Including Functional Usability Features in a Model-Driven Development Method

Jose Ignacio Panach[1], Natalia Juristo[2], and Oscar Pastor[3]

[1]Escola Tècnica Superior d'Enginyeria, Departament d'Informàtica, Universitat de València
Avenida de la Universidad, s/n, 46100 Burjassot, Valencia, Spain
joigpana@uv.es
[2] Universidad Politécnica de Madrid, Campus de Montegancedo, 28660 Boadilla del Monte, Madrid, Spain
natalia@fi.upm.es
[3]Centro de Investigación en Métodos de Producción de Software, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain
opastor@pros.upv.es

Abstract. The Software Engineering (SE) community has historically focused on working with models to represent functionality and persistence, pushing interaction modelling into the background, which has been covered by the Human Computer Interaction (HCI) community. Recently, adequately modelling interaction, and specifically usability, is being considered as a key factor for success in user acceptance, making the integration of the SE and HCI communities more necessary. If we focus on the Model-Driven Development (MDD) paradigm, we notice that there is a lack of proposals to deal with usability features from the very first steps of software development process. In general, usability features are manually implemented once the code has been generated from models. This contradicts the MDD paradigm, which claims that all the analysts' effort must be focused on building models, and the code generation is relegated to model to code transformations. Moreover, usability features related to functionality may involve important changes in the system architecture if they are not considered from the early steps. We state that these usability features related to functionality can be represented abstractly in a conceptual model, and their implementation can be carried out automatically.

1. Introduction

Nowadays, there is an ever-increasing need to improve the quality of computer systems in order to be able to compete commercially in the computer market. For this reason, many members of the Human-Computer Interaction (HCI) community have focused their efforts on improving the quality characteristics defined in the ISO/IEC 9126-1 [16]. There are two types

of recommendations according to HCI. The first type is composed of presentation issues with slight modifications of the UI design (e.g., buttons, colours, fonts, layout). The second type of recommendations are strongly related to the system functionality and may involve changes in the system architecture if they are not considered from the early steps of the software development process [2,12]. These features are called functional usability features [17] and these are the target usability features of our research. An example of functional usability feature is the Undo action, which allows the user to go back.

The Software Engineering (SE) community has been working several years on the Model-Driven Development (MDD) paradigm [22], which states that the analysts' entire effort should be focused on a conceptual model and the system should be implemented by means of model to code transformations. In MDD, a conceptual model is used to represent a system independently of platform and technology. This conceptual model is the input for a model compiler which includes transformation rules to generate the code according to the target platform. Even though existing MDD methods such as WebML [7] or UWE [19] are very powerful in building conceptual models, they do not support specific conceptual primitives to represent usability features in them. Usability features are usually included manually once the code has been generated, decreasing the analyst's efficiency, who must model the system and; later on, must implement the usability features in the code. This contradicts the ideas that claim to include functional usability features from the early steps [2, 12], and manual changes in the system architecture may result in inconsistencies between the model and the code.

To mellow these problems, we propose dealing with functional usability features within a conceptual modelling perspective. Our approach aims to represent functional usability features abstractly in any MDD method by means of conceptual primitives. By conceptual primitive, we mean modelling elements that have the capability of abstractly representing a feature of the system. Examples of conceptual primitives in an Object Model are classes of a class diagram, attributes and services of classes, etc. Conceptual primitives are gathered in a model, which aims to represent a view of the system. For example, we can use an Object Model to represent the persistency and a Task Model to represent the interaction. All the models together compose a conceptual model, which is a way of viewing domains specifically [24].

We aim to define conceptual primitives to allow functional usability features to be precisely modelled so that usable systems can be generated from a conceptual model. The main advantages of our proposal are:

- Usability can be abstractly represented in a precise notation by means of conceptual primitives.
- Usability features can be automatically or semi-automatically implemented together with the business logic by means of a model compiler. Moreover, the model compiler designs the system hidden from the analyst, which improves the analyst's efficiency [15, 28].
- The usability represented in a conceptual model is reusable. The same conceptual model can generate a system for different

platforms (Web, Desktop, PDA, etc.) depending on the existing model compiler capability.

The proposal to include usability in an MDD method is described in an abstract way so that it can be applied to any MDD-based method. However, in order to exemplify our proposal, we have selected the OO-Method [26], which has been successfully applied in an industrial tool called INTEGRANOVA [6].

The paper is structured as follows. Section 2 explains the functional usability features used in the paper. Section 3 describes our approach to include usability modelling in an MDD method. Section 4 shows a practical application of this approach to the OO-Method. Section 5 presents a metamodel to represent functional usability features abstractly. Section 6 reviews the literature. Finally, section 7 presents the conclusions.

2. Background: Functional Usability Features

Many authors have identified a set of functional features with strong impact on usability such as, Comstock [8], Lauesen [20], Perzel [27] or Tidwell [30], among others. From all the existing works, we have used the features called *Functional Usability Features (FUF)* [17]. This choice is based on the following reasons: (1) FUFs are defined with *usability requirements guidelines*, which are composed of questions that analysts ask the users in order to extract usability requirements. These guidelines are useful to identify which conceptual primitives are needed to represent each usability mechanism. (2) FUFs definition includes *usability design patterns*, which describe how to include the functionality of the FUFs in the architecture and how to implement them. These patterns are very useful to specify the changes in the model compiler to support the code generation. (3) These FUFs have been evaluated in experiments, such a way we can ensure that the effort to include these features is high since there is dependence with the architecture [18].

FUFs are defined as recommendations for improving system usability that have an impact on the architectural design. These FUFs have been derived from usability heuristics, rules, and principles. As Folmer [12] and Bass [2] state, FUFs must be included in the system architecture together with the business logic of the system as another functional requirement. Each FUF is divided into several *usability mechanisms*, which are different subtypes of the FUF. In other words, each FUF has a main goal that can be specialized into more detailed goals called usability mechanisms. Next, we present a summary of the FUFs, but more details can be viewed in [14]:

- **Feedback**: This keeps the user informed at all times. The usability mechanisms are: System Status; Progress Feedback; Warning.
- **Wizard**: This helps the users to carry out tasks that require several steps of user interaction. The only usability mechanism is Step by Step.

- **User Input Error Prevention**: This helps to prevent the users from making data input errors. The only usability mechanism is Structured Text Entry.
- **User Profile**: This lets the users adapt the system to their preferences. The only usability mechanism is Favourites.
- **Cancel**: This lets the users go back at least one step. The usability mechanisms are: Global Undo; Abort Operation.
- **Help**: This provides different help levels for different users. The only usability mechanism is Multilevel Help.

Juristo proposes including these FUFs throughout the entire software development process, from requirements capture to implementation. In the first step (requirements capture), usability requirements guidelines help the analysts to extract the user requirements related to FUFs. These guidelines are composed of questions that the analyst asks to the end user. There is a guideline for each usability mechanism. For example, some questions of the guideline to capture requirements of the mechanisms called Structured Text Entry (from User Input Error Prevention) are: *Where is input from the user required, and in which format? How to guide the user to introduce such input in the required format?* , *If the chosen option allows the user to choose from a list, discuss with the user whether or not such list has a fixed number of items.* In next steps, usability patterns are used by the analysts to build the analysis and design models from the captured requirements. Finally, in the last step, the analyst implements the system with these models.

