State-Action Pairs for Contextual Interface Editing

Inés Pederiva[1], Jean Vanderdonckt[1,2], Sergio España[1], Ignacio Panach[1], and Oscar Pastor[1]

[1]Universidad Politécnica de Valencia, Dep. de Sistemas Informáticos y Computación
Camino de Vera s/n, 46071 Valencia, Spain
[2]Université catholique de Louvain, Louvain School of Management,
Place des Doyens, 1 – 1348 Louvain-la-Neuve, Belgium
{ipederiva, jvanderdonckt, sergio.espana, jpanach, opastor}@dsic.upv.es,
jean.vanderdonckt@uclouvain.be

Abstract. The editing of a user interface resulting from model-to-model and model-to-code transformations in Model-Driven Architecture consists of performing manual changes to address user requirements which have not been supported during the transformations. These requirements may include customization, users' preferences, and compliance with corporate style guidelines. This paper introduces a editing process into a user-interface model. This process includes a series of beautification operations based on a formal definition, as well as a constrained editor that enables designers to apply these beautification operations on a user interface. All manual changes done using these beautification operations are transformed into model-to-model transformations, thus reducing the problem of round-trip engineering. The paper also demonstrates that this process significantly reduces the number of manual changes performed on user interfaces of information systems, while preserving the quality properties induced by the transformations.

Keywords: Beautification operation, beautification process, human-computer interaction model, round-trip engineering, model-driven engineering, quality by construction, user interface description language, user interface code tweaking.

1 Introduction

The complete support of User Interfaces (UIs) requirements in Model-Driven Engineering (MDE) [4,22]. The user requirements to be addressed usually fall into two categories (Fig. 1): requirements that are effectively supported by applying model-to-model (M2M) and model-to-code (M2C) transformations [10] and requirements that are not supported because they are not covered by these transformations. This dichotomy of requirements leads to two extremes: on the one hand, the UI that has been automatically generated by these transformations is assumed to be usable by the end user or it is simply taken for granted because of resource limitations, or on the other hand the UI is subject to manual modifications in an attempt to address the remaining user requirements. These manual modifications take two basic forms (Fig. 1): the generated UI code is tweaked manually or it is imported in a UI builder to be edited

by direct manipulation to produce the final UI. These modifications typically include operations like: changing the stylistic attributes of the widgets, rearranging and the resizing widgets, reshuffling contents across containers, applying a specific look & feel, and programming the widgets with behaviors that were impossible to generate before the modifications. These operations are performed to make the automatically generated UI compliant with the corporate style guidelines or to customize it with respect to the remaining requirements. All these modifications are usually referred to as *beautification operations* sinc they are intended to beautify the manual changes brought to automatically generated UI. The whole process is known as *UI beautification*.

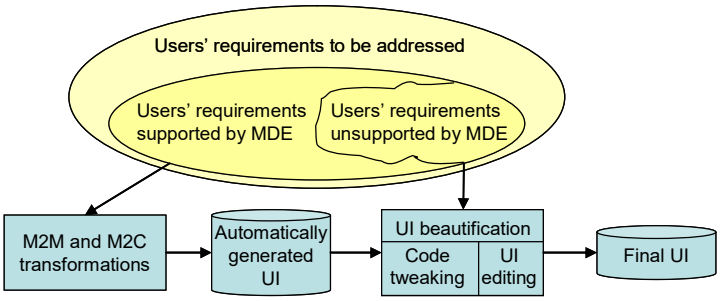

Fig. 1. The location of the beautification process in the development life cycle.

Unfortunately, this task is very sensitive to mistakes [1]: understanding generated code is usually a complex activity, and what has been constructed according to the MDE approach may easily be destroyed by manual beautification. Moreover several quality features guaranteed by construction in the MDE approach could be endangered by the beautification: usability [1] (e.g., every UI has limited usability, but if this usability is further diminished by manual modifications performed by an inexperienced designer, it will be even worse); consistency between the UI and its model [3] (e.g., any deleted widget will introduce a discrepancy with respect to the initial model); correctness [7] (e.g., if a generated behavior is modified, it may no longer preserve semantic constraints like persistency which had been ensured by construction before the modification), and error-free benefit [15] (e.g., code tweaking is very error-prone).

