IFVM Bridge: A Model Driven IFML Execution

https://doi.org/10.3991/ijoe.v15i04.9707

Sara Gotti(✉), Samir Mbarki
Ibn Tofail University, Kenitra, Morocco
gotti.sara1990@gmail.com

Abstract—Graphical user interfaces (GUIs) present a powerful part of software systems that allows a simplified assimilation and manipulation by users throw visual objects such as text, image and button. however, with the increasing complexity of GUIs and the diversity of their interaction mode required by users to access information anywhere and anytime, the need for designing efficient and more adaptive user interfaces has become a necessity. Therefore, UIs researchers have attempted to address these issues by designing user interfaces at a high level of abstraction to separate GUI's specification from its implementation. Besides, the OMG (Object Management Group) adopted the Interaction Flow Modeling Language (IFML) [1] as a standard in March 2013 for this purpose. In this paper, we present a new model driven development approach to efficiently execute the abstract representation of software's front-end with focus on navigation between the views. We introduce a IFML virtual machine IFVM which executes user interfaces by passing from IFML models to be translated into an intermediate bytecode representation proposed as the instruction set of IFVM virtual machine.

Keywords—Interaction Flow Modeling Language IFML, model execution, MDA, bytecode, virtual machine, model interpretation, model compilation, Platform Independent Model PIM, user interfaces, front end

1 Introduction

Over the last few decades, a new methodology for building systems has appeared. It has emerged with a number of approaches driven by models as the important artifact for guiding the software production. Furthermore, these model driven engineering (MDE) proposals aim at increasing productivity by starting to build software systems with abstract models and then automatically generate the software from models. Among these proposals we cite the model driven architecture (MDA) [2]; it is the object management group's (OMG) vision of MDE. Therefore, working with abstract models should be helpful to control software complexity in order to follow the imposed changes and the industry requirements related to cost and time to market. After all, high level representation by abstract models has made maintenance of software system easier and less expensive than the source code after the building of such system.

Besides, another trend has emerged which is computing everywhere. It relates to the availability of products and services on a variety of technologies and platforms. So, data can be accessed and managed anywhere through any device using multiple screens.

However, it could be difficult by enterprises to ensure the success of this trend in practice since this mobility requires the development of thousands of user interfaces to access data. In order to prevent this problem from evolving, it is needed to abstractly represent user interfaces by platform independent models according to MDA architecture to facilitate execution on different computing platforms and technologies. Accordingly, OMG group has adopted a solution in March 2013 as a standard which is the Interaction Flow Modeling Language (IFML) [1].

IFML permits the expression of content, user interaction and navigation options of the system's front-end without considering the implementation specific issues, as well as the connection with persistence and business logic to complement other modeling dimensions and to help the development process by reusing abstract models. However, in order to increase automation in the development process, models need to be executable to directly reach application binary using either a model interpretation or model compilation [3]. Moreover, IFML has been designed with executable semantic that permits a direct mapping into the application binary.

In this paper, we present a novel approach for executing IFML models. It supports elements used to specify the general organization of the interface and even the basic form of navigation that we call content independent navigation between view containers expressed with events. We define IFVM, a IFML virtual machine which directly executes IFML models through a computation based on bytecode interpreter. It starts by compiling IFML models in order to obtain the bytecode instruction set of the IFVM and finally generates the equivalent binary via model-based technics.

The paper is structured as follows. The next section discusses related works. Section 3 introduces the user description language IFML and its executability. In section 4, we describe the IFVM bytecode chosen as the instruction set for the IFML virtual machine, and then we explore the general process for executing IFML within IFVM. Section 5 provides a detailed example to validate our process. Finally, we conclude the paper with a brief summary of our key contributions and suggestions for future work.

2 Related Works

After the apparition of MDA concept, many approaches have been applied to directly execute system conceptual models with executable semantic, without generating the appropriate code. The authors here [3] present a comparative study on a number of works proposed for executing UML models using the compilation or the interpretation concept. Actually, there are many proposed solutions allowing the back-end models execution without code generation.

Regarding the importance of software front-end, a group of MDE based proposals have been defined for UIs generation through code generation, like for [4] that proposes a model driven approach for generating rich internet application UIs from IFML models chosen for abstractly design the software front-end. Another work here [5] presents an approach that has the same target as [4] but the authors here start by combining the ontology model that presents the logical description of UI components with IFML model in order to obtain a good understanding of system user interfaces. Besides, another proposal defines a comprehensive tool suite called WebRatio Mobile Platform for model-driven development of mobile applications starting by extending IFML for the mobile domain [6].

In addition to that, and specifically to generate web application UIs, the author here [7] propose a new framework for extending IFML metamodel in the hope of generating modern UIs.

IFML could be used not only in the model driven engineering, but also in the software modernization, and here we cite [8], one of the researches that proposed an architecture-driven modernization-based approach to obtain knowledge of the structure and behavior of source code by generating three independent platform combined models (KDM, IFML and TaskModel). The models capture various aspects about tasks, presentation and dialog structures and behaviors of the design knowledge, needed for the construction of the future user interface (UI).

It has been shown, during this section that many works have been proposed for executing systems back-end, and generating UIs through code generation starting with IFML as the abstract representation of systems front-end. But no solution has been presented for directly executing user interfaces and interactions designed with abstract models. Except in [9], we introduced a prototype of IFML virtual machine for executing IFML models in Java virtual machine with the use of java bytecode as the instruction set of the proposed VM. However, in this present work, we provide a new definition of IFML virtual machine by elaboration our own bytecode instruction set as detailed below.

