

A new General Approach to Model Event Handling

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—It is a complex task to model event-handling in class diagrams. In this paper, we suggest a new approach, in which the Object Constraint Language is extended by the ability to register event handlers for an event source. This approach helps to create real platform independent models for the Model Driven Architecture technology. The benefit of our new approach is that event-handling can be modeled in class diagrams concisely and uniformly.

Keywords—event handling, OCL extension, 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) [1] has aroused 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 generate one or more Platform Specific Models (PSM) from the PIM. The PSMs have involved detailed information for implementation. Hence, code generation from a PSM is straightforward. As proof of the concept, an MDA-tool called MOCCA (Model Compiler for reConfigurable Architecture) [2][3] 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. Based on our experiences with MOCCA, modeling event handling for GUI-based application in PIM using standard UML is time-consuming, because almost all programming languages have their own GUI libraries and the underlying event models.

This paper is structured as follows. At first, general problems about modeling of event handling in class diagrams will be addressed. To emphasize these problems, models based on Java and C# are used. Secondly, an abstract publishing-subscription model will be discussed in detail. This model based on the event models of the most popular programming languages. Thirdly, the Java and C# event models will be decomposed based on the publishing-subscription model, in order to find out both the common aspects and the differences between them. Finally, a new approach based

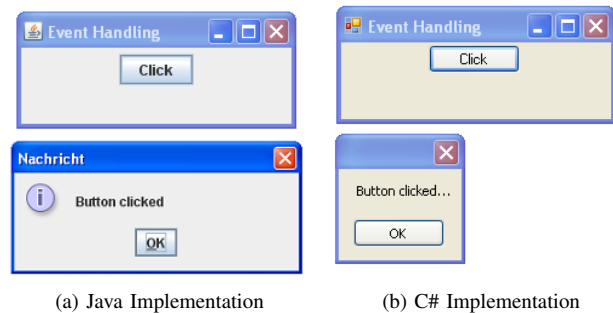


Figure 1: Example 1 – A simple application implemented in Java and C#

on the Object Constraint Language (OCL) [5][6] will be introduced. This approach allows to describe event-handling in class diagrams both concisely and platform independently.

II. THE PROBLEM OF MODELING EVENT HANDLING IN CLASS DIAGRAMS

At first view, it seems to be possible to use the OCL *isSent* (denoted \wedge) operator to model the coupling of events to their handling methods. However, the *isSent* operator can only be used in the post condition [5] of an event sending method, e.g., *fireActionPerformed()* operation of the class *JButton* in Java. It is unnecessary 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.

A very simple GUI-application, which has been implemented in both Java and C#, is shown in Figure 1. 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 2 shows both models. 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#

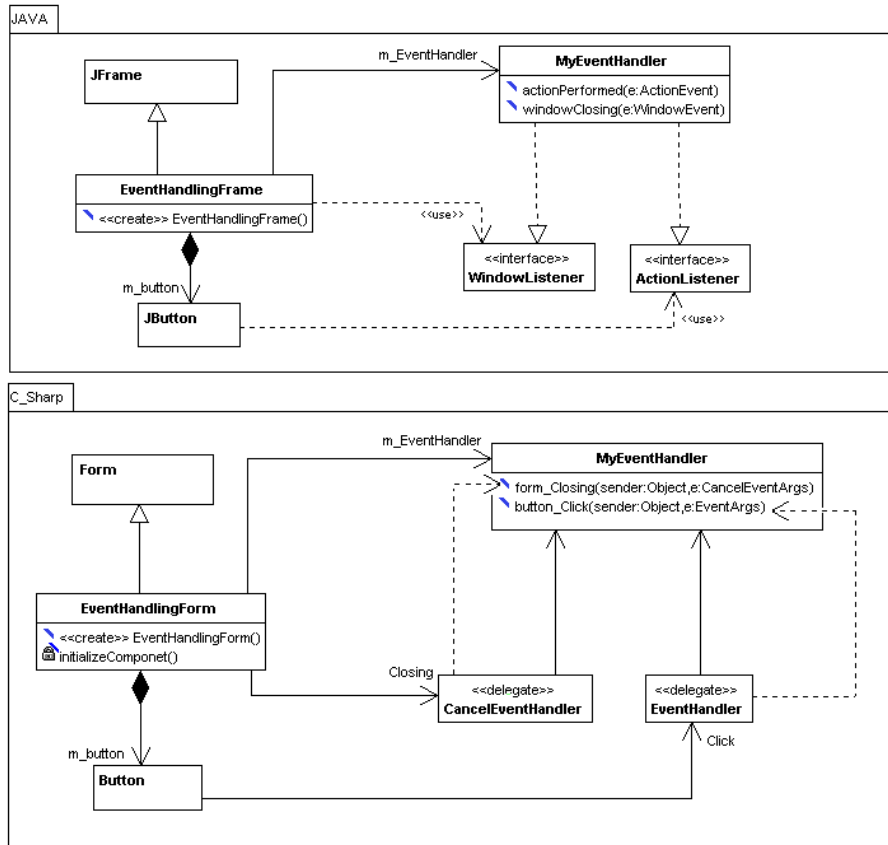
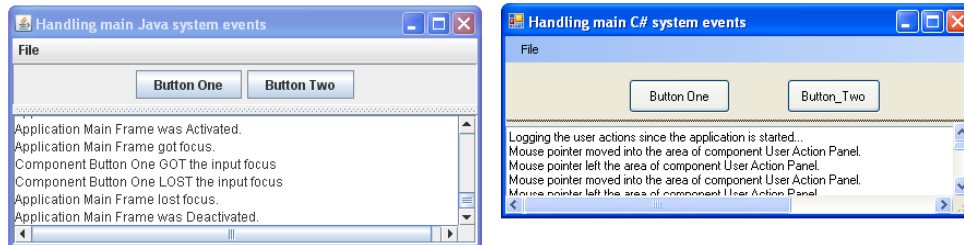