In the field of Computer Graphics, beautification [17] is referred to as the direct manual sketching of a shape, a drawing, or an illustration in an editor and its automatic replacement by a 'beautified' symbol with the correct transformation. This is performed all in one step in order to remove manual clutter [8]. Thus, it is possible to rapidly and informally draw sketches that are automatically recognized by a system and transformed into their corresponding symbol in the drawing or the illustration [18]. By analogy, in Computer Science, we define *UI beautification* as the whole process of improving the automatically generated UI with manual modifications to address unsupported user requirements.

The consequence of this manual beautification is that all efforts are not saved and are lost if a new UI is regenerated [19]: if the UI model changes, the generated UI changes accordingly but it is no longer compliant with user requirements that have been addressed in the previous phase. To alleviate this problem, researchers in MDE

have introduced various solutions to the so-called *round-trip engineering* [2]: manual modifications could be saved (e.g. as a way to easily reapply them), they could be interpreted, abstracted, and replaced by a 'beautified' operation to be propagated to the model that initiated the M2M and M2C transformations. These operations could then be replicated each time the whole set of transformations is reapplied. Although this problem has received considerable attention in MDE in general, to our knowledge no comprehensive solution currently exists to address this problem in the context of MDE of UIs. There are several reasons why this problem remains unsolved:

- Existing MDA-compliant methods and tools (commercial or research and development) either provide mechanisms to change the model that do not mesh well with beautification or do not enable any changes in the final UI resulting from the application of the MDA.
- Interpreting and abstracting manual operations is a very complex task that requires a particular implementation in a dedicated UI builder.
- The development of such a new UI builder would require considerable development costs, which are usually beyond the capabilities of research and development teams.
- There is no general framework for abstracting beautification operations so they can be automatically modeled.
- The literature does not provide any statistics about the need, the frequency, or the usage of such beautification operations.

This paper addresses the shortcomings mentioned above by introducing a framework of beautification operations that provides a constrained UI editor where these beautification operations can be applied without endangering the qualities provided by the MDE approach. One approach consists of letting designers be free to do what they want in any UI builder, however, the quality of the resulting UI will depend heavily on their expertise. Another approach is to develop a brand new UI builder that supports the beautification operations using round-trip engineering, but this requires too much effort. Our solution provides a balance between these two extremes: we present a constrained UI editor equipped with some beautification operations that can only be applied in the context of the editor, preserving the quality features provided by the MDE. The new definition of *UI beautification* becomes: the whole process of improving the automatically generated UI with beautification operations to address unsupported user requirements while preserving the qualities provided by MDE.

The remainder of this paper is structured as follows: Section 2 reviews how the beautification process was first addressed in Software Engineering, and then how it has been addressed in the domain of MDE of UIs. Section 3 develops the methodological context in which our solution of a constrained editor is developed and reports observations of the tweaking performed using OO-Method. Section 4 describes how our framework performs beautification operations on graphical representations of an underlying UI model in order to support round-trip engineering. A running example is provided for this purpose to demonstrate the effectiveness and the efficiency of this solution. Section 5 reports the experience gained by using this method and its corresponding constrained UI editor. This section also identifies directions for future work resulting from this experience.

2 State of the Art

The beautification of automatically generated computer-based systems and its corollary, the round-trip engineering problem [2], are concluded to go far beyond a simple combination of forward and reverse engineering [21]. Many different techniques have been proposed to address this problem. Some of these are design patterns, framework-specific modeling languages [2], model reconciliation [21], etc. Although these techniques are generally applicable in the discipline of software engineering, they do not exploit the full potential of UI models, which are usually visual in nature: all UIs incorporate visual aspects that should be dealt with in case of beautification.