3 Background: Conceptual Modeling of HCIs

The HCI (Human-Computer Interaction) is the discipline devoted to the design, implementation and evaluation of interactive computer systems for users. It takes care of all tools allowing a human to control, communicate and interact with an interactive system.

After the apparition of the new trend of ubiquitous computing, user interfaces need to be capable of adapting to their context of use while preserving usability [10] thing that we call plasticity. In this situation, there is a need for advanced effort to cope with platforms changes rather than wasting time to reinvent the wheel every time by developing various separate GUIs for each software application in different operating environments. Various systems have been proposed to guarantee interoperability between GUIs and the diverse platforms in which we cite the model-based approach. It allows plasticity by using platform independent HCI conceptual models from which we can derive platform specific artifacts through a model driven engineering MDE.

Conceptual modeling of HCIs presents a key allowing UIs definition in a high level of abstraction according to the four modeling criteria [11] which are: comprehensibility, relevance, predictability and low cost, thing that overcomes constraints related to the platform. The question that arose is how to achieve it. Hence, it’s required to use a formal, high-level language called User Interface Description Language (UIDL) [12].

A UIDL is a descriptive language used to abstractly describe GUIs without considering implementation details. Therefore, a number of UIDLs have emerged that will serve for describing UIs independently of any implementation. Here we are going to focus on IFML language; since it is referred to as a UIDL; as the input artifact for the desired virtual machine for the present work.

OMG launched IFML (Interaction Flow Modeling Language) a platform independent description language for visually expressing the content, user interaction and control behavior of the application front end. It covers the representation of different user interfaces aspects. We cite the view structure, the view content, the events and event transitions and finally the parameter binding with only one diagram type called interaction flow diagram [13].

The structure of the application view part with IFML is built by a number of view containers which may contain other sub containers or view components that enable content display and data entry. The event represents the user interactivity with view containers and view components. It triggers actions that may change the state of the user interface. The dependency between the view elements is defined through what we call parameter bindings associated with navigation flows or data flows that describe data transfer.

Moreover, there are so many aspects that could help understanding the IFML language [13], we cite: the IFML language definition; there are four technical artifacts that compose the IFML language specification [1]; and the IFML executability; that allows the mapping between IFML constructs to any executable UI platform.

3.1 IFML Metamodel

The IFML language definition through Meta model is considered as the best method chosen for describing the semantics of and relations between the modeling constructs of the language. It includes abstraction, modularization, reuse and extensibility.

The IFML meta-model is divided into three packages: The Core package, the Extension package, and the Data Types package.

The Core package contains the abstract and general concepts for building the infrastructure of the language such as Interaction Flow Elements, Interaction Flows, and Parameters. These defined concepts are extended by concrete concepts in the extension package to treat more complex behaviors. The IFML Meta model incorporates the basic data types defined in the UML Meta model into the third Data Types package. It specializes some UML Meta classes as the origin for IFML meta-classes, and presumes that the IFML domain model is represented in UML [1].

IFML Model represents the top level Meta class that contains other model elements; we cite the Domain Model, the Interaction Flow Model, as well as the View Point. Interaction Flow Model offers the user's view of the whole application, by reference to the "Interaction Flow Model Elements" sets, which together define a wholly functional portion of the system. The Interaction Flow Model Element is the main concept of IFML that can describe all the front-end requirements, such as View Elements, View Component Parts, Actions and Events, which participate in Interaction Flow connections.

3.2 IFML executability

In the hopes of improving productivity by increasing the levels of abstraction, automation, and analysis, a model driven development (MDD) approach utilizes three key elements in the development process: models, model transformation, and Meta models. Therefore, in order to increase the automation in the development, it is recommended to use executable models to generate an executable code (C, C++, Ada, Java, Forth, even VHDL) automatically from these models or directly execute them in order to generate the equivalent binary. The model execution could be considered as the next execution paradigm that substitutes models for code.

On the one hand, models could represent the execution of the application like code programs. On the other hand, these models must be syntactically correct in term of executability by applying verification or validation techniques. For executing models, we need to define it complete Meta model that contains two kinds of meta-elements [14], we cite:

Static part: structural definition of the model elements defining the static view of a model. For the IFML diagram, it defines the concepts of View Containers, View Components, and events.

Dynamic part: structural definition of the elements specifying the behavior (execution state) of a model. For the IFML diagram, it defines the concept of changing the state of the view in response to triggering events.

IFML is a platform-independent language but has been designed with executability in mind. This is obtained through model transformations and code generators to ensure that the conceptual IFML constructs can be mapped easily into executable applications for various platforms and devices.

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

In fact, there are two forms of triggering events produced by a user: event in a View-Container that affects another View Container by a Navigation Flow and an event that affects an element inside the same View Container. A state of interface collects visible View containers, active View Components, and events. A View Container is visible when it respects its visibility turn according to a composition model that contains the entire View Containers of the system. A View Component is active if its View Container is visible and its input parameters values are available.

The IFML execution semantics describe any IFML diagram as a machine that takes as input the interaction of the user and updates the state of the interface for the user to continue the interaction.

4 IFVM: The IFML Virtual Machine

The IFML Virtual Machine (IFVM) is a new concept of virtual machine designed for executing user interfaces trough a model driven development process. It enables model-oriented development using a UI's representation based on IFML models for the specification and execution of system's front end. We cover the general organization of the interface and high-level navigation caused by an event occurrence. We support a basic form of navigation which we call content-independent navigation. “The meaning of content-independence is that user interaction does not depend on the content of the source and destination View Containers. In implementation terms, it is not necessary to associate parameter values with the interaction in order to compute the content of the target View Container.” [13].

