Compact and Efficient Modeling of GUI, Events and Behavior Using UML and Extended OCL

Dong Liang, Bernd Steinbach
Institute of Computer Science
Freiberg University of Mining and Technology
Freiberg, Germany
email: liang@mailserver.tu-freiberg.de, steinb@informatik.tu-freiberg.de

***Abstract*—The model driven architecture (MDA) allows to move the software development form the time consuming and error-prone level of writing program code to the next higher level of modeling. The MDA requires tools for modeling, transformation of models, and code generation. In the past, we have developed such tools successfully. Using these tools we recognized serious problems preparing concise, uniform, and complete models using the unified modeling language (UML). In detail these problems concern first the specification and parameterization of GUI elements, second the event handling, and third the modeling of the required behavior. In this paper we show efficient solutions for these problems using the object constraint language (OCL) together with the UML for modeling. While the parameterization of GUI elements can be solved with the OCL directly, the last two problems were solved by an extension of the OCL into an executable OCL, which we call XOCL. We show the benefits of all three new approaches by means of an example of a complete platform independent model (PIM).**

***Keywords*-OCL extension, action language, event handling, platform independent model, class diagram**

I. INTRODUCTION

Traditionally, developing software means writing code in one of the programming languages. Recently, a novel approach called MDA (Model Driven Architecture) [6], which is proposed by OMG (Object Management Group), has evoked more and more attentions in software development. Instead of writing code directly, MDA suggests that software developers model their software products in a platform independent way. Such a software model is called Platform Independent Model (PIM). Then an MDA-tool is used to transform the PIM into one or more Platform Specific Models (PSM). The PSMs have involved detailed information for implementation. Hence, code generation from a PSM is straightforward. In order to realize model transformation, two different strategies are feasible. One of them defines the model transformation process in a high level specification language. The QVT (Query, View and Transformation) language [7][8] supported by OMG is the de facto standard for this strategy. The QVT allows to define a transformation in an imperative approach. Then a QVT compiler generates an implementation (e.g., in Java) of this transformation specified in the QVT source file as a model compiler, which is dedicated to this transformation. The other strategy is to develop the model compiler itself as an all-purpose model transformation framework as well as to provide all necessary information about the underlying target platform the PSM based on, in the form of Target Platform Models (TPM). To prove the second strategy, the model compiler MOCCA (Model Compiler for reConfigurable Architecture) [9][10] was developed in our institute. According to the actual state of MOCCA, a PIM can be transformed into C++ code, Java code for software design, as well as C++/VHDL code for software hardware co-design.

For the both strategies, one issue is still challenging. That is, how to create a PIM concisely, uniformly and completely. In this paper, we assume that all the PIMs are modeled using the Unified Modeling Language (UML) [5] and the Object Constraint Language (OCL) [4]. Based on our experiences with MOCCA, three sub-issues concerning creating PIMs are found.

1) Modeling GUI-layout of a GUI-based application and parameterize all the GUI elements in standard UML is a serious problem, because the model must contain the structural composition of GUI as well as all the geometrical and visual information of the GUI elements. There is no UML diagram type appropriate for both of these aspects.
2) Modeling event handling for GUI-based application in standard UML is time-consuming, because almost all programming languages have their own GUI libraries and the underlying event models and their details blow up the UML-model in an unnecessary manner.
3) Modeling behaviors in PIM concisely is very difficult, because on the one hand there is no standard universal action language based on the UML Action Semantics [5]; on the other hand, specifying behaviors using one of the UML behavioral diagrams can result in a model more complex than the target codes themselves.

In order to solve these problems, developing new modeling approaches based on UML and OCL are the main aims of this paper. In Section II, a new approach will be introduced that uses UML Class Diagram and OCL-init-

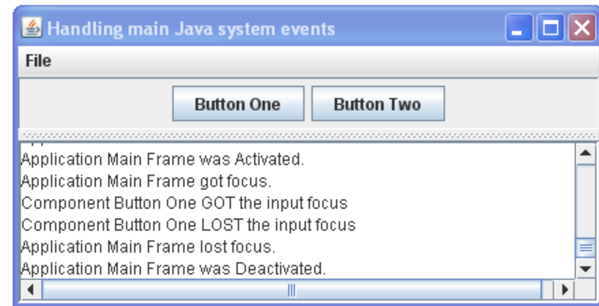
2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

expressions to model and to parameterize GUI elements. In Section III, the OCL is extended by the ability to register event handlers for an event source. This approach helps to create real PIMs for MDA-technology. In Section IV, the OCL is upgraded from a pure declarative language without side effects into an action language, which can be used to specify all kinds of operations in a concise and platform independent manner. In Section V we summarize these three new approaches in an example of a complete PIM.

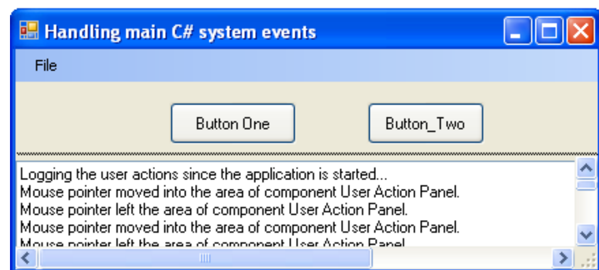
II. OCL-INIT-EXPRESSIONS – OUR NEW APPROACH TO PARAMETERIZE GUI ELEMENTS

In GUI-based application the user interacts usually with a window containing different kinds of GUI elements, which are typically menu items, icons, buttons, input fields etc. According to different platforms, these GUI elements may also be called GUI components or GUI controls. A GUI element usually represents a certain graphical entity that can be displayed on the screen. So they typically have parameters, which concern displaying them on the screen correctly. Such parameters are position, size, background- and foreground-color etc. Most modern object-oriented programming languages implement common GUI elements as classes and their important properties as attributes of the corresponding classes. They are deployed in libraries and can be used in certain languages. Hence, the programmers can use them directly. Figure 1 shows such a GUI-based application implemented in both Java and C#. Both implementations are similar in appearance, because for each GUI element involved in Java implementation, there is a C# counterpart. This analysis gives us a heuristic to model GUI in a platform independent manner. Since there are so many common points among GUI elements on different platforms, we can abstract a platform independent GUI tool-kit with most common GUI elements and their important properties for the general usage to develop a PIM for an application.

In fact, MOCCA supports this principle inherently. A Design Platform Model (DPM) [9][10] contains the most basic design types for primitive data types, IO facilities etc., has been used to establish PIMs. Such a GUI toolkit is just another extension of MOCCA DPM, which is still being developed. The class diagram in Figure 2 models the GUI elements of the application in Figure 1 in a completely platform independent manner. The types with prefix *DP*, which means Design Platform, are the common GUI elements involved in the GUI toolkit of MOCCA DPM. For example, the design type *DPWindow* can be considered as a common abstraction of both *JFrame* in Java and *Form* in C#. The properties defined in *DPWindow*, such as *length* and *height*, are used to model GUI elements exactly. The tool used to create this model is our own CASE Tool called *UML 2 Designer*. However, modeling in this way shows neither the composition structure nor the visualization of the GUI elements on the screen. The missing information has to be involved in the design model in a reasonable way.



(a) Java Implementation



(b) C# Implementation

Figure 1. A GUI-based application implemented in Java and C#

Implementing GUI-layout as source code using a modern object-oriented programming languages is as complicated as modeling it in a design model. The powerful modern IDEs usually solve this problem by integrating an additional software component, which supports visual manipulation of GUI-elements. For C# the *Form Editor* of Visual Studio IDE can be used for this purpose, whereas the *Swing GUI Builder* integrated into the NetBeans IDE is the counterpart for Java. Both of them support programmers in a similar way. For a class implementing the GUI-layout of an application, e.g., the derived class of the *Form* base class in C#, there is a *design view* associated with it. The programmer chooses the required GUI elements from a *Toolbox*, which contains all the supported GUI elements in this context, positions them on the form, and edits them visually. Figure 3 shows how to use the Form Editor to edit the GUI elements of the application in Figure 1 (b) in a manner explained above.

After editing the GUI elements, all the geometrical and visual information are stored in the *Property Window*, which is visible on the right side of Figure 3. The Visual Studio generates C# codes automatically, which reflect the visual manipulation of the GUI elements done in the design view. These generated program statements are displayed in the C# source code view. Figure 4 shows the generated C# codes initializing the first button in Figure 1 (b).