3. A Method for Including Usability Features in MDD Environments

The main target of this paper is the definition of a method for including usability features in any MDD software development process. We have called this method, *Method to Incorporate Functional Usability Features into MDD (MIFUM)*. We have used usability mechanisms defined by Juristo to identify the required conceptual primitives inside the MDD method to represent them in an abstract way. Note that our approach is not exclusive for FUFs. We have used FUFs because they present a set of advantages with regard to other existing features (as we have mentioned in section 2), but any guideline to capture usability features related to functionality can be used in the same way as we use the guidelines of FUFs.

MIFUM is divided into two stages; in the first stage we identify the properties to model and in the second stage we propose changes in the MDD method to include these properties. The first stage is defined in two steps: (1) Definition of modes of use; (2) Identification of properties. These steps are performed by a usability expert who knows how to work with usability guidelines. The second stage is composed of two steps: (3) Definition of changes in the conceptual model; (4) Description of changes in the model compiler. These steps are performed by the MDD designer, who must

enhance the MDD method to support usability features. Next, we detail these four steps that compose the method MIFUM.

Definition of Modes of Use.
A usability mechanism can be applied to the system in different ways to fulfil its goal. We have called each one of these ways ***Mode of Use (MoU)***. Therefore, each MoU has a specific target to be reached within the general goal of the usability mechanism. Targets of different MoUs that belong to the same usability mechanism attempt to reach the same overall goal and do not contradict each other. We have extracted modes of use from requirements guidelines defined by Juristo [17]. As stated in section 2, these guidelines have a questionnaire that the analysts must fill out with the user. We have used these questions [14] to detect MoUs in each usability mechanism.

We take the usability mechanism called *Structured Text Entry* as an example to illustrate the definition of MoUs from the requirements capture guidelines. This mechanism aims to help the user to insert data in a specific format. For example, a Date, a Boolean value, or an Enumerated value. We have identified that this goal can be reached through three modes of use: (1) *Specify the widget type to enter data with a specific format* (checkbox, radiobutton, etc.); (2) *Mask definition* that specifies the required format of an input text; (3) *Default values* in order to help the user to enter information. The first mode of use has been derived from the question of the requirements guideline, *what is the required format for the input data?* Both the second and the third mode of use have been derived from the question, *how should the user be guided to introduce data with a required format?* It is important to note that even though the last questions are the same, the goal of each mode of use is different. The second one aims to define a mask while the third one aims to define default values.

Identification of Properties.
The requirements guidelines also include questions to capture usability requirements related to configuration options. We have denoted the different configuration possibilities that a MoU has to adapt itself to usability requirements as ***properties***. For example, the MoU *Specify the widget type to enter data with a specific format* has a property to specify the widget type (called *Type of input widget*). Possible values for this property are all the widget types supported by the development method (checkbox, radiobutton, etc). This property is derived from the question of the requirements guideline, *which is the format for data entry?* As in the definition of MoUs, the same question of the guideline can derive several properties, each of which has the goal of configuring a different option of the MoU.

In most cases, analysts must adapt these properties to a specific system. In other cases, properties can be configured automatically by means of the model compiler without any intervention by the analysts. Therefore, there are two types of properties: configurable and non-configurable.

- **Configurable properties:** This type is composed of properties that require an analyst to make decisions about how to configure them. The analyst

must configure these properties according to user's preferences. For instance, the MoU called *Mask definition* has two configurable properties: one property named *Input field selection,* to select the widget with the mask; and another property named *Regular expression,* to define the mask through a regular expression. Both of them depend on user's decisions.

- **Non-configurable properties**: This type is composed of properties that do not have any alternative in the requirement guidelines or most guidelines agree on the same configuration for all cases. These properties are not configured by the analyst because their configuration must be the same in all developed systems. In an MDD method, the model compiler is responsible for including non-configurable properties in all generated systems, assuring that the same configuration is used in all of them. Note importantly that the use of non-configurable properties restricts the analyst's decisions, since these properties are configured in a hidden way from the analyst. However, we improve the efficiency of developing the system, since the analyst has to work with less conceptual primitives. It is important to get a good balance between configurable and non-configurable properties, classifying as non-configurable only those properties whose values are supported by several usability guidelines and they are not critical for the system. For instance, there is a non-configurable property called *Undoable Elements* that represents the number of undoable actions in the usability mechanism *Undo cancel*. Usability guidelines determine that the stack of undo actions should contain about twelve elements to be considered as usable [30].

In our research work, we have already applied the two steps that compose the first stage: Definition of MoUs and Identification of properties. A detailed explanation of the 15 MoUs with all their Properties that resulted from the run of the process can be viewed in [23]. The outcomes of this stage can be applied to any MDD method.

Definition of Conceptual Primitives.

In this step, the conceptual model of the MDD method chosen to include functional usability features is enriched with new conceptual primitives that represent configurable properties. Of the two types of properties, only configurable properties imply changes in the conceptual model. Since there are as many conceptual models as MDD methods, the definition of new concrete conceptual primitives depends on a concrete conceptual model and, consequently, on a concrete MDD method.

The current step consists of verifying whether or not there are already conceptual primitives that represent each configurable property. If there is no conceptual primitive to represent these properties, or some configuration possibilities cannot be represented, we have to enrich the conceptual model with new conceptual primitives that ensure the required expressiveness. The steps for enriching the conceptual model are the following:

1. To identify which model of all the models that compose the conceptual model should include the configurable property. This choice depends on

the functionality of the property. For instance, if the property is related to visualization options, the modification must be applied to the model that represents the system interface.

2. To identify conceptual primitives that must be included in the model. Each configurable property needs at least one conceptual primitive to represent it abstractly. In this step, we have to include the conceptual primitives that represent all the configuration possibilities in the model selected previously.

Changes Required in the Model Compiler.

The last step for including functional usability features inside an MDD method consists of describing the changes that must be applied to the model compiler. This step, such as the step where new conceptual primitives are defined, depends on the MDD method chosen because the model compiler is specific to that method. Changes for the model compiler derive from:

- **New conceptual primitives that represent configurable properties:** The model compiler must have the capability to recognize new conceptual primitives and generate the code that implements them. To do this, the resulting software must be able to represent any valid configuration alternative that could be specified with configurable properties.
- **Non-configurable properties:** Although these properties do not involve changes in the conceptual model, they concern the model compiler. The model compiler must include the functionality of non-configurable properties automatically in the generated code.

Both of these changes involve including new attributes, services and classes in the generated code. We have used the architectural usability patterns of Juristo [18] as the basis to propose the changes in the model compiler. To describe the changes in the model compiler, we use a graphical notation with UML class diagrams. Each usability mechanism is represented by means of a class diagram, which is used to represent new software classes, new attributes and new methods that must include the functionality of MoUs in the generated code. The reason why we have used class diagrams is that they are abstract enough to be understandable independently of the transformation language used by the model compiler since there are many languages to specify transformation rules (e.g. Xpand [32]). The changes in the generated code specified with a class diagram are valid for any transformation language. Each MDD designer must modify the model compiler to support MoUs according to its language.