During the process of execution, we adopt a number of technologies in order to achieve the final goal which is the automatic execution of UIs models via what we call IFVM virtual machine. The process considers two major concepts which are compilation and interpretation. In order to take the benefit from the compilers and the interpreters, many designers have mixed the compilation with the interpretation.

Generally, the implementation starts with a compiler that translates the source language into a target universal language called bytecode, which is closer to the machine language but it doesn’t depend to the machine, then an interpreter that takes care to run this program in a target language. We can have compilers for different languages that produce bytecode for the same virtual machine, and we can then run the same bytecode programs of the source language to all the processors for which there are interpreters. We outline, in the next section, the bytecode proposed as the instruction set of the VM with some details, then we introduce the IFVM basic principles.

4.1 IFVM Bytecode

There are a plenty of virtual machine architectures available on the market, especially those being stack-based architectures without registers. Examples of widely known virtual machines are Java's JVM [15], Android's Dalvik VM [16] and Python’ VM.

Generally, stack-based virtual machines architectures without registers put values directly onto the stack. The reason behind not using registers is that each CPU design has its own number of registers, so, a stack-based virtual machine could run on any CPU design with a stack. This advantage makes stack VM extremely simple, powerful and portable, since it relies on pushing and popping values on the stack. It is hardware and operating system independent, thing that makes it possible to run the same bytecode on multiple platforms using different interpreters, as well as a stack-oriented interpreter is relatively easy to implement.

As mentioned before, each element from the IFML model will be expressed in a bytecode form to be interpreted by a virtual machine in order to obtain the machine instructions understood by a computer's processor. We propose a new definition of the bytecode or what we call the instruction set of the present virtual machine. For that, we opt for a stack-based architecture for conceiving the IFVM architecture. However, since the IFML expresses the three elements composing the application front-end; the content via view elements, user interaction and control behavior; those elements have to be mapped to the IFVM bytecode. The source IFML model of a complex UI structure is transformed to build the target bytecode model by mapping each individual element to its equivalent representation that could contain several IFVM instructions. Properties inside IFML expressions could be mapped as values pushed and popped on the stack.

After a deep analysis on a number of well-known VMs, we derive a syntax of IFVM instruction set. It is similar to Java Bytecode syntax, since JVM relies on a stack-based architecture. New instructions have been added to control expressions for invoking or calling methods during the UI construction, such as instructions of type invoke that are used for this situation after storing all the required values in the stack with push instruction. Table 1 outlines the top used instructions of IFVM bytecode. Moreover, we provide corresponding instructions for expressing content independent navigation between view containers that could be caused after an event of type View Element Event triggered. An event may be produced by a user interaction (View Element Event, On Submit Event, On Select Event), by an action when it finishes (Action Event) or by the system (System Event). We will focus on the first type of events especially the View Element Event. The event instruction with indication of the event type, and the navigate instruction that specifies the target of the navigation are used for this purpose.

Table 1. IFVM instruction set

Mnemonic	**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
Event S	→ Event Object, Triggering Expression	Put on the top of the stack the expression triggering an event of type System Event and an event object
Event A	→ Event Object, [param1, param2]	Put on the top of the stack a set of Parameter Binding needed after an event of type Action Event is triggered, and an event object
Event V	→ Event Object	Create an event object of type View Element Event. There is no need of Parameter Binding since it corresponds to content independent navigation
Event Select	→ Event Object, [param1, param2,	Put on the top of the stack a set of Parameter Binding needed after an event of type On Select Event is triggered, and an event object
Event Submit	→ Event Object, [param1, param2,]	Put on the top of the stack a set of Parameter Binding needed after an event of type On Submit Event is triggered, and an event object
Navigate	→ Target Navigation	Put on the top of the stack the target of the navigation
Store_n	Object →	Store an object into local variable n
Load_n	→ Object	Load an object from local variable n

Regarding the five types of triggered events cited in table1, we will focus in this present work only on the basic form of event which is the View Element Event. It could be triggered after a user interaction to navigate into another View Container without passing parameters, we talk about the content independent navigation.

4.2 IFVM Execution

In the present work, we also followed the same concept of mixing the compilation and interpretation to build the IFVM virtual machine under a model driven architecture. The process consists of two major units: the compilation unit and the interpretation unit. (see Figure 1)

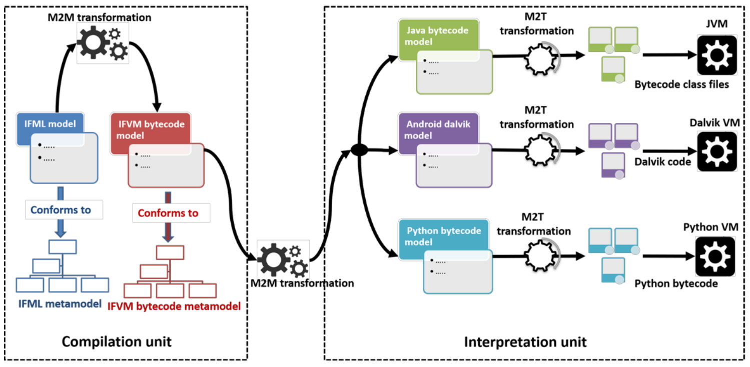


Fig. 1. IFVM Virtual machine process

Compilation Unit: After representing the UIs through IFML models conforming to IFML metamodel, comes the stage of the compilation, i.e. the translation to the IFVM bytecode. According to the model driven architecture adopted, compilation is done via a model to model transformation designed with QVTo (Query/View/Transformation) [17]. We started by elaborating the IFVM bytecode metamodel (see Figure 2) according to the instruction set detailed in Table 1.