Figure 2: Java and C# models for the application of Figure 1



(a) Java Implementation

(b) C# Implementation

Figure 3: Example 2 – A more comprehensive application implemented in Java and 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 [8] while in C# delegates are used, which are in fact type safe callbacks based on function pointer [9]. 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 semantic 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. Figure 3 shows another more comprehensive application, which is also implemented in Java and C#. Its Java PSM is shown in Figure 4. Due to space limitation, the corresponding C# PSM with similar complexity is not shown 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.

III. OCL EXTENSION – OUR NEW APPROACH TO MODEL EVENT HANDLING

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 to 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 [4]:

- *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 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 [7].
- *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. The analysis in Section II shows, 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 paper 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 *EventArgs* 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.

The class diagram in Figure 5 models the C# implementation in Figure 3. Compared with the Java PSM in Figure 4, 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

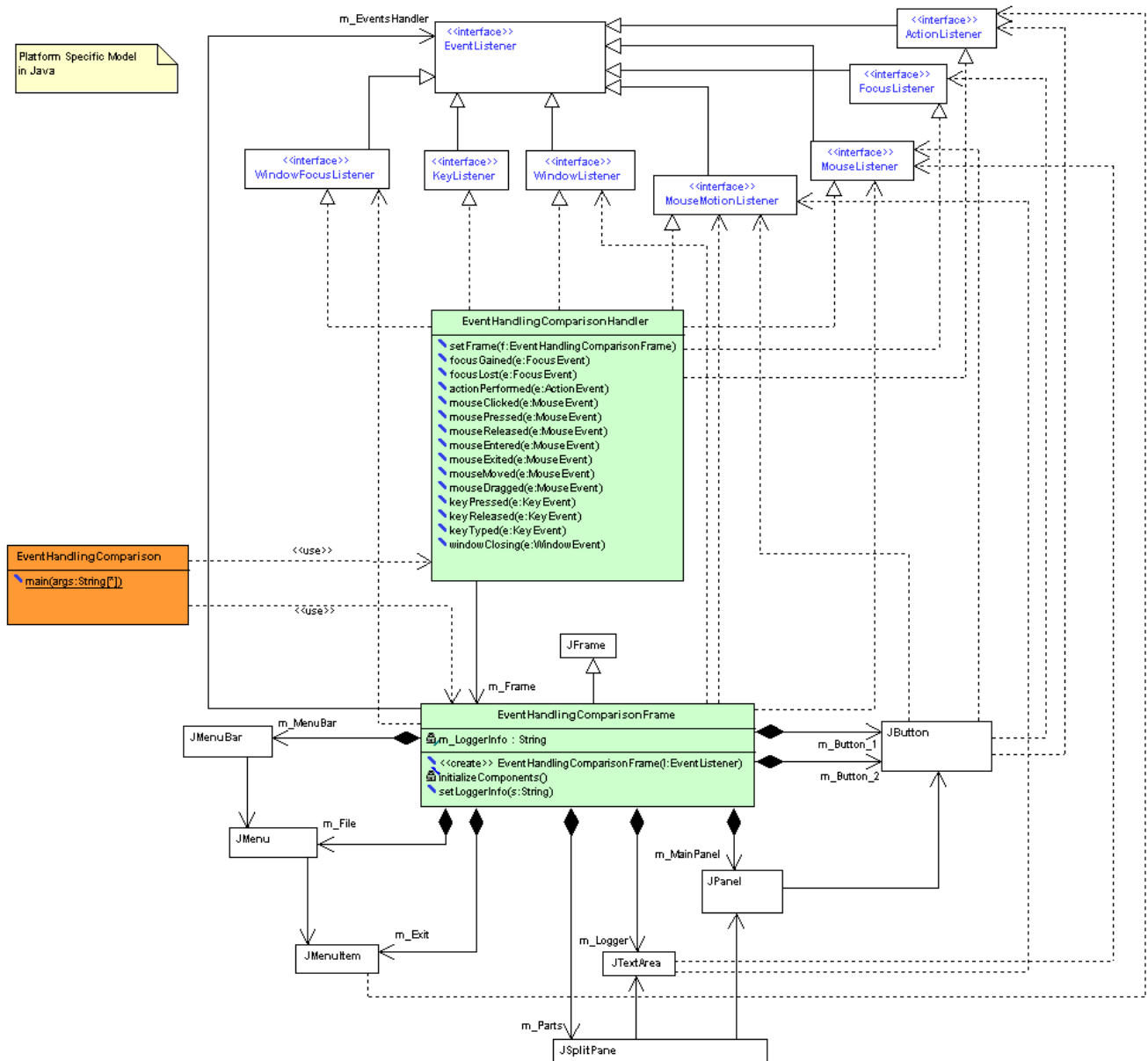


Figure 4: The PSM for the Java implementation of Figure 3

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 buttons *m_Button_1* and *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 5 shows that OCL-event-expressions lead to a very compact C# PSM.

The modified Java PSM is shown in Figure 6. Compared to the Java PSM in Figure 4, this model is both very compact and similar to the C# PSM in Figure 5. 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 *m_Button_1*. Modeling in this way breaks the constraints in the Java event model in the following way.

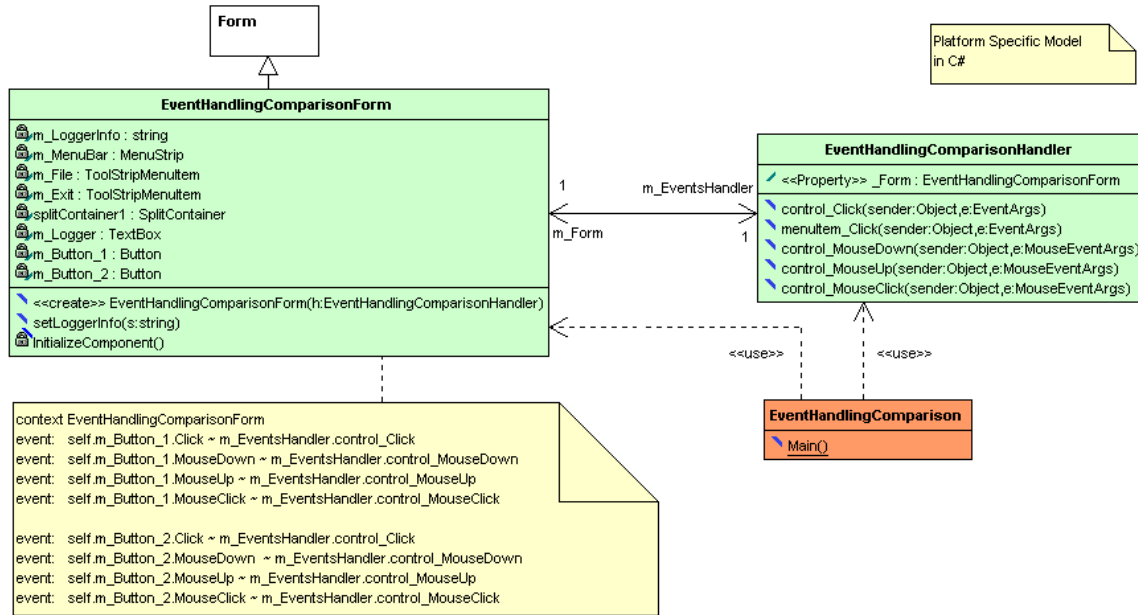


Figure 5: C# PSM using suggested OCL-event-expressions to model event-handler registration