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

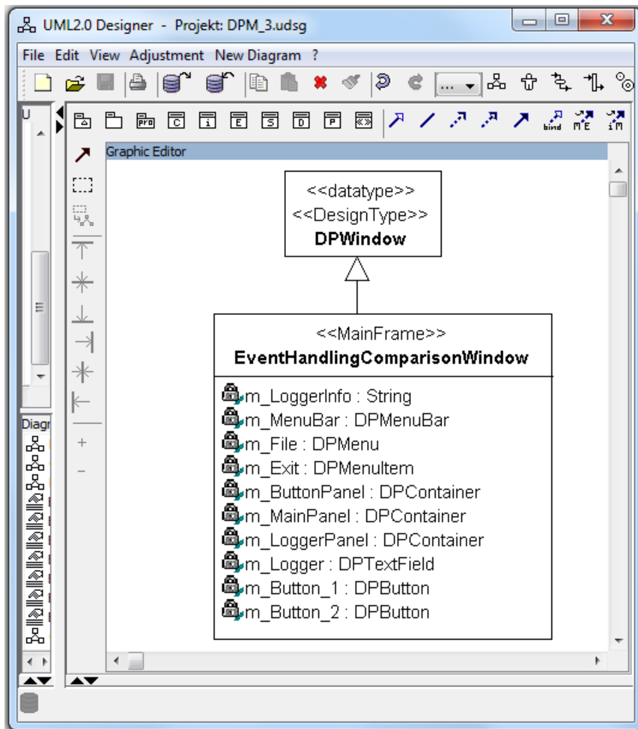


Figure 2. Platform independent model for the main window of the application in Figure 1

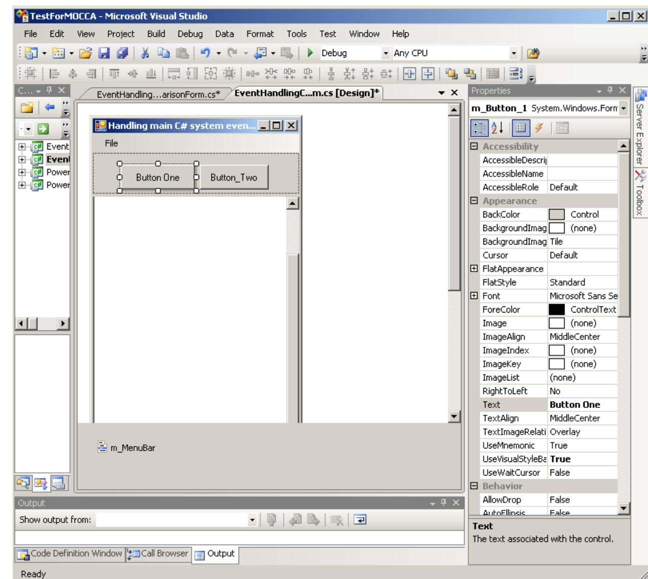


Figure 3. The Form Editor of Visual Studio 2008 used to edit the GUI elements of the application in Figure 1 (b)

```
#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    // m_Button_1
    this.m_Button_1 = new System.Windows.Forms.Button();
    this.m_Button_1.Location = new System.Drawing.Point(35, 15);
    this.m_Button_1.Name = "m_Button_1";
    this.m_Button_1.Size = new System.Drawing.Size(93, 31);
    this.m_Button_1.TabIndex = 0;
    this.m_Button_1.Text = "Button One";
    this.m_Button_1.UseVisualStyleBackColor = true;
    this.m_Button_1.Click +=
            new System.EventHandler(this.m_Button_1_Click);
}
```

Figure 4. C# code generated by Visual Studio Form Editor

It is easy to understand that the *Swing GUI Builder* generates Java codes to reflect the visual manipulation of the Swing components. It is clear that both scenarios mentioned above are platform specific. Based on the considerations above, we suggest a solution to model the GUI-layout in the phase of establishing the PIM of a GUI-based application, which can be summarized as follows:

- A platform independent GUI toolkit is required that contains common GUI elements and their properties as the building blocks of a PIM. As explained, the prototype of this GUI toolkit has been created for our MOCCA DPM and can be used directly in our CASE Tool UML 2 Designer.
- The logical structure of an application window containing various GUI elements can be modeled in UML class diagram, as shown in Figure 2.
- The GUI-layout of an application window can be visually manipulated in a *view* associated to the UML class, which models the logical structure of an application window. This additional view is an add-on software component of the UML 2 Designer, whose prototype has been developed in a bachelor thesis and its upgrade version will be developed in another bachelor thesis. We call this software component the *Smart GUI Editor*. Figure 5 shows its usage. It is easy to understand that the Smart GUI Editor shares the same principles as the Form Editor and Swing GUI Builder.
- In contrast to the Form Editor in Visual Studio, which can produce C# code to reflect the visual manipulation, a platform independent manner is required for our Smart GUI Editor, which can be transformed easily into target code in the later phase of model transformation. As solution, we suggest *using OCL-init-expressions to represent the information generated by our Smart GUI Editor*.

As explained in [3], the OCL-init-expressions can be used to give the initial values of attributes or association ends of a class at the moment that an instance of this class is created. Hence, the original OCL-init-expressions are usually attached to the properties of a class as their context. In this paper, the syntax of the original OCL-init-expressions have been slightly extended such that all the automatically generated OCL-init-expressions are attached to

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

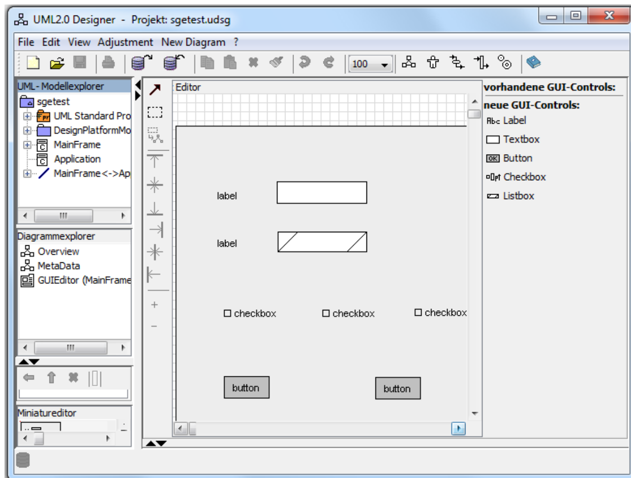


Figure 5. The Smart GUI Editor used to visually manipulate the GUI elements modeled in UML class diagram

the class directly, which is in our application a *GUI window*. According to our code-generation strategy, for initializing the properties of the GUI window itself, the Smart GUI Editor will generate an OCL-init-expression concatenating all the properties using OCL *and* operator. For each GUI-element contained in this window, the Smart GUI Editor will generate an additional OCL-init-expression specifying all its properties, again, connecting them by OCL *and* operator. The window instance can be retrieved by the OCL keyword *self*. Hence, the OCL-init-expressions in Listing 1 can be generated and attached to the class *EventHandlingComparisonWindow* as *constraints*. These OCL-init-expressions initialize the structural composition and geometrical information of the application window and its contained GUI elements *m_MainPanel*, which is a split panel, as well as the button *m_Button_1*. Other used GUI elements can be parameterized in the same way.

```
1  init: self.length = 480
2          and self.height = 629
3          and self.title = 'Main Window'
4
5  init: self.m_MainPanel.split = true
6          and self.m_MainPanel.horizontal = false
7          and self.m_MainPanel.owner = self
8
9  init: self.m_Button_1.posX = 35
10         and self.m_Button_1.poxY = 15
11         and self.m_Button_1.length = 93
12         and self.m_Button_1.height = 31
13         and self.m_Button_1.text = 'Button One'
14         and self.m_Button_1.owner = self.m_ButtonPanel
```

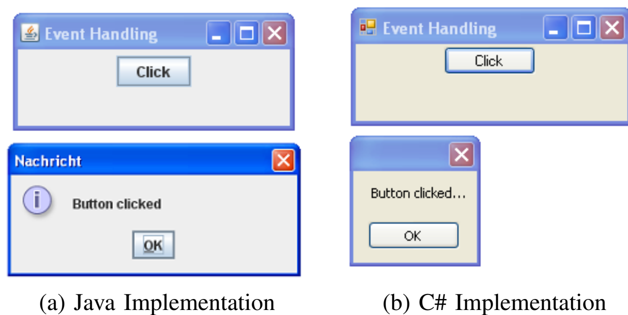
Listing 1. OCL–init–expressions to parameterize GUI elements

One more question must be answered. *How does the GUI-layout modeled in a PIM as suggested above make sense for the final GUI-layout mapped on a specific hardware platform?* In order to answer this question, we should make a difference between the GUI-layout and GUI-look-and-feel. For example, the both applications in Figure 1 do have the same GUI-layout but slightly different look-and-feels due to the underlying implementation platforms, say, Java-Swing and C#-FCL. Hence, modeling GUI-layout as suggested in this paper concentrates on the logical structure of an application window. That means which GUI-elements belong to which window, or to which panel etc. On the other hand, geometrical information can be modeled in a device independent coordinate system, which can be transformed onto concrete platform in the phase of model transformation by providing the model mapper with additional information about the underlying implementation platform.

III. OCL-Event-Expression – our new Approach to Model Event Handling

Another important issue related to model GUI-based application in PIM is to model event handling [1]. It is difficult to deal with this modeling issue by using standard UML and OCL.

At first view, it seems to be possible to use the OCL *isSent* operator (denoted by ˆ) to model the coupling of events to their handling methods. However, the *isSent* operator can only be used in the post condition [3] of an event sending method, e.g., *fireActionPerformed()* operation of the class *JButton* in Java. It is needlessly for application modelers to specify post conditions of this type of operations, because it does not belong to the application model, but to the used GUI library. Hence, the *isSent* operator is not appropriate to connect events of GUI elements to their handling methods. To model the coupling of events to their handling methods using class diagrams in a traditional way is a serious problem, too. This will be illustrated by examples.