The IFVM bytecode metamodel admits iroot Meta class as a root Meta class through which we instantiate all the IFVM machine instructions. Instructions fall into seven groups: Load and store, Arithmetic and logic, Type conversion, Object creation and manipulation, Operand stack management, Event and navigation, Method invocation and return.

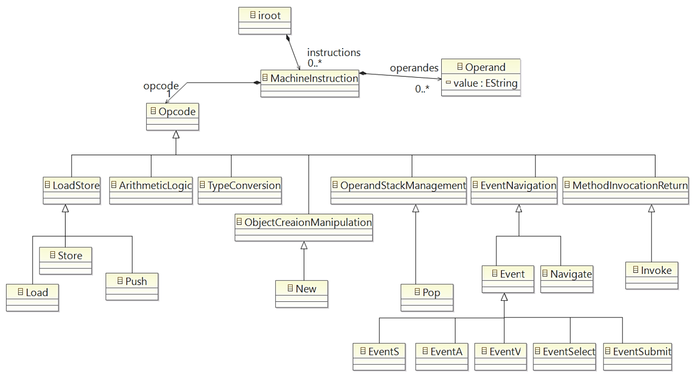


Fig. 2. IFVM bytecode metamodel

A model to model transformation, taking IFML metamodel as input, is lunched to generate the equivalent bytecode model according to the IFVM bytecode metamodel. Several rules were applied in the model to model transformation algorithm. The algorithm below shows an extract of it.

```
Input ifml : IFML
Output ibytecode : IFVMBytecode
begin
  map ifmlmodelToiroot(ifml.interactionFlowModel);
end
mapping ifmlmodelToiroot(imodel:interaction FlowModel:iroot
begin
for all w ϵ imodel.interactionFlowModelElements
        if w is window
            map windowToNew(w)
        end if
end for
end
mapping windowToNew(w:window): new
begin
        foreach p ϵ w.properties
      map propertyToPush(p)
        end for
        create store_i object //Store w into variable i
        for all e ϵ w.viewElementEvents // form of event
            map eventToeventV(e)
     map navigationFlowToNavigate(e.NavigationFlow)
        end for
        for all e ϵ w.viewElements
      if e is form or list or details
        map elementToNew(e)
        create load_i object //Load window w from i
        create invoke object //Binding w with e
      end if
        end for
end
```

The transformation algorithm is based on the mapping between the source and target metamodels elements. So, the interaction flow model will be mapped to iroot element in the bytecode model, its windows will be then transformed into machine instructions of type New to instantiate the frame, and other instructions of type Invoke that calls the constructor of the frame, and instructions of type Push to store the properties on the stack. Next, we can map the rest of the View Components as we did before. Regarding the link between the View container and the view components, it could be mapped as machine instructions of type Load to load onto the stack the references of these latter followed by Invoke machine instruction that calls the method responsible for making the binding. Each IFML event will be mapped to its corresponding IFVM bytecode event instruction followed by mapping the caused NavigationFlow to Navigate instruction for expressing the target frame to be shown as detailed in the previous algorithm.

Interpretation Unit: The IFVM bytecode was chosen as being an intermediate representation for the present virtual machine in order to gain optimization and portability. We mean by portability, the possibility to represent the obtained bytecode in different platform independent or dependent forms. However, to benefit from the advantages of existent VMs; we cite Java Bytecode, Android Bytecode and so on, we decided to transform the bytecode result into these VMs platform independent bytecode.

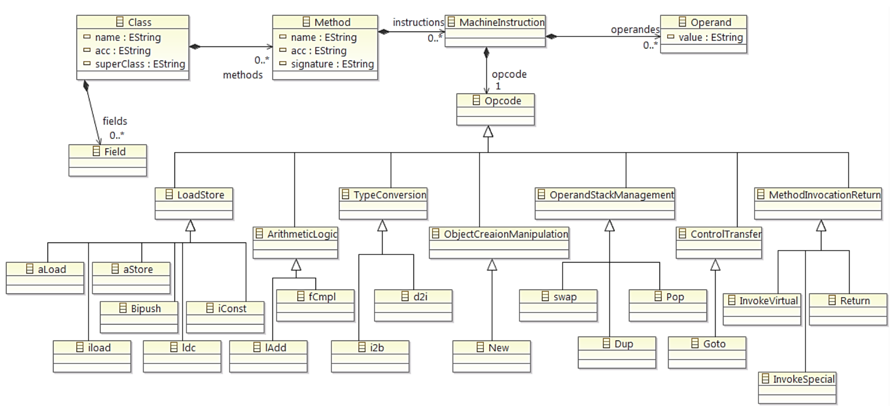


Fig. 3. Java Bytecode metamodel

As shown in Figure 1, after the execution of the compilation unit, comes the interpretation step, in which we execute another model to model transformation implemented by QVTo language. It admits as input the IFVM bytecode obtained from previous step, and as output the models of other bytecode forms of existing VMs.

In this present work, we focused on the mapping into the Java bytecode model that respects its metamodel figured in Figure 3. For that, we set up a number of transformation rules. Table 2 shows an extract of correspondence between elements from the two metamodels.