4. A Lab Experiment: The OO-Method

This section presents a practical application of MIFUM to a specific MDD method widely used in the industry: the OO-Method [6]. We want to emphasize that what we apply specifically to the OO-Method could also be instantiated to any other MDD-based method by adapting the abstract

concepts to the specific features of the conceptual model being considered. We have selected the OO-Method based on two characteristics. First, the OO-Method is supported by an industrial tool called INTEGRANOVA which allows us to explain how the analyst can work with FUFs once they have been included. Second, the conceptual model of OO-Method is abstract enough to include new conceptual primitives that express all FUF properties. The OO-Method conceptual model is composed of four complementary views:

- **The Object Model:** This specifies the system structure in terms of classes of objects (with attributes and services) and their relations.
- **The Dynamic Model:** This represents the valid sequences of events for objects.
- **The Functional Model:** This specifies how events change object states.
- **The Interaction Model:** This models the interaction between the system and the user by means of two views: the Abstract Interaction Model and the Concrete Interaction Model [1]. *The Abstract Interaction Model* represents the interface independently of the types of interaction and the peculiarities of the platform. The *Concrete Interaction Model* specifies the interface representation in terms of elements that can be perceived by the end-user.

As an example of how to include FUFs in the OO-Method, we selected *Feedback*, whose goal is to provide feedback to the user. We selected one of the usability mechanisms of this FUF: *System Status Feedback*. This mechanism aims to provide feedback to the user about the system all the time. We have selected this usability mechanism because it contains several MoUs that are not yet supported by the OO-Method and its goal is very simple, which facilitates the didactic task.

The system selected to demonstrate the effects of including this usability mechanism is a (necessarily simple) system for managing a car rental business. The users of this system are employees that are distributed throughout several offices. Next we explain the two steps of the first stage: Definition of modes of use and Identification of properties.

4.1. Modes of Use of System Status Feedback

This usability mechanism has the functionality of informing about important changes in the system or when an error occurs. We have derived the following MoUs:

- **MoU_SSF1: Inform about the success or failure of an execution:** This MoU has the functionality of informing whether or not a service has been executed successfully. This MoU is extracted from the question of the usability guideline [17, 14]: *Does the user want to be provided with notification of system failures?* This MoU ensures that the user is aware of the system state after a service execution. For instance, in the rent-a-car system, after each service is executed, the system should inform about its success or its failure.

- **MoU_SSF2: Show the state of the stored information:** This MoU aims to show information about the system state that is useful for the end-user before triggering a service. The question of the guideline used to define this MoU is [17, 14]: *Will the system have the capability to report system status?* This question has the goal of capturing usability requirements to show the system state not only after the service execution, but also before. For instance, in the rent-a-car system, it is essential to show the number and model of available cars before executing the service *rent car*.
- **MoU_SSF3: Show the state of visible actions:** Depending on the system status, some visible actions can or cannot be executed. Actions that cannot be executed will show an error message when the user triggers them. In order to avoid errors of this type, this MoU states that actions which cannot be executed should be disabled. The question of the requirements guidelines used to define this MoU is [17, 14]: *Will the system have the capability to report system status?* Even though the origin of this MoU and MoU_SSF2 is the same question, the goal of each one is different. In MoU_SSF2, the goal is to show the system status as information, while in this MoU the goal is to prevent errors. For instance, in the rent-a-car system, the button for printing an invoice should only be available when the client has returned the car or has finished the rental.

4.2. Properties of System Status Feedback

This section shows the properties that are grouped by modes of use. The first mode of use, ***Inform about the success or failure of an execution (MoU_SSF1)***, has two configurable properties:

- **Service selection:** This property specifies which services will show the success or failure of its execution. This property is non-configurable because, according to the ergonomic criteria *Immediate feedback* of Bastien and Scapin [3], all the services should inform about success or failure. This property is derived from the question of the guideline: *Which failures does the user want to be notified about?*
- **Message visualization:** The feedback can be represented in different ways: using icons or textual messages, in an emergent window or in the main window. Analysts must take the best option to satisfy user's requirements; therefore, this property is configurable. The property is derived from the following questions of the guideline: *Which information will have to be displayed obtrusively because it is related to a critical situation? Which information will have to be highlighted because it is related to an important situation? Which information will be displayed in the status area?* All these questions have the goal of capturing how the users want to visualize the information about the system status.

The second mode of use, ***Show the state of the stored information (MoU_SSF2)***, has three configurable properties:

- **Static information:** This property defines the information about the system state that will be shown statically in all the interactions. It is derived from the question of the guideline: *Which information will be shown to the user?*
- **Dynamic information:** This property specifies which information could be extracted from the system database dynamically in order to show the state of the system. The analysts should be capable of defining formulas based on stored information in order to extract this type of information. The question of the guideline that is used to derive this property is the same as the question used in the previous property. Both properties together specify the content of the information that shows the system state.
- **Message visualization:** This property is used to specify how the system state will be shown to the user. The questions of the guideline used to derive this property are: *Which information will have to be displayed obtrusively because it is related to a critical situation? Which information will have to be highlighted because it is related to an important situation? Which information will simply be displayed in the status area?*

The third mode of use ***Show the state of visible actions (MoU_SSF3)*** has two configurable properties:

- **Action selection:** This property has the goal of selecting the actions that should be disabled depending on the system state. The question of the guideline used to derive this property is: *Which information will have to be disabled?*
- **Condition to disable:** This property is used to define the predicate that must be satisfied to disable the actions according to the system state. This property is also derived from the same question as the previous property.

Next, we explain how the outcomes of the first stage drive the changes in the MDD method (INTEGRANOVA) throughout the steps of the second stage: Definition of conceptual primitives and Changes required in the model compiler. Both steps must be performed by the INTEGRANOVA designer.

4.3. New Conceptual Primitives for System Status Feedback

Each configurable property implies changes in the OO-Method conceptual model and thus, in the industrial tool that supports the OO-Method: INTEGRANOVA. The INTEGRANOVA designer must include new conceptual primitives that represent every configurable property. Analysts that work with INTEGRANOVA will deal with FUFs in the developing systems by means of those conceptual primitives. In the following, we present these changes.

Inform about the success or failure of an execution (MoU_SSF1):

This mode of use has two properties: *Service selection* and *Message visualization.* Only the second one affects the conceptual model since the first one is non-configurable. Currently, generated systems with INTEGRANOVA

inform when a service execution fails but the analyst cannot specify how the message is displayed. Table 1 explains why *Message visualization* is not completely supported by INTEGRANOVA. In order to support the property *Message visualization* completely, we must change the following two OO-Method models:

- **Concrete Interaction Model:** The analysts must use new conceptual primitives for modelling the visual feature in these two circumstances: (1) messages when the execution has finished successfully; (2) messages when the execution has failed. For both circumstances, the analysts must choose among these display options:

Table 1. Unsupported property of MoU_SSF1 in INTEGRANOVA

Property	Elements of the property that are not supported
Message visualization	-INTEGRANOVA does not allow specifying how the messages are displayed to the user. -INTEGRANOVA allows defining error messages belonging to constraints or preconditions related to a service, but they cannot define error messages caused by triggering the execution of a service in an invalid state of the object (a transition between states that is not defined in the Dynamic Model). In this case, the error message is a generic textual message.

  - Showing the message textually: The possible values are:
    - Within a new window:
      1. Obtrusively (modal)
      2. Type of message: warning, error or alert
      3. Text font
      4. Size
      5. Colour
      6. Alignment
    - Within the main window
      1. Position in the window
      2. Text font
      3. Size
      4. Colour
      5. Alignment
  - Showing the message graphically: The possible values are:
    - An icon to show, for instance, when the execution finishes successfully ✓ and when the execution fails ✗
    - Where to show the icon within the main window
  - Not showing any information about the execution state

Each one of these alternatives is represented by means of a conceptual primitive. In order to facilitate the analysts' work, these conceptual primitives should have a default value in case analysts do not want to configure them.