(a) Java Implementation (b) C# Implementation

Figure 6. A simple GUI-based application implemented in Java and C#

A very simple GUI-application, which has been implemented in both Java and C#, is shown in Figure 6. There is a single button in the main window of the application. When this button is pressed, a message box will be launched to confirm this operation. The same behavior occurs by pressing the closing symbol of the main window. Even for a simple application like this, the corresponding PSMs in Java and C# are not the same. Figure 7 shows both models.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

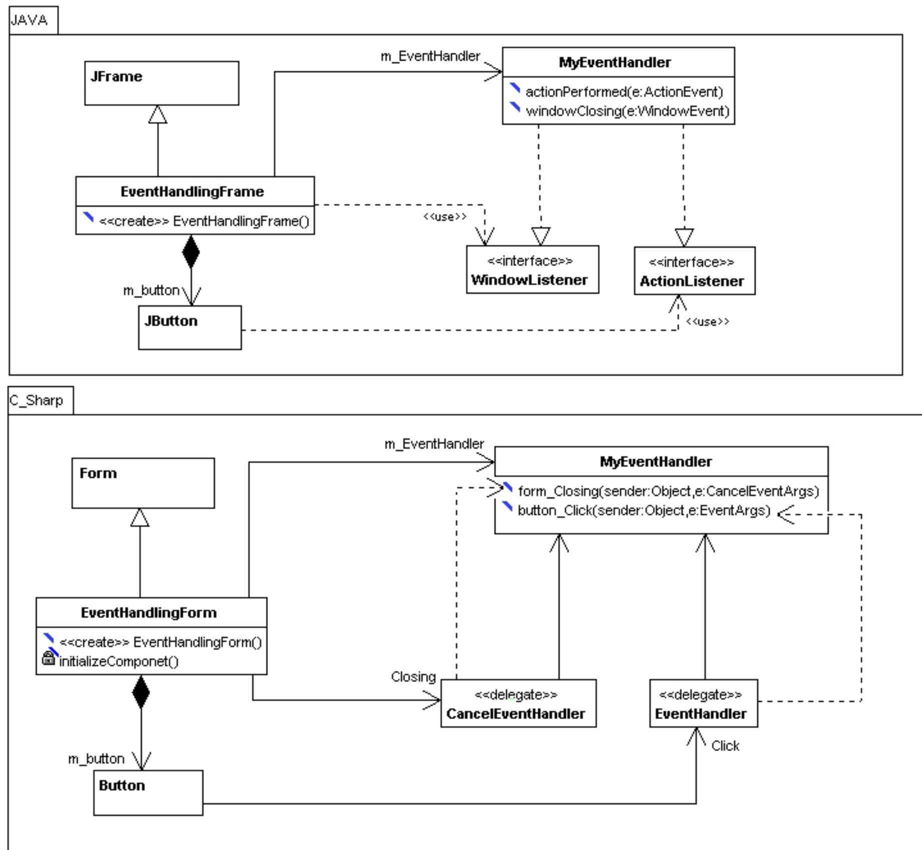


Figure 7. Java and C# models for the application of Figure 6

At first glance, these UML-models are similar to each other, because for each class and interface involved in the Java model, a counterpart can be found in the C# model. Only the number and types of the ”lines” between these elements are different. These differences are caused by the requirements of the languages. In Java, interfaces are used to connect events with their handler [12] while in C# delegates are used, which are in fact type safe callbacks based on function pointer [13]. The concept *delegate* is not supported by standard UML directly, so a stereotype must be defined in order to allow marking a class as delegate. In order to model the semantics of function pointers, two additional dependencies are used between the delegates and their pointed methods. The analysis of these UML models achieves an important conclusion: *the most complexities were brought into these models by modeling event handling in a too detailed manner.*

Taking into account the increased complexity of real GUI elements, modeling in such a way reduces the readability of class diagrams dramatically. The PSM for the Java implementation of the application in Figure 1 is shown in Figure 8. Due to poor readability, the corresponding C# PSM with increased complexity is not included in this paper. The Java PSM explores another remarkable drawback of traditional modeling of the event handling in class diagrams, which is: *only the three classes with colored background belong to the classes to develop; the other classes with white background are GUI-related library types*.

Due to these findings, a novel approach of modeling event handling in much simpler manner must be found. In this new approach, the tedious details explored in the PSMs must be hidden to the application modeler. The class diagram should contain as few library types as possible.

In order to find a unified and tight model for event-handling, a thorough understanding of the underlying event handling mechanisms is required. An event enables an object of a class (or a class itself) to publish changes of its state. Other objects and classes can then react to this change. This mechanism is usually called *Publishing – Subscription* model. Despite different implementations of this model in concrete programming languages, the entire event handling process can be divided into four parts [14]:

- *Static publishing* requires, that some kinds of events can be specified as members of their source. For example, events such as *Window Closing*, *Button Click* must be specified in GUI elements representing an application

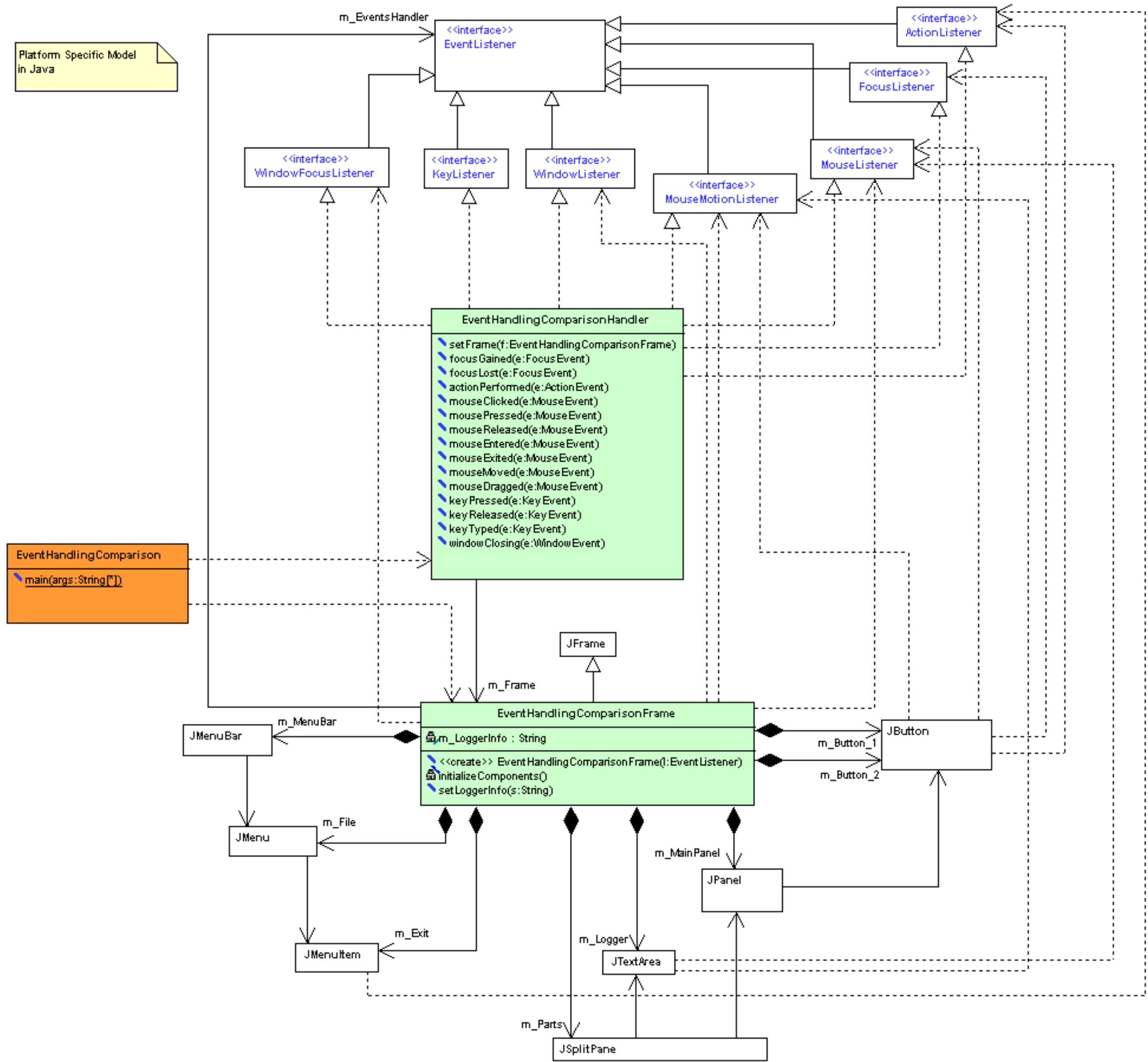


Figure 8. The PSM for the Java implementation of Figure 1

window or a button, respectively. This part is only important for customized events. The most significant GUI events have been defined by GUI developers.

- *Dynamic publishing* allows the transmission of the events. In both Java and C# this part is realized by a method, which triggers the execution of one or several dedicated event handling methods. Similar to static publishing, this part has been again implemented by GUI library designers.
- *Static subscription* requires the implementation of all event-handling methods. In fact, exactly this part specifies what must be performed when an event occurs. This part has to be modeled by application modeler. Along with dynamic publishing, this part belongs to behavioral specification of an application model and should be modeled compactly using some kind of high level action language, which will be addressed in next section.
- *Dynamic subscription* is done by establishing the connection between the event-source and the event-handling method. Such a process is often called registration of event handlers. Based on the analysis above,

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

modeling this part makes the class diagrams complex and heterogeneous, because various listener interfaces are used in Java to connect events with their handling methods loosely while C# delegates set up these connections directly. *The new approach discussed in this section is designed to simplify exactly this part of the entire event handling process*.

In order to develop a unified model for event-handler registration, a thorough comparison between Java and C# event models is completed to extract the similarities from them and to recognize the differences between them. The conclusion of this comparison can be summarized as follows:

1) In Java, the signature of an event-handling method is completely specified in one of the listener interfaces while C# *EventHandler* delegate (and its subtypes) determines only the parameter list and the type of the return value of possible event-handling methods.
2) In Java, the individual event cannot be referenced as a member of its source separately whereas C# supports it by using *event* key word to define each event as a separate member of its source object.
3) In both Java and C#, when an event occurs, certain additional information can be passed to the corresponding event-handling method. Subtypes of *EventObject* are used in Java to represent such information whereas there are EventArgs and its subtypes as counterparts in C#.
4) In Java, each invocation of one of the *addXXListener()* methods on an event-source can connect a group of related events with their corresponding handling methods implicitly, whereas the "+=" operator of C# connects one single event with its handling method, explicitly.