Table 2. Mapping between IFVM bytecode instructions and JVM bytecode instructions

IFVM Bytecode					Java Bytecode
push					bipush
pop					pop
new					new
invoke					Invoke Special (e.g. invoking constructor method) Invoke Virtual
Events	Event A	Event V	Event Select	Event Submit	Invoke Virtual (the action listener method)
navigate					new
store_n					astore_n
load_n					aload_n

The reason behind choosing the java virtual machine as our interpreter for IFVM is because of two things: firstly, the IFVM bytecode used in the process is similar to the JVM instruction set syntax. Secondly, is because the JVM represents an interface that allows running the same bytecode on any platform, so, "write once, run anywhere" JVM makes it become a reality.

After the construction of the java bytecode model, comes the stage of the interpretation by Java virtual machine. It needs, for the execution, a set of machine instructions written in Java bytecode format. So, there is a need to translate the obtained Java bytecode model into its equivalent bytecode format. In other words, a model to text transformation can be launched to generate the java bytecode instruction set. We opted for a code generation using the open-source Acceleo [18], an Eclipse implementation of the OMG MOF2Text Transformation Language that maps model elements into text instructions.

The generation starts by invoking the template that includes the necessary code to be generated by the transformation. Indeed, depending on the bytecode structure represented as byte arrays, manipulating it through the Acceleo template is considered difficult and is usually performed using libraries, that's why we thought to use the ASM library [19] that facilitates the bytecode manipulation and analysis. So, instead of writing the bytecode structure in the Acceleo template which is difficult, we incorporate, in the template, the program using the ASM library that generates dynamically and directly the bytecode class file.

5 Running Example

In order to validate our process of IFVM virtual machine, we applied it to execute the UI's representation of a Library System. The application allows user to manage a database library by adding, listing, searching for books through user interfaces. Figure 4 illustrates the IFML model; using IFML Editor; of navigation between the toolbar elements and the other views expressed with events.

The UI principal window contains a top-level toolbar permitting five actions: add book, add member, list books, list members, search book. During this section, we will focus on the action of Adding book. The triggered event causes a content independent navigation targeting the display of Add Book form.

So, we start by elaborating the view design model of the entire system UIs through an instance of IFML metamodel, that is made up of elements figured in Figure 2. Figure 5 shows the corresponding IFML model.

Once the model has been built, it is then the time to get it as input in the compilation unit to be mapped into the IFVM bytecode model with a set of machine instruction. Figure 6 illustrates the compilation unit result.

The result IFVM bytecode model will be taken, in its turn, as input of the interpretation unit.

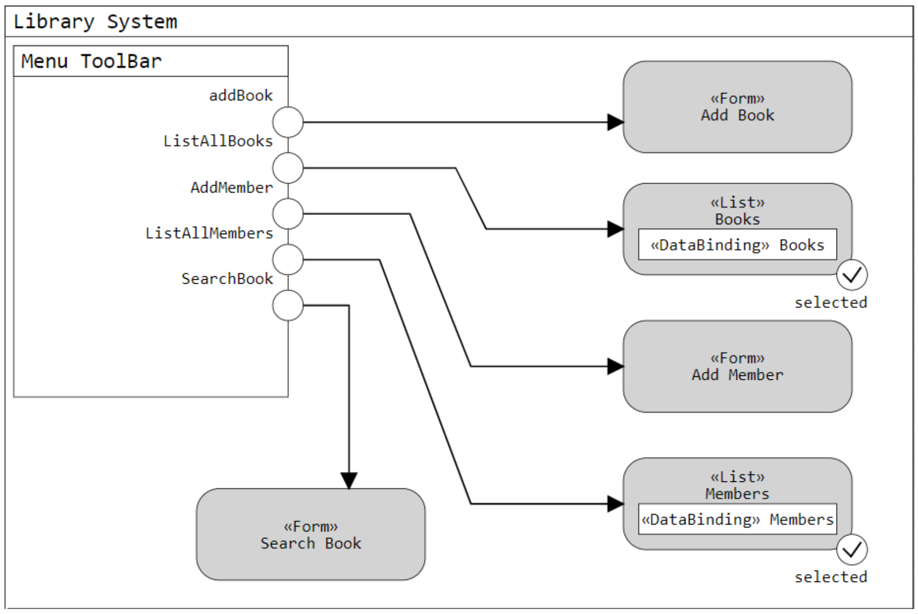
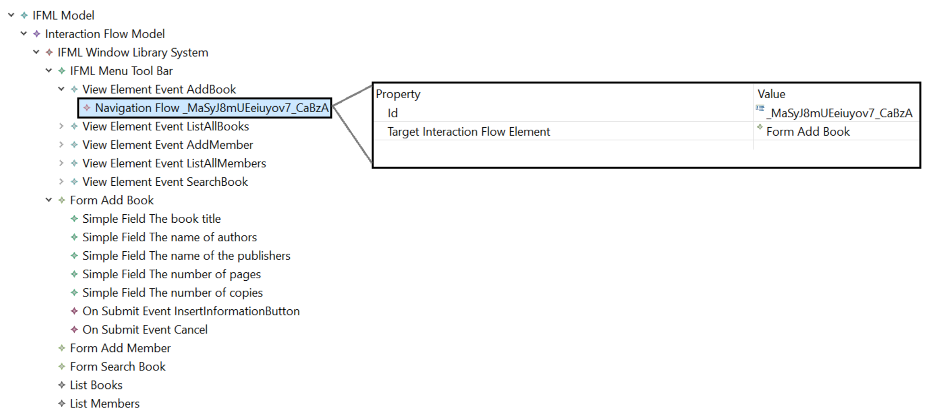


