Faculty of science, Ibn Tofail University, Morocco.*

Her main research interests are related to the establishment of a model compiler/interpreter,

***Samir Mbarki**, he received his B.S. degree in applied mathematics from Mo-hammed V University, Morocco, 1992, and Doctorate of High Graduate Studies degrees in Computer Sciences from Mohammed V University, Morocco, 1997. In 1995, he joined Ibn Tofail University, Morocco where he is currently a Professor in Department of Mathematics and Computer Science. His research interests include software engineering, model driven architecture and natural language processing.*

***Zineb Gotti**, PhD Student. Received her master degree in software quality in 2013. Her activities of research focusing on interactive systems moderni-zation and evolution at MISC laboratory in Faculty of science, Ibn Tofail University, Morocco.*

***Naziha Laaz**, PhD Student, holding a Master Degree in software quality. Her main research interests are related to the model driven engineering using ontologies for graphical user interfaces generation at MISC laboratory in Faculty of science, Ibn Tofail University, Morocco.*