Default values should be the values that are the most frequently used. Analysts can change these default values to adapt the conceptual primitives to the user's requirements. By default, the value of these conceptual primitives is to notify the successful execution graphically with the icon ✔ in the right corner within the status bar of the main window. Default values for error messages are to show the message textually in a modal window of error type with Arial font, size 10, black colour, and centred alignment.

Figure 1 shows a prototype of the Concrete Model of INTEGRANOVA to model the format of the success message with the new conceptual primitives. In that window, the analyst can decide the different visualization possibilities for the service *Create a car*. This figure shows the default options.

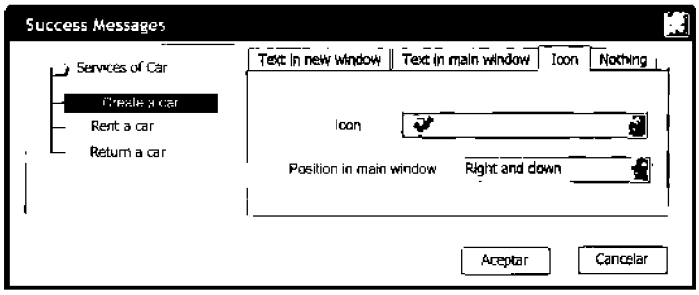

Fig. 1. Example of modelling MoU_SSF1 in the Concrete Interaction Model

- **Object Model:** This model is enriched with two conceptual primitives that are used to define textual messages.
  - Error message by an invalid transition between states: This error occurs when a user triggers an action within a state of the object where this action cannot be executed. If the analyst does not define any message, the model compiler will include a textual generic message. Since texts of the other error messages (constraint and precondition violation) can be currently defined, they do not require new conceptual primitives.
  - Success message: For each service of the Object Model, the analyst can decide the text that will be shown to the user when the execution has been a success. If the analyst does not define any message, the model compiler will include a generic message for all the services.

We propose including a new window in INTEGRANOVA where analysts can insert either of these text messages. Figure 2 shows the two components that must be included in the Object Model of INTEGRANOVA to model the success and failure messages (only for an invalid transition between two states of the object).

Show the state of the stored information (MoU_SSF2):

The second mode of use, *Show the state of the stored information (MoU_SSF2),* can be applied to the definition of navigation buttons with dynamic information in their aliases. Currently, OO-Method can only specify

static aliases for the navigations, therefore, OO-Method only supports the property *Static information.* Using dynamic information in navigation buttons, the user can query information of the target context without performing the navigation. For example, in the rent-a-car system, the user can navigate from the list of customers to the list of orders and invoices for a specific customer.

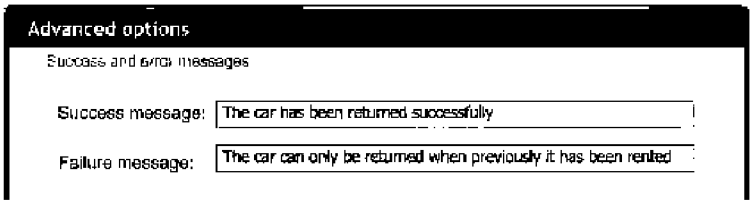

Fig. 2. Example of modelling MoU_SSF1 in the Object Model

Applying dynamic aliases to these buttons, the system can display the number of orders, the number of unresolved invoices, and the number of paid invoices for a selected customer without performing the navigation. This dynamic information is displayed in the label of the navigation buttons. Figure 3 shows an example of buttons that support dynamic information.

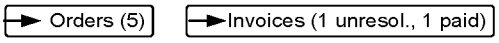

Fig. 3. Dynamic alias in navigation buttons with visualization format by default

Table 2 specifies which configurable properties are not supported currently in INTEGRANOVA: *Dynamic information* and *Message visualization*. The property *Static Information* is already supported by aliases of navigation buttons defined in the Abstract Interaction Model.

Table 2. Unsupported properties of MoU_SSF2 in INTEGRANOVA

Properties	Elements of the property that are not supported
Dynamic information	The analyst cannot specify aliases on navigation buttons that depend on stored information.
Message visualization	The analyst cannot define how the aliases will be displayed to the user.

In order to support the properties *Dynamic information* and *Message visualization* and solve the problems specified in Table 2, we must change the following OO-Method models:

- **Abstract Interaction Model:** This model should include a conceptual primitive to specify the formula that represents the property *Dynamic information.* The formula could be built using class attributes, standard functions, user functions, and arithmetic operators.

Figure 4 shows how the analyst can model the *Dynamic information* in the Abstract Interaction Model in INTEGRANOVA. This is a part of the window where the analyst defines navigations. The field *Static Alias* currently exists,

but the field *Dynamic Alias* has been added as a new element. The bulb is a wizard that defines the formula for the dynamic alias. The glass is a zoom that defines a formula with many sentences.

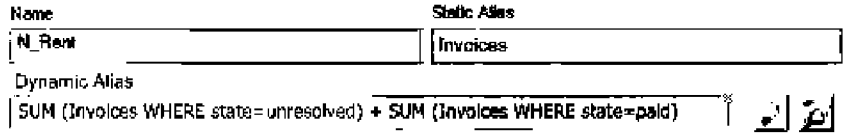

Fig. 4. Example of modelling MoU_SSF2 in the Abstract Interaction Model

- **Concrete Interaction Model:** This model must be modified to support the property *Message visualization.* New conceptual primitives must represent the different possibilities for visual features: text format of the navigation button; text size of the navigation button; alignment of the text; colour of the text; icon of the navigation button; size of the navigation button. By default, the text format of the label will be Arial, size 12, align centred, black colour, and the button size will be the size necessary to include the alias.

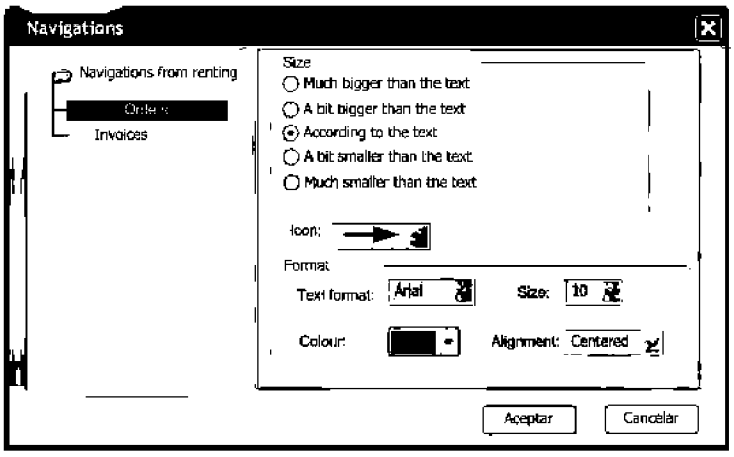

Fig. 5. Example of modelling MoU_SSF2 in the Concrete Interaction Model

Figure 5 shows how to model the format of the navigations in the Concrete Interaction Model of INTEGRANOVA. The window includes several primitives to model all the possibilities of the property *Message visualization*.

Show the state of visible actions (MoU_SSF3):

The third mode of use *Show the state of visible actions (MoU_SSF3)* is not yet supported by the OO-Method. The application of this mode of use is useful in two actions that can be triggered by the user: service execution and navigation to other contexts.

- **Service execution**: The buttons that execute a service should be disabled if the service cannot be triggered due to a precondition or a state of the object that is not valid.

- **Navigation to other contexts:** Navigation buttons can be disabled depending on a condition specified by the analyst (e.g. when the target context is empty).