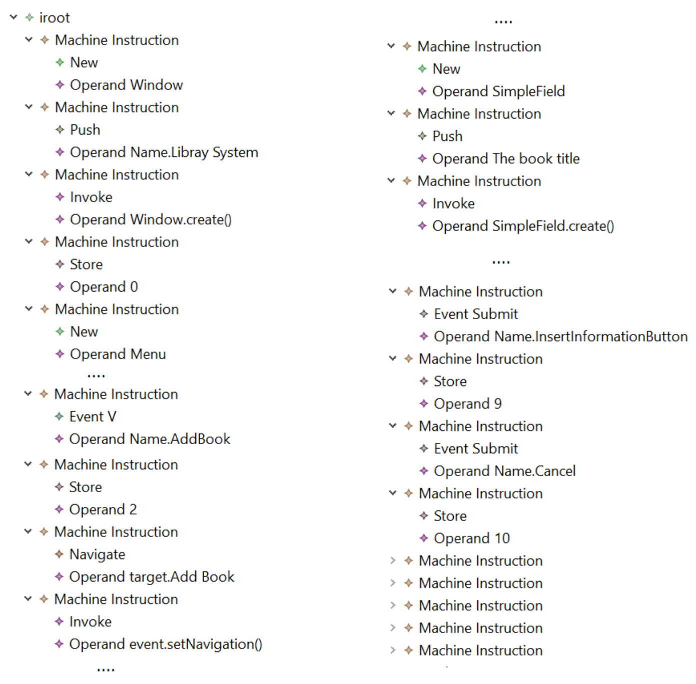
Fig. 4. Navigation flows between View Components of Library System



```
IFML Model
  Interaction Flow Model
    IFML Window Library System
      IFML Menu Tool Bar
        View Element Event AddBook
          Navigation Flow _MaSyJ8mUEeiuyov7_CaBzA
        View Element Event ListAllBooks
        View Element Event AddMember
        View Element Event ListAllMembers
        View Element Event SearchBook
      Form Add Book
        Simple Field The book title
        Simple Field The name of authors
        Simple Field The name of the publishers
        Simple Field The number of pages
        Simple Field The number of copies
        On Submit Event InsertInformationButton
        On Submit Event Cancel
      Form Add Member
      Form Search Book
      List Books
      List Members
```

Property	Value
Id	_MaSyJ8mUEeiuyov7_CaBzA
Target Interaction Flow Element	Form Add Book

Fig. 5. IFML model of UI library system

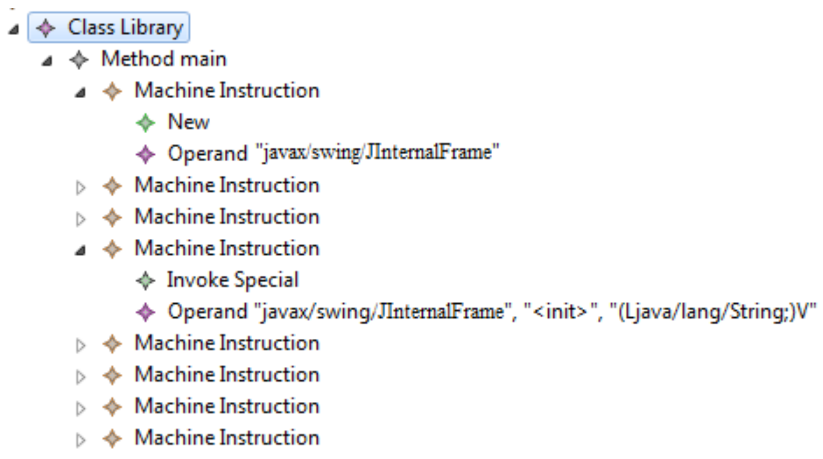


```
iroot
  Machine Instruction
    New
    Operand Window
  Machine Instruction
    Push
    Operand Name.Libray System
  Machine Instruction
    Invoke
    Operand Window.create()
  Machine Instruction
    Store
    Operand 0
  Machine Instruction
    New
    Operand Menu
    ....
  Machine Instruction
    Event V
    Operand Name.AddBook
  Machine Instruction
    Store
    Operand 2
  Machine Instruction
    Navigate
    Operand target.Add Book
  Machine Instruction
    Invoke
    Operand event.setNavigation()
    ....

    ....
  Machine Instruction
    New
    Operand SimpleField
  Machine Instruction
    Push
    Operand The book title
  Machine Instruction
    Invoke
    Operand SimpleField.create()
    ....
  Machine Instruction
    Event Submit
    Operand Name.InsertInformationButton
  Machine Instruction
    Store
    Operand 9
  Machine Instruction
    Event Submit
    Operand Name.Cancel
  Machine Instruction
    Store
    Operand 10
  Machine Instruction
  Machine Instruction
  Machine Instruction
  Machine Instruction
  Machine Instruction
```