In Human-Computer Interaction (HCI), MECANO [20] was the first project to recognize the need for beautification support in the generation process: the methodological guidelines recommended that designers propagate their manual modifications into elements and relationships in the underlying model for reuse. Clerckx *et al.* enumerate an extensive list of similar rules that can be manually applied to the model after a transformation has been performed. Their DynaMo-AID design process is divided into several steps providing rules for propagating changes across models that are involved in different levels. MOBI-D [19] and TEALLACH [6] enable a designer to start a project from any model (task, domain, UI), thus propagating the consequences to the other models by linking and derivation. These mechanisms are similar, but not intended to really support UI beautification. WISDOM [14] also recommends keeping the models consistent with each other when a mode has been updated. In da Silva's survey of model-based tools and techniques [4], none of them provides any explicit support for UI beautification. This shortcoming is also observed in major commercial software that automatically generates UI, such as Genova [5], JaxFront [9], and OlivaNova [15]. In the following sections, we formally define beautification operations so that, in theory, they could work in any of the above environments. To be practical, we illustrate the beautification process in the context of a specific MDA-based method: the OO-Method [16].

3 Model-Driven Engineering of User Interfaces in the OO-Method

OO-Method [16] is a software development method that is MDA-compliant, i.e., it involves models of the future interactive system at different levels of abstraction (CIM, PIM, PSM [10] – Fig. 2) and provides an explicit transformation mechanism between them. The method is initiated by specifying the system functional requirements and develops the final interactive system through consecutive transformations. This method is supported by a software suite [15] that edits the various models involved and applies subsequent transformations until the final code is generated for different computing platforms: ASP, .Net, .JSP, Java, and C#.

The requirements elicitation in OO-Method [16] gathers all the user requirements in the *Requirements Model* by specifying the system functionality in the *Mission Statement* and the *Function Refinement Tree*. The *Use Cases* detail each function of the tree. When this model is complete, system specifications are output independently of the implementation or the technological space (*Computation Independent Model* - CIM).

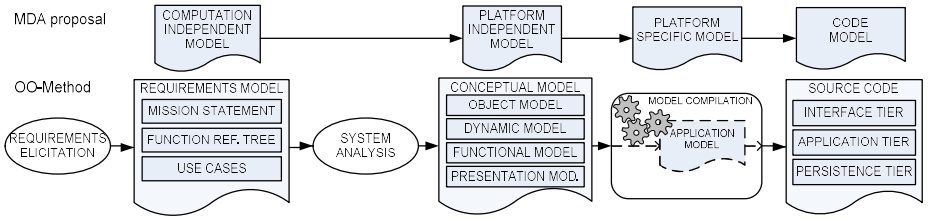

Fig. 2. Correspondences between the MDA proposal and the OO-Method

The *Conceptual Model* is equivalent to the *Platform-Independent Model* (or PIM) for MDA. This model goes from the problem space to the solution space by specifying four complementary system views: the *Object Model* specifies the static properties of the interactive application by defining the classes and their relationships; the *Dynamic Model* controls the application objects by defining their life cycle and their interaction with each other; the *Functional Model* describes the semantics associated with the changes in the object states; and the *Presentation Model* models the UI.

The *Conceptual Model*, once achieved, is submitted to model compilation (M2C transformation) in OlivaNova. For any target computing platform, the *Source Code* is automatically generated and structured according to architecture with three tiers: the *Interface Tier*, the *Application Tier*, and the *Persistence Tier*. A fully functional interactive application can be generated, which is not limited to database or UI (for more detail, see [15,16]). To obtain the UI of this fully functional application, the *Presentation Model* (PM) abstractly defines how the user will manipulate the final UI based on a set of UI design patterns organized in three levels as can be seen in Fig. 3. The PM is decomposed into *Interaction Units* (IUs), which represent the main interactive operations that can be performed on the domain objects. T and which characterize the UI widgets supporting these operations [11]:

1. The *Instance IU* shows a single object at a time, that is, one instance of a class.
2. The *Population IU* shows a group of similar objects.
3. The *Master/Detail IU* shows a hierarchical view of relationships between objects.
4. The *Service IU* modifies objects, their attributes and their relationships.