Table 3. Unsupported properties of MoU_SSF3 in INTEGRANOVA

Properties	Elements of the property that are not supported
Action selection	INTEGRANOVA cannot specify which services and navigations must be disabled when a condition is satisfied.
Condition to disable	INTEGRANOVA allows defining conditions that must be satisfied to execute a service; therefore this property already exists for services. However, the analyst cannot relate a condition to a navigation.

Both properties of *Show the state of visible actions* (*Action selection* and *Condition to disable*) are not supported by INTEGRANOVA currently (Table 3). In order to solve the problems specified in Table 3 and provide a complete support to MoU_SSF3, we must change the following OO-Method models:

- **Object Model:** This model must include a conceptual primitive per service to specify when the service must be disabled. This primitive allows modelling the property *Action selection* for a service.
- **Abstract Interaction Model:** The definition of navigations in this model must be enhanced with two new primitives. First, we need a conceptual primitive per navigation to specify whether or not the navigation can be disabled. This primitive allows modelling the property *Action selection* for a navigation. Second, we need a conceptual primitive also per navigation to specify the condition when the navigation will be disabled. This primitive aims to support the property *Condition to disable* for navigations.

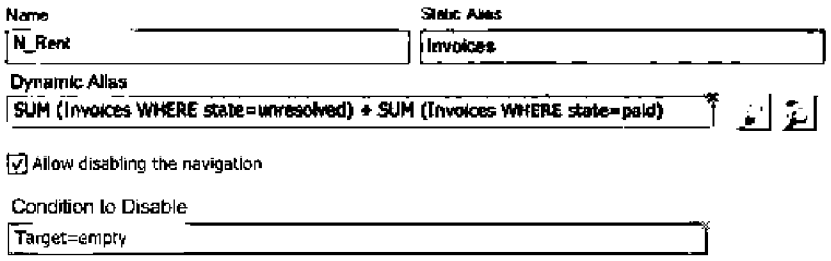

Fig. 6. Example of modelling MoU_SSF3 in INTEGRANOVA

Figure 4 shows a portion of the window that models the navigation in the Abstract Interaction Model. In this example, the analyst has specified that the navigation defined in Figure 4 will be disabled when the target context is empty.

All new conceptual primitives specified in this section must be included in INTEGRANOVA by the designers. Once they have been included,

INTEGRANOVA will allow the analysts to work with FUFs by means of interfaces like the prototypes used in this section.

4.4. Changes Required in the Model Compiler by System Status Feedback

Both configurable properties and non-configurable properties require changes in the model compiler. The designer of the MDD method must specify the changes in the model compiler needed to generate the code that implements the FUFs. Generated code in the OO-Method has a client/server architecture, and changes affect both the client and the server. Client classes represent interaction issues, which can be divided into three types of Interaction Units (IU) in the OO-Method: (1) an IU called *Instance* that shows a specific object; (2) a IU called *Population* that shows a list of objects that are instances of the same class; (3) a IU called *Service* for entering data required to execute a service. Server classes implement the business logic and carry out service executions. To understand the practical implications of the conceptual updates that have been proposed, we give a brief description of the classes affected by the changes that we have introduced:

- *Class X action* implements the business logic of the system in the server. There is one of these classes for each class defined in the Object Model. The letter X represents all the classes defined in the Object Model: for instance, in the rent-a-car system, the class *Car*, which is represented abstractly in the Object Model, generates a class in C# called *Class car action*, which implements the business logic of the car.
- *FrmGnInstance* implements a context that shows the information of an object instance.
- *FrmGnPopulation* implements a context that shows the list of object instances.
- *Form X* implements a context used to insert data and trigger a service that needs this data. This class has *OK* and *Cancel* buttons.
- *Service wrapper* is used to connect client classes with server classes.
- *Alert manager* controls how the information about the system state will be shown to the user.
- *Navigation* implements a navigation inside *FrmGnInstance* and *FrmGnPopulation* from one context to another.

The class diagram in Figure 7 shows the architecture to represent all the modes of use of *System Status Feedback*. New software classes that need to implement a usability mechanism appear with a grey background; classes with some methods that have been modified to add the usability feature appear with a background crossed by diagonal lines; finally, those classes that do not change appear with a white background. The only class included from scratch is *Alert manager*. The classes *OK* and *Cancel* should not be modified. The rest of the classes must add new methods or attributes:

- *FrmGnInstance* and *FrmGnPopulation* should add methods to request disabling actions and navigations in order to implement the properties *Action selection* and *Condition to disable* of MoU_SSF3.
- *Service wrapper* should add a method called *Invoke_disable_actions* to trigger *Class X action* in order to verify actions and navigations to disable. This change implements the properties *Action selection* and *Condition to disable* of MoU_SSF3.
- *Form X* should add methods to trigger the methods of *Alert manager* that inform the user about the success or failure of the execution by means of a message. It should also add an attribute to save the visualization options of that message. These changes implement the properties *Service Selection* and *Message visualization* of MoU_SSF1.
- *Class X action* should add a method to check which actions must be disabled. This change implements the properties *Action selection* and *Condition to disable* of MoU_SSF3
- *Navigation* should add attributes to represent the alias of the navigation, whether or not the navigation can be disabled and the condition to disable. The alias also needs a method to obtain the dynamic part. These changes implement the properties *Static information, Dynamic information*, and *Message visualization* of MoU_SSF2 and *Action selection* and *Condition to disable* of MoU_SSF3.

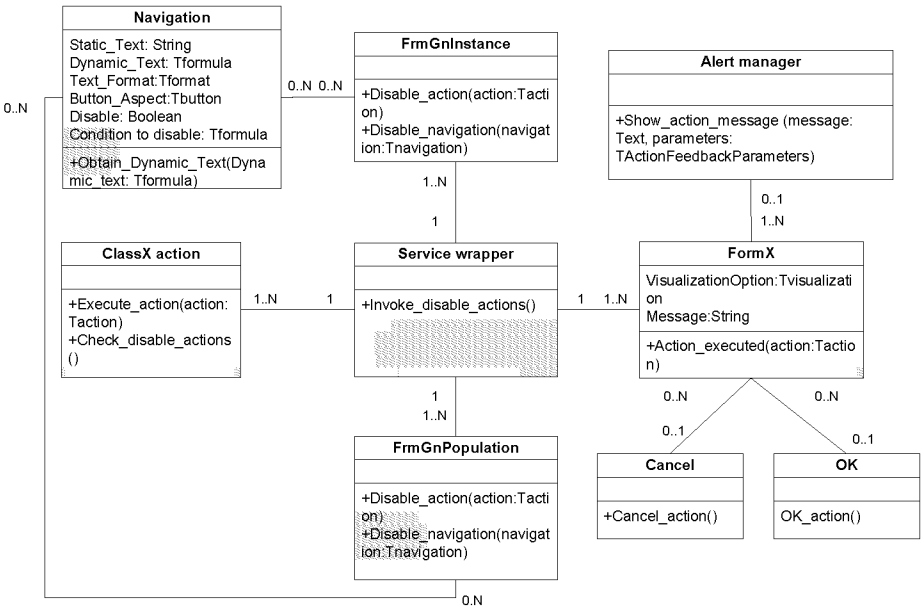

Fig. 7. Class diagram for System Status Feedback

The inclusion of these changes in the model compiler assures the automatic generation of the code that implements the FUFs starting from a conceptual model. Figure 8 shows examples of interfaces that include the

System Status Feedback mechanism. The left part of the figure displays an example of MoU_SSF1 with two types of messages. The failure is represented by means of an emergent window while the success is represented with an icon in the main window. The values for MoU_SSF1 properties are:

- **Service selection**: Create a new account for a customer.
- **Message visualization**: Error messages are displayed in an emergent window while success messages are displayed with an icon in the main window.