Fig. 6. IFVM bytecode result of compilation unit

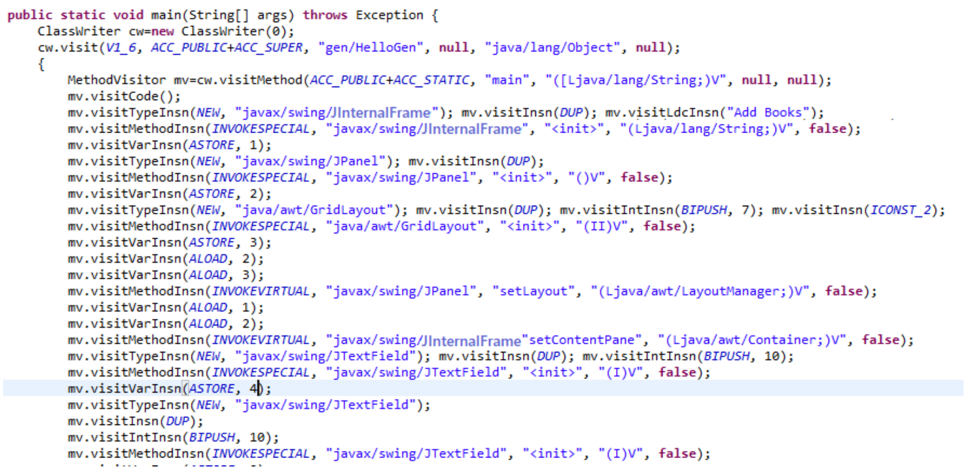
We launch a model to model transformation to generate a model of Java bytecode instruction set. Each view will be treated in separate java bytecode model to facilitate then the generation of class files to be inserted in JVM virtual machine to be interpreted. Figure 7 shows one of the generated java bytecode models that represents the Add Book form.



```
Class Library
  Method main
    Machine Instruction
      New
      Operand "javax/swing/JInternalFrame"
    Machine Instruction
    Machine Instruction
    Machine Instruction
      Invoke Special
      Operand "javax/swing/JInternalFrame", "<init>", "(Ljava/lang/String;)V"
    Machine Instruction
    Machine Instruction
    Machine Instruction
    Machine Instruction
```

Fig. 7. Java bytecode model of Add book UI

After that, comes the stage of the model to text transformation in order to generate the equivalent bytecode class file by applying ASM library instructions. Figure 8 shows a portion of the generated ASM program that will generate the equivalent bytecode class file for the Add Form J Internal Frame later.



```
public static void main(String[] args) throws Exception {
    ClassWriter cw=new ClassWriter(0);
    cw.visit(V1_6, ACC_PUBLIC+ACC_SUPER, "gen/HelloGen", null, "java/lang/Object", null);
    {
        MethodVisitor mv=cw.visitMethod(ACC_PUBLIC+ACC_STATIC, "main", "([Ljava/lang/String;)V", null, null);
        mv.visitCode();
        mv.visitTypeInsn(NEW, "javax/swing/JInternalFrame"); mv.visitInsn(DUP); mv.visitLdcInsn("Add Books");
        mv.visitMethodInsn(INVOKESPECIAL, "javax/swing/JInternalFrame", "<init>", "(Ljava/lang/String;)V", false);
        mv.visitVarInsn(ASTORE, 1);
        mv.visitTypeInsn(NEW, "javax/swing/JPanel"); mv.visitInsn(DUP);
        mv.visitMethodInsn(INVOKESPECIAL, "javax/swing/JPanel", "<init>", "()V", false);
        mv.visitVarInsn(ASTORE, 2);
        mv.visitTypeInsn(NEW, "java/awt/GridLayout"); mv.visitInsn(DUP); mv.visitIntInsn(BIPUSH, 7); mv.visitInsn(ICONST_2);
        mv.visitMethodInsn(INVOKESPECIAL, "java/awt/GridLayout", "<init>", "(II)V", false);
        mv.visitVarInsn(ASTORE, 3);
        mv.visitVarInsn(ALOAD, 2);
        mv.visitVarInsn(ALOAD, 3);
        mv.visitMethodInsn(INVOKEVIRTUAL, "javax/swing/JPanel", "setLayout", "(Ljava/awt/LayoutManager;)V", false);
        mv.visitVarInsn(ALOAD, 1);
        mv.visitVarInsn(ALOAD, 2);
        mv.visitMethodInsn(INVOKEVIRTUAL, "javax/swing/JInternalFrame"setContentPane", "(Ljava/awt/Container;)V", false);
        mv.visitTypeInsn(NEW, "javax/swing/JTextField"); mv.visitInsn(DUP); mv.visitIntInsn(BIPUSH, 10);
        mv.visitMethodInsn(INVOKESPECIAL, "javax/swing/JTextField", "<init>", "(I)V", false);
        mv.visitVarInsn(ASTORE, 4);
        mv.visitTypeInsn(NEW, "javax/swing/JTextField");
        mv.visitInsn(DUP);
        mv.visitIntInsn(BIPUSH, 10);
        mv.visitMethodInsn(INVOKESPECIAL, "javax/swing/JTextField", "<init>", "(I)V", false);
```

Fig. 8. Extract of generated ASM program

Finally, the resulting class files will be considered as input for interpretation by java virtual machine. Figure 9 shows the execution result of tested system.

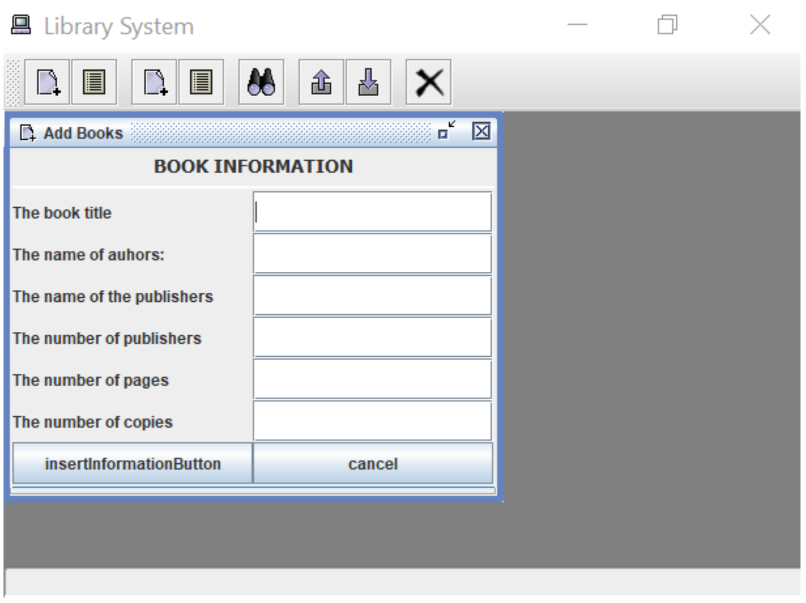


Fig. 9. Add Book UI result execution

Our contribution for automatically executing the abstract representation of UIs has made running, building and maintenance of software UIs easier and less expansive in comparison to the classic approach by manually developing the UIs.