Based on the comparisons above, the C# manner is more flexible and clearer in terms of expressiveness and it provides us a heuristic to develop a way, in which *dynamic subscription* in event-handling can be modeled uniformly for different platforms. All the essential elements involved in the *dynamic subscription* are:

- the *event-source object*, which are usually the GUI elements of a window or the window itself,
- different types of *event* of an event-source,
- *event-handling methods*, which are implemented in event-handler classes, and
- a *connection operator* that allows to set up the connection of an event to its event-handling method.

As solution to the problems mentioned above we suggest extending the OCL by a new expression, which is labeled by the keyword *event*. In such an *OCL-event-expression* the new registration operator "∼" is used to establish the connection between an event on the left hand side and an event-handling method on the right hand side of this operator. We defined that the new OCL-event-expression in the above form has the type *OclVoid* and consequently no value.

Listing 2 shows the concrete syntax of the OCL-event-expression. Its abstract syntax is shown in Figure 9. An instance of *OCLEventExp* associates with two instances of the abstract syntax type *OCLFeatureCallExp* representing the event and its handling method respectively. The extended OCL abstract syntax will be discussed in next section more detailed.

```
1 <EventExpCS> ::= 'event' ':'
2 <OCLFeatureCallExpCS>'~'<OCLFeatureCallExpCS>
```

Listing 2. Grammar rule deriving OCL–event–expression

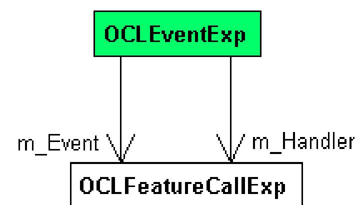


Figure 9. The abstract syntax of the OCL-event-expression

The class diagram in Figure 10 models the C# implementation in Figure 1. Compared with the Java PSM in Figure 8, this C# PSM is much simpler. Because the association ends to GUI elements have been modeled as normal properties, most the cumbersome "lines" could be removed. The connections between the events and event-handling methods are modeled in a tight and well understandable manner using our new suggested OCL-event-expressions.

For example, to specify the method *control_Click()* as the handling method for the *Click* event of both the button *m_Button_1* and the button *m_Button_2* in the derived *Form* class, two expressions

event:
self.m_Button_1.Click∼m_EventHandler.control_Click
event:
self.m_Button_2.Click∼m_EventHandler.control_Click

can be written in the context *EventHandlingComparisonForm*.

As part of our new approach the OCL-event-expression allows to model *dynamic subscription* especially for GUI elements in class diagrams. Figure 10 shows that OCL-event-expressions lead to a very compact C# PSM.

The modified Java PSM is shown in Figure 11. Compared to the Java PSM in Figure 8, this model is both very compact and similar to the C# PSM in Figure 10. The two extended OCL expressions mean that the event-handling methods *button_1_actionPerformed()* and *button_1_mouseClicked()* are connected to their corresponding events of the button

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

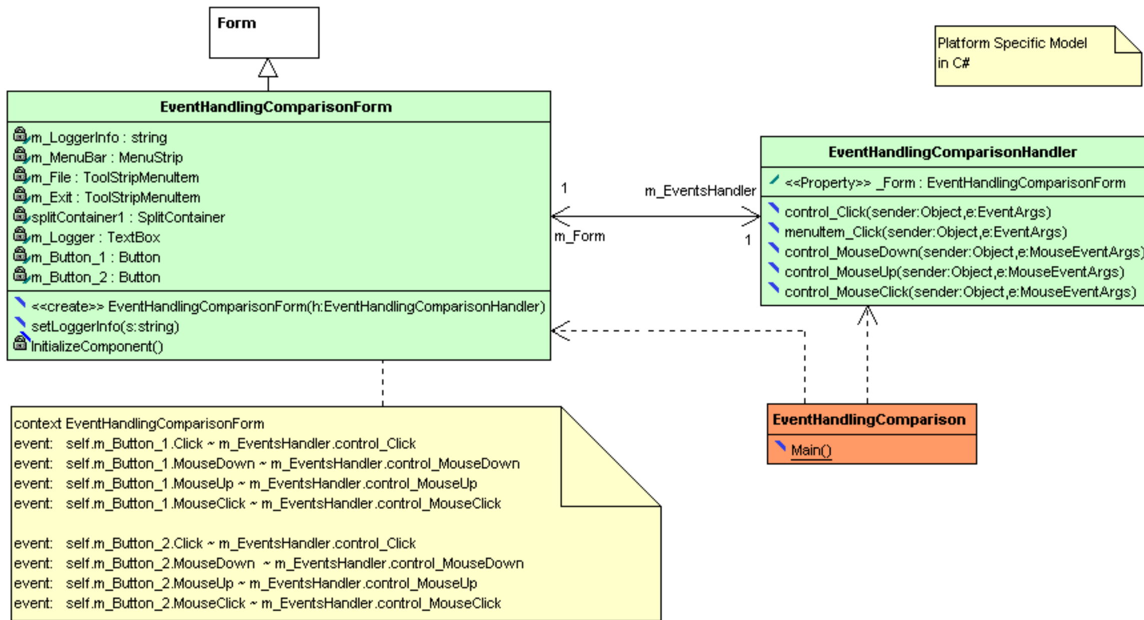


Figure 10. C# PSM using suggested OCL-event-expressions to model event-handler registration

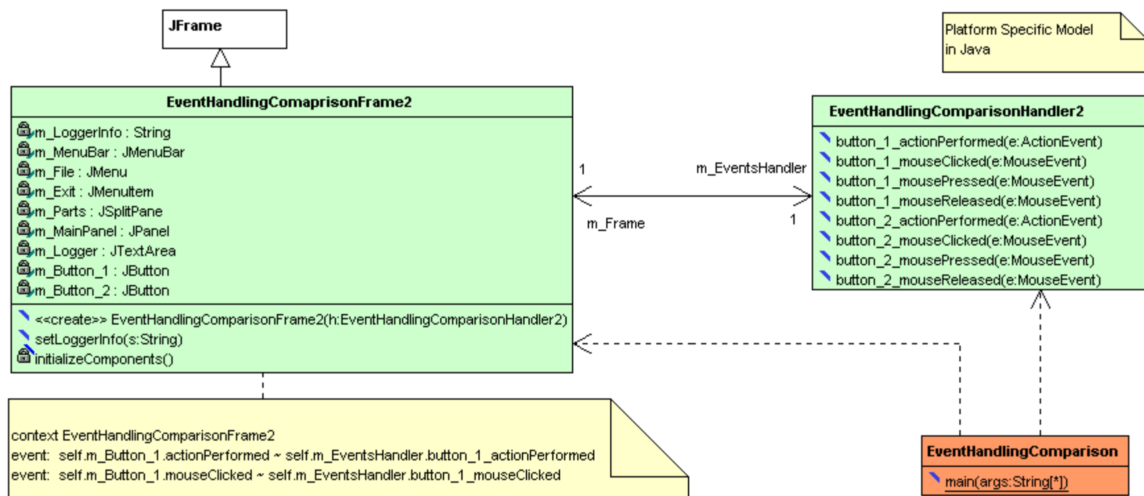


Figure 11. The modified Java PSM with our new approach

m_Button_1. Modeling in this way breaks the constraints in the Java event model in the following way.

- The event-handling methods can be declared as flexibly as in C#. Specifically, their names do not need to be pre-coded. Hence, it is not required any more that the event handler class implements the relevant listener interfaces. This is the solution for point one in the comparison given above.
- As solution of point two, an approach has to be found to identify a single event on an event source. An intuitive candidate may be a Java event object, e.g., *WindowEvent*, *MouseEvent* etc., but they are similar to their corresponding listener interfaces, which group several related events together. It requires an extra effort to select a single event of such a event-collection. As result of our detailed analysis, we found that it is possible to adopt the method name defined in the event listeners to identify a single event. If an event-source can register several event listeners, the method names in this set of listeners classify the events exactly.

Because the Java event-handling framework specifies that event-handling methods must connect to events via methods

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

named *addXXListener()*, only methods registered in this way can be used in the *dynamic publishing* phase of event handling. In order to overcome this restriction of connecting the event-handler with the event, a Java *anonymous class* [12] can be created as a bridge between the fixed method-name of a Java event-handling method and the free chosen name of the event-handler in the UML-model. Both expressions in Figure 11 can be transformed into the Java code as shown in Figure 12 either automatically by a model compiler or manually by Java programmer. For that, it is necessary to choose a correct listener interface or adapter class to declare the anonymous class. This is possible because the event-handling methods used to identify events have been declared or implemented in each listener interface or adapter class clearly. The matching between them is unambiguous.

```
this.m_Button_1.addActionListener(
    new ActionListener()
    {
     public void actionPerformed(ActionEvent e){
         m_EventsHandler.button_1_actionPeformed(e);
     }
    }
 );
this.m_Button_1.addMouseListener(
    new MouseAdapter()
    {
        public void mouseClicked(MouseEvent e){
           m_EventsHandler.button_1_mouseClicked(e);
        }
    }
);
```

Figure 12. Java Code corresponding to the OCL-event-expressions of Figure 11

Although the names of GUI elements, event objects and even the concrete event-handling methods are different, the underlying PSMs are similar to each other both logically and visually. Even more, the reduced PSMs share the same logical structure with the PIM in Figure 2. Hence, we can extend the common GUI elements involved in the MOCCA DPM with common events. Then the OCL-event-expressions can be used the same way as OCL-init-expressions to parameterize GUI elements in order to model the registration of events to their handling methods platform independently.