With regard to the right part of the figure, we can see an example of dynamic information depending on the selected car (MoU_SSF2) and an example of the state of visible actions (MoU_SSF3). The values for MoU_SSF2 properties are:

- **Static information**: Orders and Invoices.
- **Dynamic information**: The number of orders, the number of unresolved invoices and the number of paid invoices.
- **Message visualization**: Text is displayed with Arial font and with a blue arrow.

Moreover, the navigation button to the list of orders is disabled when there are no instances in the target context. The values for MoU_SSF3 properties are:

- **Action selection**: Navigation to Orders.
- **Condition to disable**: When the list of orders is empty.

Modelling the values of these properties with the conceptual primitives we have presented in this section, the generated system has the interfaces shown in Figure 8. This paper focuses only on the usability mechanism *System Status Feedback,* but there are other 8 usability mechanisms that we have not described in this paper due to space reasons. The list of all the MoUs, properties, changes in the INTEGRANOVA conceptual model and model compiler is detailed in [25].

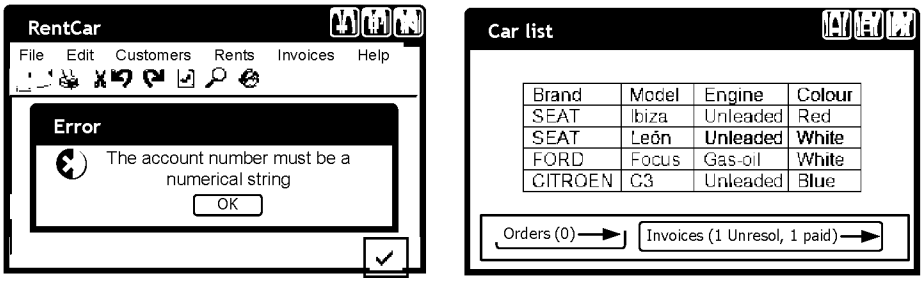

Fig. 8. Examples of interfaces with usability mechanisms

5. A Metamodel to Represent the Modes of Use

In our approach, there is a dependence on the MDD method where we include functional usability features. Definition of Modes of Use (first step) and Identification of Properties (second step) is valid for any MDD method. However, Definition of Conceptual Primitives (third step) and Changes Required in the Model Compiler (fourth step) depend on a specific MDD method, since each MDD method has an exclusive conceptual model and an exclusive model compiler. We have defined new primitives and changes in the model compiler for OO-Method as illustrative example. The changes applied to OO-Method can be useful to guide the analyst to apply the proposal to other MDD method different from OO-Method. However, the reuse of conceptual primitives and changes in the model compiler is not straightforward. In order to mellow this drawback, we need a notation to represent Modes of Use abstract enough to be used in any MDD method. Metamodels [13] are used in software engineering when we aim to define a new language, an alternative to UML. Since usability features depend on interaction features (apart from functionality), and these features are hardly represented within UML, we propose using a metamodel to deal with our approach. The Properties of the Modes of Use are represented in the meta-model by means of classes, attributes and relationships among classes.

Figure 9 shows the metamodel that represents Modes of Use identified from FUFs [23]. Each Mode of Use is represented with a class with the prefix MoU in the meta-model. Next, we explain the meaning of each class:

- **Class, Attribute and Service**: they represent a class with attributes and services.
- **User interface:** this represents an interface.
- **Navigation**: this represents a navigation between two interfaces.
- **Widget**: this represents a widget inside an interface.
- **Display option**: this represents how customizable elements will be displayed, such as labels, backgrounds, buttons, etc. A textual language such as UsiXML [21], can be used to define the visual appearance.
- **Menu entry**: this represents an option of the menu.
- **MoU_SSF1:** this represents MoU_SSF1 (*Inform about the success or failure of an execution*), which informs about the success or failure of each service. The analyst can customize how the feedback will be displayed to the user.
- **MoU_SSF2:** this represents MoU_SSF2 (*Show the state of the stored information*), which provides dynamic and static alias.
- **MoU_SSF3:** this represents MoU_SSF3 (*Show the state of visible actions*), which disables navigations and services that cannot be triggered on a specific condition.
- **MoU_PF:** this represents MoU_PF (*Show the progress of the execution*), which shows a progress bar for complex services. The analyst can choose different display options for the bar.

- **MoU_W:** this represents MoU_W (*Warning message*), which warns subjects about the consequences of executing a service that cannot be undone.
- **MoU_WD:** this represents MoU_WD (*Define a wizard*), which splits complex services into easier steps.
- **Step:** this represents a step of a MoU_WD. Every step has a previous step and a next step (except for the first and the last one, respectively)
- **MoU_STE1:** this represents MoU_STE1 (*Specify the widget type to enter data with a specific format*), which allows to specify the type of the input widget that better helps the user to insert information with a specific format.
- **MoU_STE2:** this represents MoU_STE2 (*Mask definition*), which specifies a mask to help the user insert data according to a specific format.
- **MoU_STE3:** this represents MoU_STE3 (*Default values*), which specifies default values to help the user insert data according to a specific format.
- **MoU_F:** this represents MoU_F (*Favourites definition*), which allows the end-user to define shortcuts to access to favourite interfaces.
- **MoU_GU1 and MoU_GU1:** they represent MoU_GU1 and MoU_GU2 (*Undo change* and *Redo change*), which allow the end-user to undo and to redo last changes respectively.
- **MoU_AO1:** this represents MoU_AO1 (*Cancel during the execution*), which allows the end-user to cancel the execution of a service.
- **MoU_AO2:** this represents MoU_AO2 (*Exit from a scene*), which allows the end-user to leave from interfaces.
- **MoU_MH1:** this represents MoU_MH1 (*Dynamic help*), which displays a dynamic help that appears automatically when the user needs it.
- **MoU_MH2:** this represents MoU_MH2 (*Static help*), which includes in the system a set of help files.

Each instance of this metamodel is a different configuration of functional usability features for a system. There are several advantages of working with the metamodel. First, we can define transformation rules to derive the code that supports the Properties from the instances of the metamodel. These transformations can be defined with existing languages, such as Xpand [32]. Second, the metamodel can be enriched with new usability features that have not been currently studied (different to FUFs), such as, multiple windows or use of toolbars. If we aim to include new features, we must identify new Modes of Use and their Properties and enrich the metamodel to represent identified Properties. Next, we must define new transformation rules to generate the code that implements these new Properties.