6 Conclusion

In this paper we presented a new concept for executing user interfaces and interactions designed with IFML OMG standard. We described a model driven development process named IFVM. The process admits as input, within the compilation unit, the IFML model to be mapped into IFVM bytecode model using QVTo language. Then, in the interpretation unit, we launch another model to model transformation to generate the other forms of well-known bytecode models, we focused on just one mapping that generates java bytecode model. After that, we generate the bytecode class file from the obtained java bytecode model through Acceleo language and the ASM library. Finally, the result bytecode class file will be passed into the JVM to be interpreted.

We concentrated our treatment on just one type of navigation which is the content independent navigation expressed by View Element Event. Future works will cover the integration of the other forms of navigation with data transferred between windows, and we think to combine also the back-end with the front-end representation of the systems in order to produce a complete model driven executable system.

7 References

[1] OMG, Interaction Flow Modeling Language. Version 1.0. IFML (2015), available at http://www.omg.org/spec/IFML/1.0/

[2] OMG, MDA. "MDA Guide Version 2.0." ,2015.

[3] Gotti, S., & Mbarki, S. (2016, March). UML executable: a comparative study of UML compilers and interpreters. In Information Technology for Organizations Development (IT4OD), 2016 International Conference on (pp. 1-5). IEEE. https://doi.org/10.1109/IT4OD.2016.7479251

[4] Roubi, S., Erramdani, M., & Mbarki, S. (2016). A Model Driven Approach based on Interaction Flow Modeling Language to Generate Rich Internet Applications. International Journal of Electrical and Computer Engineering (IJECE), 6(6), 3073-3079. https://doi.org/10.11591/ijece.v6i6.10541

[5] Laaz, N., & Mbarki, S. (2016, September). Combining Ontologies and IFML Models Regarding the GUIs of Rich Internet Applications. In International Conference on Artificial Intelligence: Methodology, Systems, and Applications (pp. 226-236). Springer, Cham. https://doi.org/10.1007/978-3-319-44748-3_22

[6] Acerbis, R., Bongio, A., Butti, S., & Brambilla, M. (2015, May). Model-driven development of cross-platform mobile applications with WebRatio and IFML. In Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems (pp. 170-171). IEEE Press.

[7] Wakil, K., & Jawawi, D. N. (2017). Extensibility interaction flow modeling language metamodels to develop new web application concerns. Kurdistan Journal of Applied Research, 2(3), 172-177. https://doi.org/10.24017/science.2017.3.23

[8] Gotti, Z., & Mbarki, S. (2016). Java Swing Modernization Approach-Complete Abstract Representation based on Static and Dynamic Analysis. In ICSOFT-EA (pp. 210-219). https://doi.org/10.5220/0005986002100219
[9] Gotti, S., & Mbarki, S. (2016). Toward IFVM Virtual Machine: A Model Driven IFML Interpretation. In ICSOFT-EA (pp. 220-225).
[10] Sottet, J. S., Calvary, G., & Favre, J. M. (2006). Models at runtime for sustaining user interface plasticity. In Models@ run. time workshop (in conjunction with MoDELS/UML 2006 conference).
[11] Selic, B. (2003). The pragmatics of model-driven development. IEEE software, 20(5), 19-25. https://doi.org/10.1109/MS.2003.1231146
[12] Shaer, O., Jacob, R. J., Green, M., & Luyten, K. (2008, April). User interface description languages for next generation user interfaces. In CHI'08 Extended Abstracts on Human Factors in Computing Systems (pp. 3949-3952). ACM.
[13] Brambilla, M., & Fraternali, P. (2014). Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML. Morgan Kaufmann.
[14] Cariou, E., Ballagny, C., Feugas, A., & Barbier, F. (2011, June). Contracts for model execution verification. In European Conference on Modelling Foundations and Applications (pp. 3-18). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-21470-7_2
[15] Lindholm, T., Yellin, F., Bracha, G., & Buckley, A. (2014). The Java virtual machine specification. Pearson Education.
[16] Bornstein, D. (2008, May). Dalvik vm internals. In Google I/O developer conference (Vol. 23, pp. 17-30).
[17] OMG, Q. V. T. (2008). Meta object facility (mof) 2.0 query/view/transformation specification. Final Adopted Specification (November 2005).
[18] Acceleo. Available online at: http://www.eclipse.org/acceleo/documentation/
[19] Bruneton, E. (2007). ASM 3.0 A Java bytecode engineering library. URL: https://download.forge.ow2.org/asm/asm3-guide.pdf.

8 Authors

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 received his B.S. degree in applied mathematics from Mohammed 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.

Article submitted 13 October 2018. Resubmitted 17 November 2018. Final acceptance 05 January 2019. Final version published as submitted by the authors.