IV. A Small Set of Additional OCL Expressions – our new Approach to Specify the Behavior of Platform Independent Models

Generally speaking, there are two possibilities to model behaviors platform independently. One of them is to use UML behavioral diagrams, e.g., state chart, activity diagram or sequence diagram. The other one means specifying behaviors in high level action language. Based on our experiences with MOCCA, in some circumstances, the behaviors modeled by, e.g., activity diagrams become more complex than the target code itself. So we suggest *modeling behaviors using an action language*. The actual situation seems a bit awkward, since the OMG has only specified the Action Semantics [5] without one standard surface language. Instead of defining a brand new OMG action-semantics-compliant action language, we find that the widely spread Object Constraint Language (OCL) seems closely to be a good action language due to the following reasons:

- OCL is a standard part of the UML 2 Specification. It has been widely used and proven its value.
- OCL covers large part of the entire Action Semantics of UML. Only the semantics involving changing the states in the model are missing. Such missing semantics can be easily added to the standard OCL with new syntax constructs. We introduce the required small set of additional OCL Expressions in this section.
- The OCL collection types and their predefined operations are powerful in terms of expressiveness and concise in terms of syntax. Together with OCL-body expression, very complex query operations in PIM can be specified both concisely and exactly.

Before we get into the technical issues, an example will be used to show the advantage of specifying complex query operations in OCL. Figure 13 shows a piece of class diagram modeling a payback management system. The query operation *selectPartnersHaveNoPointsInServ()* of the class *Payback* gathers all the program partners, which do not provide services awarding points to the customers. Listing 3 shows the OCL expression specifying the complex logic mentioned above, while Listing 4 shows its corresponding Java implementation. Evidently, the standard OCL expression can be used to specify complex querying operation both exactly and compactly. This is what needed in the modeling phase.

```
1 body:  self.m_Partners->select(p:ProgramPartner |
2            p.m_DeliveredServices -> forAll(s:Service |
3            not s.m_PointsIn ))
```

Listing 3. Specifying complex querying operation in OCL

```
1  ArrayList<ProgramPartner> selectResult=
2        new ArrayList<ProgramPartner>();
3  Iterator<ProgramPartner> itr_0=
4        this.m_Partners.iterator();
5  while(itr_0.hasNext()){
6    ProgramPartner p= itr_0.next();
7    boolean forAllResult= true;
8    Iterator<Service> itr_1= p.getDeliveredServices().
         iterator();
9    while(itr_1.hasNext()){
10     Service s= itr_1.next();
11     forAllResult= forAllResult && !s.isPointsIn();
12   }
13   if(forAllResult)
14     selectResult.add(p);
15 }
16 return selectResult;
```

Listing 4. Java implementation of the OCL expression in Listing 3

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

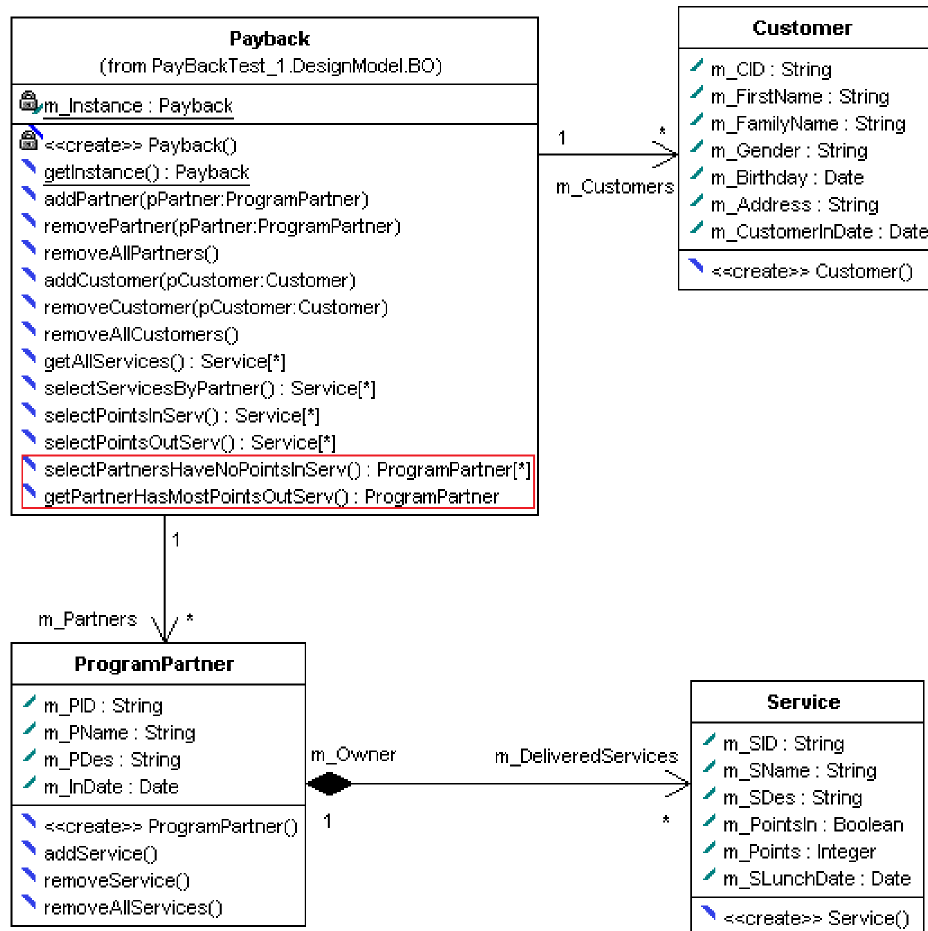


Figure 13. A part of the PIM modeling the business objects of a payback management system

UML supports modeling behaviors in arbitrary action languages inherently. Figure 14 (a) illustrates this support in UML meta-model. Every operation associates with its implementation represented by the abstract meta-class *Behavior*, which can be an activity, an interaction or an opaque behavior concretely. The opaque behavior is a behavior, whose semantics is determined by its body string, while the body can be specified in any kind of action language. Hence the opaque behavior can be used to carry (extended) OCL expressions at the time of modeling behaviors in PIM. This principle has been implemented in our UML 2 Designer. The opaque behavior containing OCL expression is created as a subordinate element of the operation, whose behavior has been specified by this expression.

As yet, the benefits of using standard OCL expression and its concrete usage in a tool have been illustrated. Towards upgrading OCL to a real action language, the next task is to select the important missing action semantics, add them to the standard OCL by defining new syntax constructs and extending the corresponding abstract syntax. We call the extended OCL XOCL, X means executable. Based on the achievement in [2] and the explanation in [5] as well as our research, the following actions, whose semantics are predefined in [5], are involved in the XOCL.

- *AddStructuralFeatureValueAction*,
- *RemoveStructuralFeatureValueAction*, and
- *ClearStructuralFeatureValueAction*:

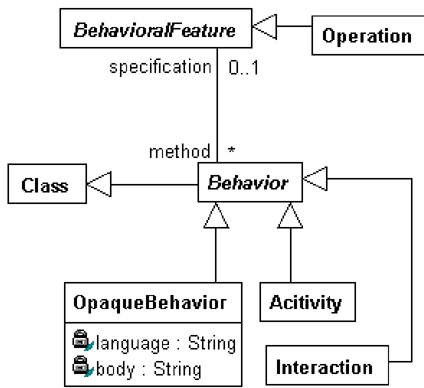
These actions are important to update the state of an object, e.g., to assign the attributes of a class with new values. The syntax construct supporting them is specified by the grammar rule in Listing 5.

```
1 <PropertyAssignExpCS> ::= 'self' '.' ID ':=' <OCLExpCS>
```

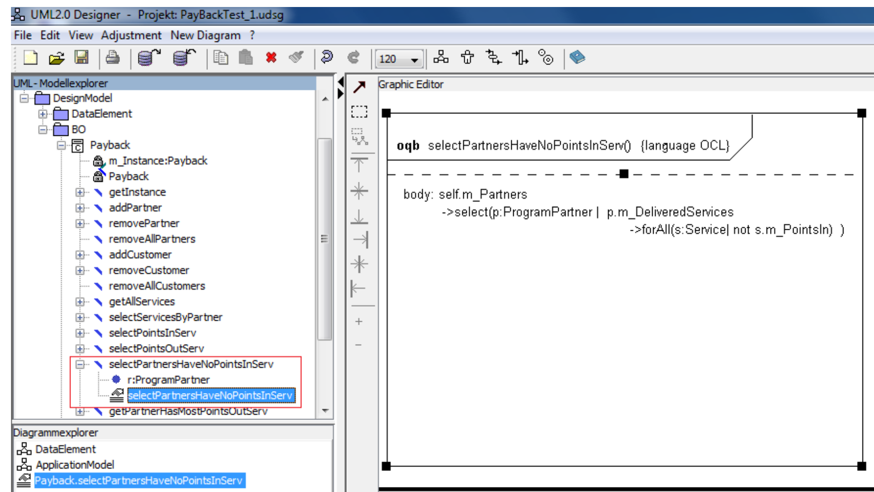
Listing 5. Grammar rule deriving a property assignment expression

The notations of *GOLD Parsing System* [11] are used in this paper, because the GOLD Parsing System is used to implement an XOCL compiler for these extensions. According to GOLD, non-terminals are delimited by the angle

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org



(a) UML meta-model of behavior



(b) Supporting of opaque behavior in our own CASE tool - UML 2 Designer

Figure 14. Behavior as a meta-class in UML and its implementation in UML 2 Designer

brackets and terminals are delimited by single quotes or not delimited at all. As suggested in [4], each non-terminal has one synthesized attribute that holds the instance of the XOCL Abstract Syntax returned by the rule. For this rule, an instance of type *PropertyAssignExp* will be created with all information needed. Figure 15 shows all the types of XOCL Abstract Syntax. This inheritance hierarchy contains also the standard part of OCL Abstract Syntax, which is slightly different with the one specified in [4], in order to make the implementation efficiently.