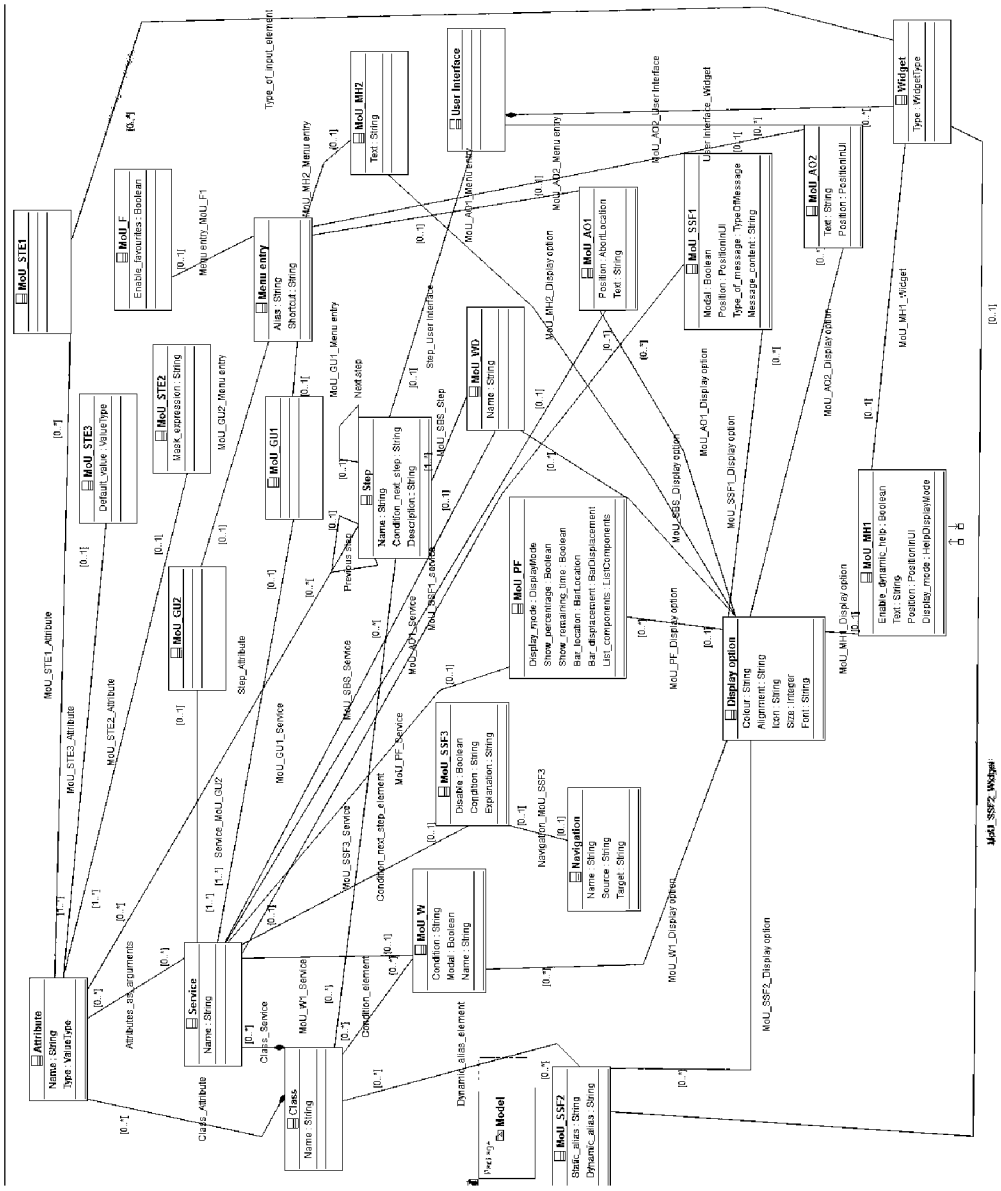

Fig. 9. Metamodel to support FUFs

To sum-up, our approach has two alternatives to include usability features in an existing MDD method: (1) modifying the existing MDD method (its conceptual model and its model compiler) such as we have applied for OO-Method; (2) instantiating the metamodel according to usability requirements, applying transformation rules to generate the code, and combining this code with the code already supported by the existing MDD method. This paper

focuses on the first alternative by several reasons: (1) if we use an existing MDD method that generates fully functional systems (such as OO-Method), we can extend the existing conceptual model with new primitives and the existing model compiler with new rules. At the end, we can generate automatically fully functional systems that support usability features. (2)

The code generated from the metamodel only supports the implementation of functional usability features. Next, we must combine this code with the code generated with the existing MDD method (for example, system functionality or persistency are not included in the metamodel). This combination is not easy in general, and may involve some manual work. (3) The inclusion of new usability features is also supported if we enrich the existing MDD method with new primitives and rules in the model compiler. In this case, the changes are applied directly to the existing MDD method, adding new conceptual primitives and modifying the model compiler.

6. State of the Art

The concept of pattern is one of the most widely used concepts to include usability in the first steps of the software development process because it combines both interaction and functionality. Many authors, such as Tidwell [30], have worked on the definition of usability patterns. The patterns described by Tidwell represent not only usability, but also interaction. Following the same trend, Perzel [27] describes a set of patterns that are oriented to web environments. Perzel distinguishes between patterns for web applications (users must introduce data) and patterns for web sites (users only navigate and visualize information). Another work that aims to bring usability patterns closer to the end-user is proposed by Welie [31]. The patterns of Tidwell and Perzel differ from the patterns of Welie in that Welie distinguishes between the user's perspective and the designer's perspective.

The design patterns proposed by all these authors contain short descriptions about the implications of including patterns in the architecture. However, these descriptions do not explain in detail how to include the patterns in the system and should be expanded with guidelines. This ambiguity is minimized in our proposal, where we specify a set of conceptual primitives to represent usability features and the changes in the model compiler to generate the code that implements these primitives.

Other researchers have been working on defining techniques to capture usability requirements. One of the most relevant works is the one by Bevan [4], who has proposed including usability in the process of requirements capture. The main disadvantage of Bevan's proposal is that he does not specify how the steps should be carried out. Other studies adapt existing requirements models to specifically capture usability requirements (e.g., Cysneiros [9]). Cysneiros proposes modelling usability requirements using the i* notation. He suggests building a catalogue to guide the requirements capture. This notation provides a total view of the requirements and their

relationships with each other including the relationship between usability and functional requirements. The main disadvantages of the Cysneiros's work with regard to our proposal is that the i* notation is ambiguous, far from natural language, and may present contradictions [10]. In our proposal, we present an unambiguous method to deal with usability features. We aim to define conceptual primitives not far from natural language in order to minimize the difficulty in working with them.

The inclusion of usability in an MDD method has not been widely promoted in the HCI and SE communities. When it is discussed, there is a lack of precise detail that makes it difficult to understand how these approaches could work correctly in practical settings. Only a few authors such as Fernández [11], Cachero [5] or Tao [29] have dealt with usability in MDD. Fernández has proposed a Usability Model to evaluate, not model, system usability from conceptual models. His Usability Model has been built using attributes and sub-characteristics defined in the ISO/IEC 9126-1 [16] and in ergonomic criteria [3]. The main disadvantage of Fernández's proposal is that the Usability Model does not include metrics for subjective attributes, such as attractiveness. Cachero proposes measuring the usability of the systems based on navigational models provided by Web engineering methodologies. The author has defined a process to evaluate and progress navigational models. However, some usability features are strongly related to functionality, such as our work states. Therefore, the usability of the system cannot be measured or improved only with a navigational model. Tao has proposed modelling usability by means of State Transition Diagrams, which is very limited. In his proposal, each state transition diagram can be used to represent an interaction between the system and the user. However, when this proposal was evaluated with students in an academic environment, the results showed that this representation is very complex to work with. Also, state transition diagrams cannot represent all the usability features.

Based on the related work, we can conclude that usability is a characteristic of quality that must be considered from initial steps of the software development process to reduce analysts' effort. However, few works have been done to include usability in a holistic software development process based on model-driven development. The contribution of our proposal is to establish how to precisely represent usability characteristics at an abstract level. From the analyst perspective, the system architecture must combine usability and system functionality.