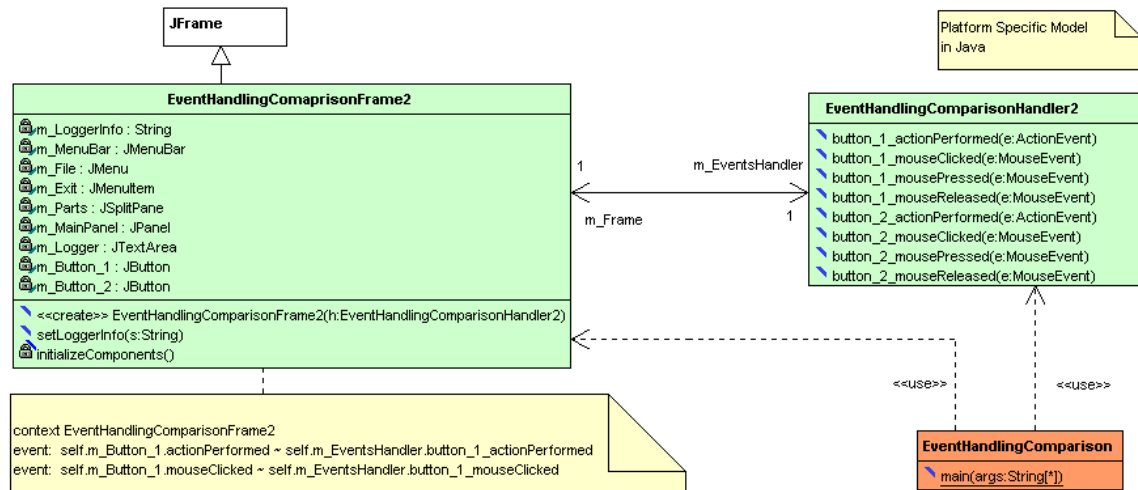


Figure 6: The modified Java PSM with our new approach

- 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 `addXXListener()` methods, 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* [8]

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 6 can be transformed into the Java code as shown in Figure 7 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. 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. Hence, we can abstract a platform independent GUI tool-kit with the most common GUI elements and their important events for the general usage to develop a PIM for an application. A Design Platform Model (DPM) [2][3] which contains the most basic design types for primitive data types, IO facilities etc., has been used for MOCCA to establish PIMs. Now the MOCCA and its DPM is being upgraded to support our new approach.

```

this.m_Button_1.addActionListener(
    new ActionListener()
    {
        public void actionPerformed(ActionEvent e){
            m_EventsHandler.button_1_actionPerformed(e);
        }
    }
);
this.m_Button_1.addMouseListener(
    new MouseAdapter()
    {
        public void mouseClicked(MouseEvent e){
            m_EventsHandler.button_1_mouseClicked(e);
        }
    }
);

```

Figure 7: Java Code corresponding to the OCL-event-expressions of Figure 6

IV. CONCLUSION

In this paper, we introduced a simple OCL extension. The suggested OCL-event-expressions allow to model the registration of event-handler especially for GUI elements in a more compact, well understandable, and uniform way for different implementation platforms. Using this new approach, 39 "lines" and 15 GUI library types involved in Figure 4 could be removed from this UML-model of a simple application. The result of this simplification without lost of information is the concise UML-model of Figure 6. Hence, the suggested OCL-event-expressions allow a strong improvement for establishing PIMs for GUI-based applications in MDA process. The further advantages are summarized as follows:

- 1) The syntax of the suggested OCL-event-expressions is straightforward and its semantics is clear. Hence, an existing OCL compiler can be easily modified to translate such additional expressions into target codes.
- 2) All the aspects, which are important to connect an event-handler to an event, are involved in the single suggested OCL-event-expression. Hence, this new approach can be used not only for the explored Java and C# languages but also to model event-handling for other platforms, i.e., ABAP Objects [4] or Qt [10].

Although the approach presented is primarily designed for modeling the event-handling of the common GUI elements, it is also valuable to deal with the general events defined by the application designer. A possible solution is, for static publishing, certain application events can be declared as members of a developing class (event-sender). For dynamic publishing, certain event-sending methods can be modeled using the OCL *isSent* operator in the context of the event-sender. The event-handling methods can be modeled in a developing class (event-handler) as normal. This is the part of static subscription. For the remaining dynamic subscription, the presented approach can be used as described in this paper to connect application-events to their event-handling methods.

REFERENCES

- [1] J. Warmer, A. Kleppe, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley, 2003.
- [2] D. Fröhlich, *Object-Oriented Development for Reconfigurable Architectures*, Dissertation of Dr. Fröhlich, TU Freiberg, Germany, available May 2010. URL: <http://fridolin.tu-freiberg.de/archiv/pdf/InformatikFrXXhlichDominik80246.pdf>
- [3] B. Steinbach, D. Fröhlich, and T. Beierlein, *Hardware/Software Codesign of Reconfigurable Architectures Using UML*, UML for SOC Design, chapter 5, Springer, 2005.
- [4] H. Keller and S. Krüger, *ABAP Objects*, 2nd ed. Galileo Press, 2007.
- [5] J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 2003.
- [6] OMG: Object Constraint Language Specification 2.0, 2006, <http://www.omg.org/spec/OCL/2.0/>
- [7] K. Jiang, L. Zhang and, S. Miyake, *Using OCL in Executable UML*, MoDELS 2007
- [8] C.S. Horstmann and G. Cornell, *Core Java*, 8th ed. Prentice Hall, 2008.
- [9] A. Troelsen, *Pro C# 2008 and the .Net 3.5 Platform*, 4th ed. Apress, 2007.
- [10] Qt Online Documentation, available May 2010: <http://qt.nokia.com/products>