- *AddVariableValueAction*,
- *RemoveVariableValueAction*, and
- *ClearVariableValueAction*:

These actions are used to assign and update local variables. The syntax construct supporting them is specified by the grammar rule in Listing 6. This rule returns an instance of the type *LocalVarAssignExp*. In fact, this rule can be involved in the rule of Listing 5. We make difference between them in order to keep type checking more efficiently, due to partition of symbol table into two parts.

```
1 <LocalVarAssignExpCS> ::= ID ':=' <OCLExpCS>
```

Listing 6. Grammar rule deriving a local variable assignment expression

- *CreateObjectAction*:

This action is used to create an instance of a class defined in UML model. Listing 7 shows its concrete syntax. An instance of the abstract syntax type *CreateObjectExp* will be created by this rule.

```
1 <CreateObjectExpCS> ::= 'new'
2        <OCLFullNameExpCS> '(' <OCLArgumentsCS> ')'
```

Listing 7. Grammar rule deriving a create object expression

- *DestroyObjectAcion*:

At the time of creating a PIM, the application logic cannot trust in the garbage collection system of the later target platform. So the feature of modeling destructing an object explicitly is provided by this action. Similar to other actions, Listing 8 shows its concrete syntax while *DestroyObjectExp* represents the abstract syntax.

```
1 <DestroyObjectExpCS> ::= 'delete' ID
```

Listing 8. Grammar rule deriving a destroy object expression

- *ReplyAction*:

The most imperative languages support this action with *return* keyword. We use the same keyword in XOCL. An instance of *ReplyExp* will be created to represent this action in abstract syntax.

```
1 <ReplyExpCS> ::= 'return' ID
```

Listing 9. Grammar rule deriving a reply expression

In order to make XOCL a complete action language, three elementary control flows, namely, sequential execution, conditional execution and iterative execution must be supported. In UML Action Semantics [5] they are represented by the meta-class *SequenceNode*, *ConditionalNode* and *LoopNode*. In OCL the conditional execution has been represented by OCL-if-expression; iterative execution has been hidden in the semantics of different *loop* operations, which are not appropriate for all the situations where the iteration semantics are needed; the explicit sequential execution are not supported at all. The extended syntax constructs supporting the elementary flow control semantics are added to the XOCL as follows:

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

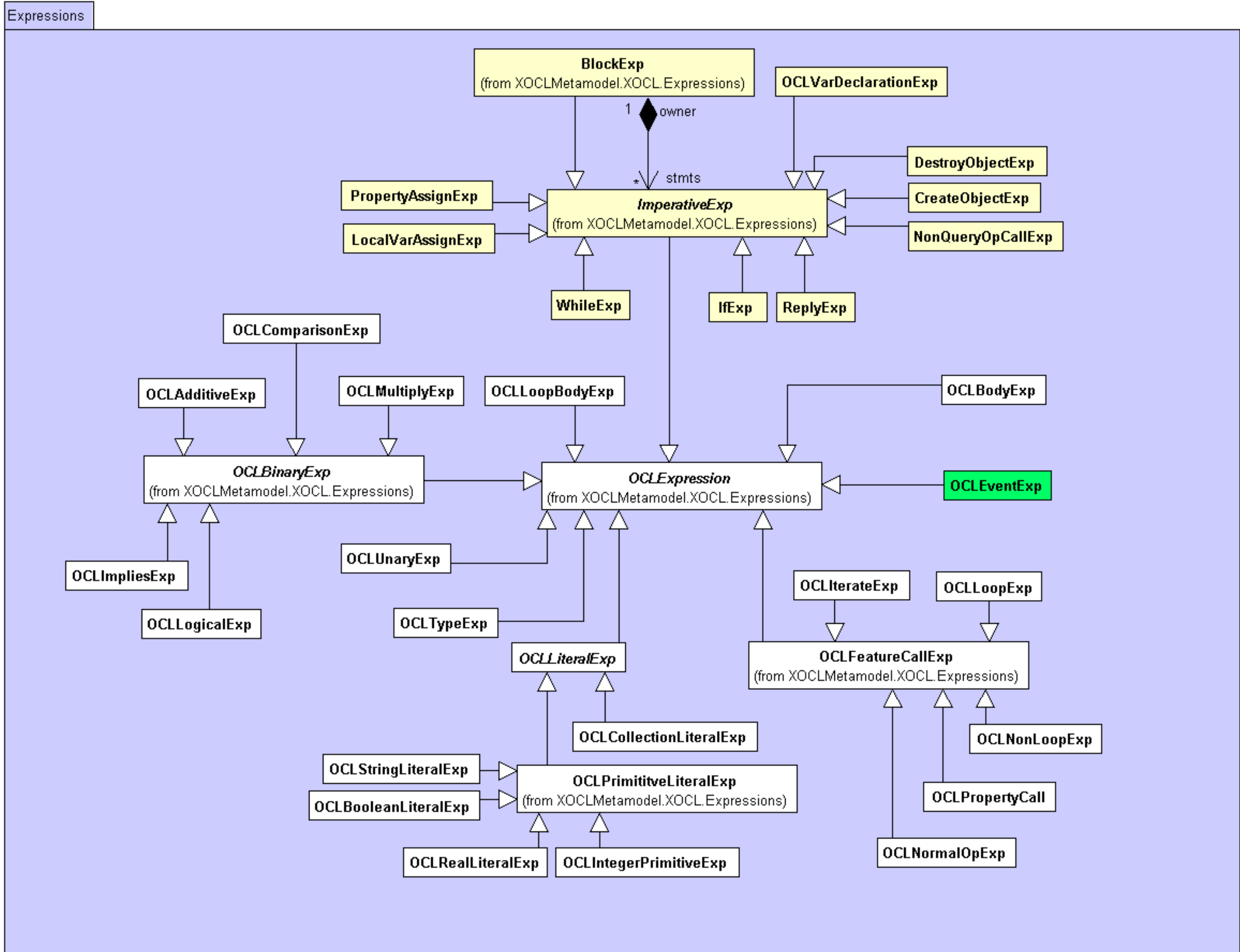


Figure 15. The inheritance relationships of XOCL abstract syntax

```
1 <BlockExpCS> ::=  'begin' <ImperativeExpList> 'end'
2                 |  'begin''end'
3
4 <ImperativeExpList> ::= <ImperativeExpList>
5                            <ImperativeExp>
6                         |  <ImperativeExp>
7
8 <ImperativeExp> ::= <WhileExpCS>
9                   |  <IfExpCS>
10                  |  <OCLVarDeclarationExpCS>      ';'
11                  |  <PropertyAssignExpCS>         ';'
12                  |  <LocalVarAssignExpCS>         ';'
13                  |  <DestroyObjectExpCS>          ';'
14                  |  <ReplyExpCS>                  ';'
15                  |   'call' <OCLFeatureCallExpCS >';'
```

Listing 10. Grammar rules representing XOCL code block

- *Sequential Execution* is represented in XOCL by a block consisting other XOCL-expressions between the *begin* and *end* keywords. It can also be empty. Listing 10 shows all the grammar rules related to derive XOCL-block-expression. The alternatives in the body of the production for non-terminal *ImperativeExp* are the possible expressions, which can be used in an XOCL code block. The upper part of Figure 15 illustrates the corresponding abstract syntax of the concrete syntax in Listing 10. An instance of the abstract syntax type *BlockExp* serves as a container of other XOCL expressions. This can be recognized by the *composition pattern* in the class diagram of Figure 15.

```
1 <IfExpCS> ::= 'if' <OCLExpCS> 'then' <BockExpCS> 'endif
  '
2             | 'if' <OCLExpCS> 'then' <BockExpCS> 'else'
                   <BlockExpCS> 'endif'
3             | 'if' <OCLExpCS> 'then' <OCLExpCS>  'else'
                   <OCLExpCS>   'endif'
```

Listing 11. Grammar rules deriving XOCL conditional execution

- *Conditional Execution* can be specified in XOCL using *if*-expression. Listing 11 shows its syntax. The third

alternative represents the standard OCL-if-expression, the other two allow block expression to be used as body of an XOCL-if-expression. In abstract syntax, *IfExp* represents what is returned by these rules.

- *Iterative Execution* is represented by *while*-expression. Listing 12 shows its concrete syntax. The type *While-Exp* is used in the compilation to represent the abstract syntax of iterative execution.

```
1 <WhileExpCS> ::= 'while' <OCLExpCS> <BlockExpCS> '
      endwhile'
```

Listing 12. Grammar rule deriving XOCL iterative execution

The *CallOperationAction* has been involved in OCL. However, as explained in [3], only the query operations can be called in the normal OCL expressions. In XOCL non-query operations can also be called within an XOCL code block. The last alternative in the body of the production for the non-terminal *ImperativeExp* in Listing 10 makes difference between calling non-query operation in XOCL and calling query operations in original OCL by using a new keyword *call*. The non-query operation *call* is represented in the abstract syntax by the type *NonQueryOpCallExp* that functions as a wrapper of *OCLFeatureCallExp*.