7. Conclusions

This paper proposes a concrete method (called MIFUM) for including functional usability features in any MDD method, where usually, usability features are manually implemented. We have focused our work on FUFs defined by Juristo, but our approach can be applied to other usability features. Proposals different from FUFs with guidelines that describe the alternatives to

configure the usability features are better to apply MIFUM than guidelines with few details, since we can identify conceptual primitives easier.

As an example to illustrate the practical applicability of MIFUM, we have used the OO-Method, an MDD industrial method that generates full functional systems from conceptual models. New primitives and changes in the model compiler of the OO-Method can be customized to any other software development method based on conceptual models. The difficulty of including FUFs in a specific MDD method exclusively depends on the expressiveness of its conceptual model and the level of automation of the model compiler. The OO-Method has a model to represent the interaction and the model compiler can generate full functional systems automatically. However, MDD tools with no model to represent the interaction and a model compiler that generates code semi-automatically would require more effort to include FUFs.

Our approach focuses on enriching the existing conceptual model of the MDD method and its model compiler to support Modes of Use. We have also presented an alternative to this process based on a metamodel. This way, usability features are exclusively represented in the metamodel and we do not need to modify the conceptual model and the model compiler of the existing MDD method. Pros and cons of using this metamodel have been discussed.

The existence of conceptual primitives to represent functional usability features does not guarantee that analysts will use them properly. As future work, we plan to define a tutorial with "best practices" in order to guide the analyst in the use of the new conceptual primitives. In addition, we plan to perform an experiment to measure the level of usability improvement in the generated systems after applying MIFUM to the OO-Method. If functional usability features are defined with primitives, we can define metrics to measure the system usability before generating the system, just using these primitives. As future work, we plan to define metrics to anticipate usability problems. To sum up, this paper is a step forward to provide a holistic MDD method where the conceptual model represents not only functionality and persistency, such as the SE community has been working on, but also interaction and usability features.

Acknowledgment. This work has been developed with the support of MICINN (PROS-Req TIN2010-19130-C02-02, TIN2011-23216), UV (UV-INV-PRECOMP12-80627), GVA (ORCA PROMETEO /2009 /015), and co-financed with ERDF. We also acknowledge the support of the ITEA2 Call 3 UsiXML (20080026) and funding by the MITYC under the project TSI-020400-2011-20.

References

1. Aquino, N., Vanderdonckt, J, Valverde, F., Pastor, O.: Using Profiles to Support Model Transformations in the Model-Driven Development of User Interfaces, Proc. of 7th Int. Conf. on Computer-Aided Design of User Interfaces CADUI'2008, Albacete, Spain, Springer, 35-46, (2008).

2. Bass, L., and Bonnie, J.: Linking Usability to Software Architecture Patterns Through General Scenarios, The journal of systems and software, Vol. 66, 187-197, (2003).
3. Bastien, J.M., Scapin, D.: Ergonomic Criteria for the Evaluation of Human-Computer Interfaces, Rapport technique de l'INRIA, (1993).
4. Bevan, N., and Bogomolni, I.: Incorporating User Quality Requirements in the Software Development Process, Proc of International Software and Internet Quality Week Conference (QWE), (2000).
5. Cachero, C., Meli, S., Genero, M., Poels, G., and Calero, C.: Towards improving the navigability of Web applications: a model-driven approach, European Journal of Information Systems, vol. 16, 420–447, (2007).
6. CARE Technologies S.A., 2013, http://www.care-t.com
7. Ceri, S., Fraternali, P. and Bongio, A.: Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. Computer Networks and ISDN Systems, 33(1-6), 137-157, (2000).
8. Comstock, E. and W. Duane (1996). Embed User Values in System Architecture: The Declaration of System Usability. CHI 96.
9. Cysneiros, L. and Kushniruk, A.: Bringing Usability to the Early Stages of Software Development, Proc of 11th International Requirements Engineering Conference, California, USA, IEEE, 359- 360, (2003).
10. Estrada, H., Martínez, A., Pastor, O. and Mylopoulos, J.: An empirical evaluation of the i* framework in a model-based software generation environment, Proc of 18th CAISE, Luxemburg, Springer LNCS 4001, 513-527, (2006).
11. Fernández, A., Abrahao, S. and Insfran, E.: A Web Usability Evaluation Process for Model-Driven Web Development. 23rd International Conference on Advanced Information Systems Engineering, CAiSE2011, London, Springer.108-122, (2011).
12. Folmer, E., Bosch, J.: Architecting for usability: A Survey. Journal of Systems and Software, Vol. 70(1), 61-78, (2004).
13. Fuentes-Fernández, L. and A. Vallecillo-Moreno (2004). "An Introduction to UML Profiles " European Journal for the Informatics Professional 5(2): 5-13.
14. FUFs 2013: http://hci.dsic.upv.es/FUFandMoU/FUFList.html
15. Hailpern B., Tarr, P.: Model-Driven Development: the Good, the Bad, and the Ugly, IBM Syst. J., vol. 45, pp. 451-461, (2006).
16. ISO/IEC 9126-1, Software engineering - Product quality - 1: Quality model, (2001).
17. Juristo, N., Moreno, A.M. and Sánchez-Segura, M.: Guidelines for Eliciting Usability Functionalities, IEEE Transactions on Software Engineering, Vol. 33,744-758, (2007).
18. Juristo, N., Moreno, A.M., Sánchez-Segura, M.: Analysing the Impact of Usability on Software Design. Journal of System and Software, Vol. 80(9), 1506 -1516, (2007).
19. Koch, N., Knapp, A., Zhang, G., and Baumeister, H.: UML-Based Web Engineering, An Approach Based On Standards. In Web Engineering, Modelling and Implementing Web Applications, Springer, 157-191, (2008).
20. Lauesen, S. (1998). Usability Requirements in a Tender Process. Computer Human Interaction Conference, 1998, Australia.
21. Limbourg, Q., Vanderdonckt, J.: Usixml: A User Interface Description Language Supporting Multiple Levels Of Independence. Engineering Advanced Web Applications. Rinton Press, Paramus, New Jersey (2004).
22. Mellor, S., Clark, A.N., and Futagami, T.: Guest Editors' Introduction: Model-Driven Development. IEEE Software, Vol. 20, 14-18, (2003).
23. MoU 2013: http://hci.dsic.upv.es/FUFandMoU/UW_list.html
24. Olivé, A.: Conceptual Modeling of Information Systems, Springer, (2007).

25. Outcomes 2013: http://hci.dsic.upv.es/FUFandMoU/ChangesList.html
26. Pastor, O. and Molina, J.: Model-Driven Architecture in Practice. Valencia, Springer, (2007).
27. Perzel, K. and Kane, D.: Usability Patterns for Applications on the World Wide Web, Proc. of PloP'99 Conference, (1999).
28. Sendall S. and Kozaczynski W.: Model Transformation: The Heart and Soul of Model-Driven Software Development, IEEE Software, vol. 20, pp. 42-45, (2003).
29. Tao, Y.: An Adaptive Approach to Obtaining Usability Information for Early Usability Evaluation, Proc of IMECS, 1066-1070, (2007).
30. Tidwell, J.: Designing Interfaces, O'Reilly Media, (2005).
31. Welie, M.v., Traetteberg, H.: Interaction Patterns in User Interfaces. Proc of 7th Pattern Languages of Programs Conference, Illinois, USA, (2000).
32. Xpand: http://www.eclipse.org/modeling/m2t/?project=xpand