Figure 15 illustrates the important inheritance relationships between the abstract syntax types. There are also important associations between these types as well as between them and the UML meta-classes. Instead of listing all the class diagrams modeling these relationships, an abstract syntax tree (AST) generated by our XOCL compiler after parsing the XOCL expression in Listing 3 will be used to illustrate both the associations between the types involved in the XOCL abstract syntax and the working principle of our XOCL compiler. Figure 16 shows this abstract syntax tree in the form of a *UML Object Diagram*. The both subtrees rooting in a *OCLLoopExp* represent the OCL *select()* and *forAll()* operation. At the runtime both instances of *OCLLoopExp* hold the information to identify the individual OCL loop operation. The *OCLFeatureCallExp* owns generally two branches; the one on the left-hand side is the caller while the other one is the callee. The callee usually uses the type information carried by the caller for type checking and code generation. Each object in yellow color represents a model element defined in the UML model. At the time of constructing an AST, each token, which may represent a model element, is type-checked against the symbol table, which is created based on the underlying UML model. If it is found, the corresponding model element will be associated with the AST node representing the actual token. For example, the AST node representing the token *m_Partners* has been linked with the association end *m_Partners* defined in the class diagram shown in Figure 13.

Since the abstract syntax tree stores all the type information that an XOCL expression involves, code generators for different platforms traverse each node in an AST, generate implementation codes on target platform by consulting the information modeled in the corresponding TPM. The construction of an AST for an XOCL expression happens in the phase of *Model Validation*, while code generation is done in *Model Transformation*. For example, after traversal the AST depicted in Figure 16, a *Java Code Generator* will emit the Java code shown in Listing 4 for the XOCL expression in Listing 3.

To conclude this section, we will use the extended language constructs of XOCL to specify another complex querying operation *getPartnerHasMostPointsOutServ()* of the class *Payback* in Figure 13. This operation gives back the program partner, who provides the most services that award points to their customers. Listing 13 shows the specification of this operation in XOCL. The new features of the XOCL allow software modelers to design their own implementation logics with the help of the existent OCL features on a higher abstraction level. The way of using XOCL is more or less similar to pseudo code used by mathematicians.

```
1  begin
2    resultPartner:ProgramPartner=self.m_Partners.first();
3    numberOfPointsOutServ:Integer=resultPartner.
         m_DeliveredServices->select(s:Service|s.
         m_IsPointsIn = false)->size();
4    index:Integer= 2;
5
6    while index<=self.m_Partners.size()
7    begin
8        num:Integer= self.m_Parnters.at(index).
             m_DeliveredServices->select(s:Service| s.
             m_IsPointsIn = false)->size();
9        if  num > numberOfPointsOutServ then
10       begin
11           numberOfPointsOutServ := num;
12           resultPartner :=  self.m_Parnters.at(index);
13       end
14       endif
15       index := index+1;
16   end
17   endwhile
18   return resultPartner;
19 end
```

Listing 13. XOCL expression specifying an operation

There is one more issue, which must be clarified before ending this section. Originally, the OCL was developed as a declarative language without side effect, which was primarily used in the UML models to specify constraints. After extending OCL into XOCL, the "purely declarative" characteristic is gone. Is this a problem? The answer is no. Firstly, the original OCL was completely contained in our XOCL as a subset. That means the OCL expressions can be used as usual to specify constraints as well as query operations. Secondly, the extended imperative language constructs are only required to specify non-query operations, which are excluded by the OCL. Finally, our language extensions are

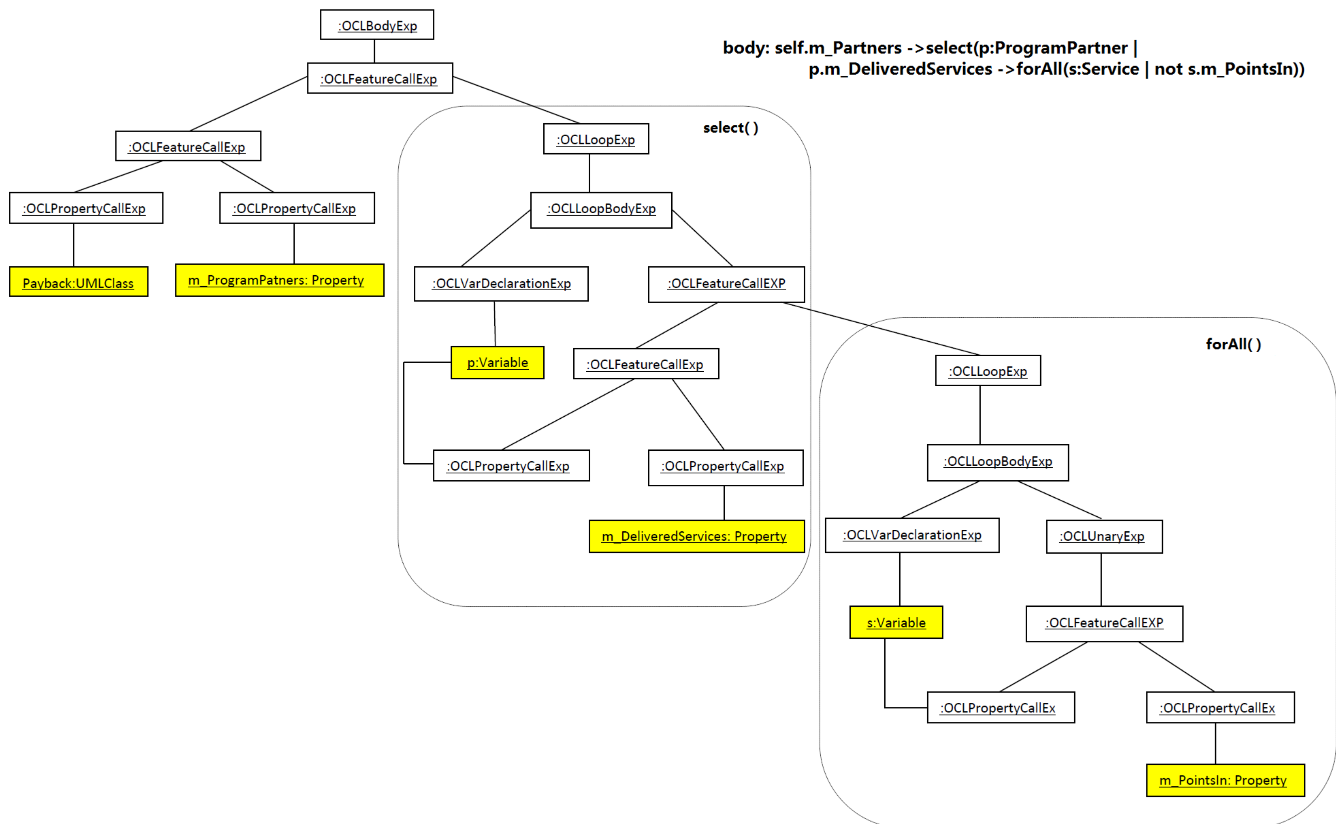


Figure 16. The abstract syntax tree of the XOCL expression in Listing 3

based on the OCL style, making its usage relatively easy.

An alternative to the slightly extended OCL into XOCL will be an extensive action language (AL), which must be defined from the scratch and must repeat many of the powerful data types and operations, which are already defined in OCL. In this unfavorable scenario three languages (UML + OCL + AL) are required to specify a complete model so that one main issue of models namely *simplicity* is not achieved.

V. Example of a Complete PIM

As an important experimental result to show the benefits, the platform independent model (PIM) of the GUI-based application of Figure 1 has been created using all three new approaches discussed in this paper. As Figure 1 illustrates, this application reacts to the events by means of logging them in the text area below the both buttons.

In the PIM created in this section, we concentrate on illustrating principles. Hence, only the *click*-events of the both buttons will be logged. The application will react to the *closing*-event of the main application window and the *click*-event of the menu item called *Exit* by means of disposing the held system resources explicitly. Figure 17 shows the class diagram of our example PIM. After manipulating the GUI elements in Smart GUI Editor, the OCL-init-expressions that parameterize them can be generated automatically and saved in a *constraint* attached to the class *EventHandlingComparisonWindow*. Listing 14 shows these generated OCL-init-expressions.

To register the important GUI-events to their corresponding handling methods, our OCL-event-expressions can be used. Listing 15 shows four examples. The other important events can be registered in the same manner. The abstract event-objects, such as *click*, *closing* have been modeled as properties of the corresponding GUI types in the MOCCA DPM.

The OCL-event-expressions must be used in the same *constraint*, which contains the OCL-init-expressions. After model transformation, all these (extended) OCL-expressions will be transformed into target language code and saved into the body of a special method called *initializeGUIComponents()*, which will be automatically generated and added to the class representing *EventHandlingComparisonWindow* on the target platform.

To model behaviors, or in other words, to specify the implementation of methods the XOCL-expressions are used as the body of an opaque behavior, which belongs to the method to be specified. To specify the implementation logic of the handling method for the event *Clos-*

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

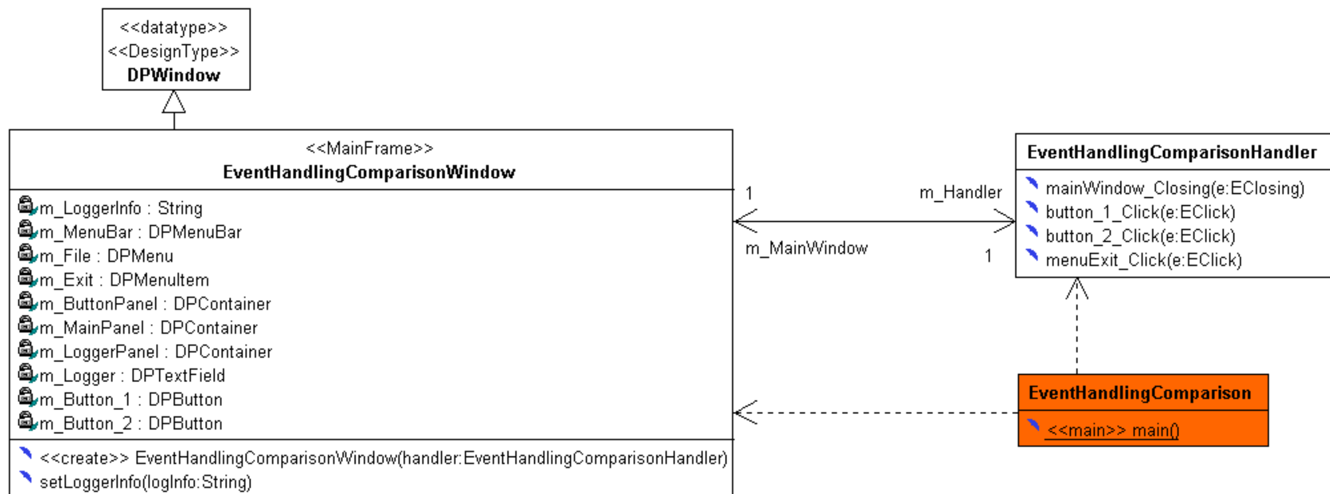


Figure 17. Platform independent model of the application of Figure 1

ing of the class *EventHandlingComparisonWindow*, XOCL-destroy-expression in Listing 16 is a good choice.

In model transformation, the semantics of destroying an application window will be translated into the target language constructs, e.g., *dispose()* together with *System.exit(0)* in Java. Listing 17 shows the same logic as Listing 16, because if the menu item *exit* is clicked, the entire application will be finished.

Listing 18 specifies the event handling method button_1_Click(). The platform independent semantics of the condition in *if*-expression can be understood as a test against null pointer in Java.

Listing 19 specifies the similar event handling method button_2_Click().

The XOCL-property-assignment-expressions in Listing 20 initialize the event handler as well as the logger string in the constructor of the class *EventHandlingComparisonWindow*. The identifier *handler* in Listing 20 is the formal parameter of the constructor.

In model transformation they are translated into target language constructs with the same semantics and additionally, the generated *initializeGUIComponents()* method will be appended in the transformed constructor to initialize the GUI elements as well as to register GUI events to their handling methods.

In Listing 21 the XOCL-create-expressions have been used to construct the necessary instances. Instead of using *main* method directly, we call such a unique method *start-up* method. Because no matter what it is called in PIM, the start-up method, which is identified by the stereotype *main* will be always transformed into the corresponding language construct on target platform.

As a review to the content discussed in Section IV, it should be understood that all the XOCL-expressions will be parsed into the abstract syntax trees like what is represented in Figure 16 in the phase *model validation*. These XOCL ASTs will be processed by consulting the corresponding TPM in the phase *model translation* to generate language construct on the target platform.

```
1  init:  self.posX = 100 and self.posY=100 and
2         self.length = 500 and self.height = 800 and
3         self.title = 'Main Window'
4
5  init:  self.m_MenuBar.owner = self
6
7  init:  self.m_File.text = 'File' and
8         self.m_File.owner = self.m_MenuBar
9
10 init:  self.m_Exit.text = 'Exit' and
11        self.m_Exit.owner = self.m_File
12
13 init:  self.m_MainPanel.split = true and
14        self.m_MainPanel.horizontal = false and
15        self.m_MainPanel.owner = self
16
17 init:  self.m_ButtonPanel.split = false and
18        self.m_ButtonPanel.owner = self.m_MainPanel
19
20 init:  self.m_Button_1.posX = 35   and
21        self.m_Button_1.posY = 15   and
22        self.m_Button_1.length = 93 and
23        self.m_Button_1.height = 31 and
24        self.m_Button_1.text= 'Button One' and
25        self.m_Button_1.owner = self.m_ButtonPanel
26
27 init:  self.m_Button_2.posX = 285  and
28        self.m_Button_2.posY = 15   and
29        self.m_Button_2.length = 93 and
30        self.m_Button_2.height = 31 and
31        self.m_Button_2.text= 'Button Two' and
32        self.m_Button_2.owner = self.m_ButtonPanel
33
34 init:  self.m_LoggerPanel.split = false  and
35        self.m_LoggerPanel.owner = self.m_MainPanel
36
37 init:  self.m_Logger.multiLines = true and
38        self.m_Logger.owner = self.m_LoggerPanel
```

Listing 14. OCL–init–expressions parameterizing GUI elements in Class EventHandlingComparisonWindow

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
1 event: self.closing ~ self.m_Handler.mainWindow_Closing
2 event: self.m_Exit.click ~ self.m_Handler.
      menuExit_Click
3 event: self.m_Button_1.click ~ self.m_Handler.
      button_1_Click
4 event: self.m_Button_2.click ~ self.m_Handler.
      button_2_Click
```

Listing 15. OCL–event–expressions registering events of the GUI elements in Class EventHandlingComparisonWindow to their handling methods

```
1 begin
2   delete self.m_MainWindow;
3 end
```

Listing 16. XOCL–expression specifying the event handling method mainWindow_Closing()

```
1 begin
2   delete self.m_MainWindow;
3 end
```

Listing 17. XOCL–expression specifying the event handling method menuExit_Click()

```
1 begin
2   if not self.m_MainWindow.oclIsUndefined() then
3   begin
4      call self.m_MainWindow.setLoggerInfo('Button 1 was
              clicked!');
5   end
6   endif
7 end
```

Listing 18. XOCL–expression specifying the event handling method button_1_Click()

```
1 begin
2   if not self.m_MainWindow.oclIsUndefined() then
3   begin
4     call self.m_MainWindow.setLoggerInfo('Button 2 was
            clicked!');
5   end
6   endif
7 end
```

Listing 19. XOCL–expression specifying the event handling method button_2_Click()

```
1 begin
2   self.m_Handler := handler;
3   self.m_LoggerInfo :='Logging the user actions...';
4 end
```

Listing 20. XOCL–expressions specifying the constructor of the class EventHandlingComparisonWindow

```
1 begin
2   handler: EventHandlingComparisonHandler = new
         EventHandlingComparisonHandler();
3   window: EventHandlingComparisonWindow  = new
         EventHandlingComparisonWindow(handler);
4   handler.m_MainWindow := window;
5 end
```

Listing 21. XOCL–expressions specifying the start–up method of the entire application

VI. Conclusion

The UML is a powerful language for preparing models of object oriented software. Such models can be used as documentation of existing software for their maintenance, and as source of new software to develop. In order to create precise models the OCL is used to specify the UML meta-model, and can be used for design models with the UML commonly.

The aim of the MDA technology is the generation of program code for a certain platform based on a complete UML/OCL model. In order to gain benefit from this innovative technology, it is necessary to create the basic platform independent model (PIM) concisely, uniformly, completely, and especially with low effort. Preparing such UML/OCL - PIMs we recognized three serious problems. In this paper we emphasize these problems and suggest efficient solutions, which base on UML and OCL.

1) How it is possible to describe both structural compositions and visual parameters of GUI elements?
   We suggest to use a normal UML-class of a window, to model the GUI elements as their attributes, and to specify the parameter values in OCL-init-expressions attached to their class-context. These OCL-init-expressions can be generated from a Smart GUI Editor based on the visual information.
2) How it is possible to model the connection between an event source and an event handler in a platform independent manner?
   We suggest a simple OCL extension. This single new OCL-event-expression allows to model the registration of handling methods to event sources in a more compact, well understandable, and uniform manner independent from the implementation platform. This new approach simplifies the class diagrams strongly without loss completeness. The application of our new OCL-event-expression leads to strong benefit in models of many GUI elements.
3) How it is possible to model the behavior in PIM both concisely, exactly, and compactly?
   We suggest a restricted extension of the OCL. The expressive, declarative OCL is upgraded into an imperative action language, which we called XOCL. With XOCL, complex query operations can be specified as usual while non-query operation with complex control flows can also be specified using the extended language constructs.

Using our three new approaches it is possible to create platform independent models efficiently, concisely, uniformly and completely.

References

[1] D. Liang and B. Steinbach, *A new General Approach to Model Event Handling*, ICSEA 2010, pp.14-19, 2010 Fifth

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

International Conference on Software Engineering Advances, 2010.

[2] K. Jiang, L. Zhang, and S. Miyake, *Using OCL in Executable UML*, MoDELS 2007.

[3] J. Warmer and A. Kleppe, *The Object Constraint Language, Getting Your Models Ready For MDA*, Addison-Wesley & Pearson Education, Boston, MA, USA, 2003.

[4] OMG: Object Constraint Language Specification 2.0, 2006. http://www.omg.org/spec/OCL/2.0/

[5] OMG: UML 2.4 Superstructure Specification, 2011. http://www.omg.org/spec/UML/2.4/Superstructure/Beta2/PDF/

[6] J. Warmer, A. Kleppe, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley, 2003.

[7] OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, 2008. http://www.omg.org/spec/QVT/1.0/PDF/

[8] S. Nolte, *QVT Operational Mappings*, Springer, 2009.

[9] D. Fröhlich, *Object-Oriented Development for Reconfiguralbe Architectures*, Dissertation of Dr. Fröhlich, TU Freiberg, Germany, available July 2011. http://www.qucosa.de/fileadmin/data/qucosa/documents/2209 /InformatikFrXXhlichDominik80246.pdf

[10] B. Steinbach, D. Fröhlich, and T. Beierlein, *Hardware/Software Codesign of Reconfigurable Architectures Using UML*, UML for SOC Design, chapter 5, Springer, 2005.

[11] GOLD Parsing System Online Documentation, available July 2011. http://www.devincook.com/goldparser/index.htm

[12] C.S. Horstmann and G. Cornell, *Core Java*, 8th ed. Prentice Hall, 2008.

[13] A. Troelsen, *Pro C# 2008 and the .Net 3.5 Platform*, 4th ed. Apress, 2007.

[14] H. Keller and S. Krüger, *ABAP Objects*, 2nd ed. Galileo Press, 2007.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org