The four IUs mentioned above define the second layer of patterns of a PM. The next level of decomposition of the PM consists of restricting and specifying the behavior of each IU in the PM into an elementary pattern. For example, if a *Population IU* is being specified, then five elementary patterns could be attached to it [11]:

(a) A *Filter Pattern* filters any set of objects to display only the objects needed.
(b) The *Order Criteria* specifies the order in which the objects are shown.
(c) The *Display Set* restricts which attributes or properties of the objects are going to be presented. When the objects are listed with these restrictions, the user can execute actions on the objects or reach related objects by the use of the following two elementary patterns.
(d) The *Navigation Pattern* specifies which navigation type should be ensured between the objects.
(e) The *Action Pattern* specifies what semantic functions (the methods of the UML class diagram that were used to represent the Object Model) can be triggered for each object in the UI.

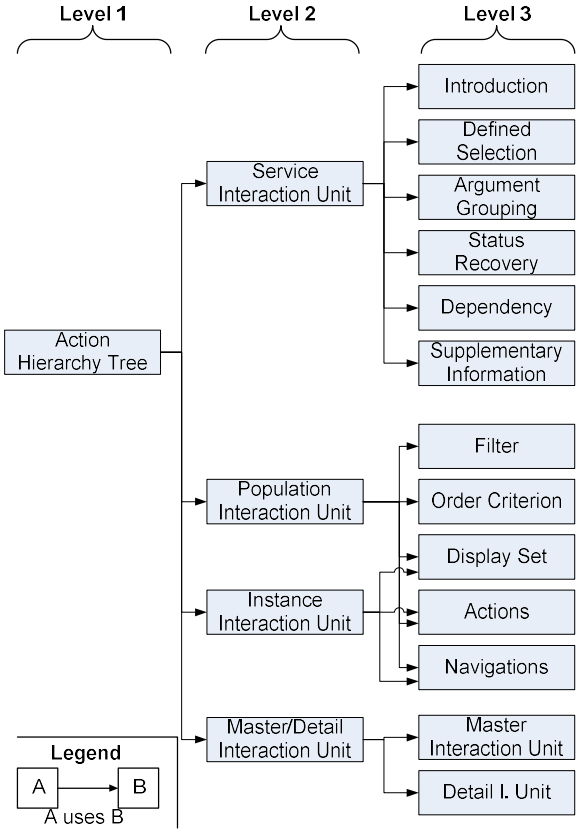

Fig. 3. The user interface design patterns defined in [11]

To illustrate the beautification process, we have selected a Population IU extracted from a real-world application delivered to Aguas del Bullent S.A. (http://www.aguas delbullent.com/index_en.html), a drinking water supply service company located in Oliva, Alicante (Spain). This application was chosen for several reasons: it has been entirely produced according to the OO-Method and generated by the OlivaNova without any manual modification, it represents a medium-sized interactive application of moderate complexity; it is a genuine application which is used today by several end users; and it has received the Microsoft certification of quality. Fig. 4 reproduces a Population pattern for displaying the water meters of customers located in a certain region. This Population pattern is decomposed into the five elementary patterns described above. The Display Set is the only mandatory elementary pattern required to compose a Population pattern. The UI shown in Fig. 4 was chosen because it includes all five elementary patterns simultaneously. The following section shows how this automatically generated UI is subjected to beautification operations requested by the end users through their user requirements.

The UI in Fig. 4 only addresses the portion of user requirements supported by the MDA approach adopted here. However, the beautification process addresses the other portion of MDA-unsupported user requirements.

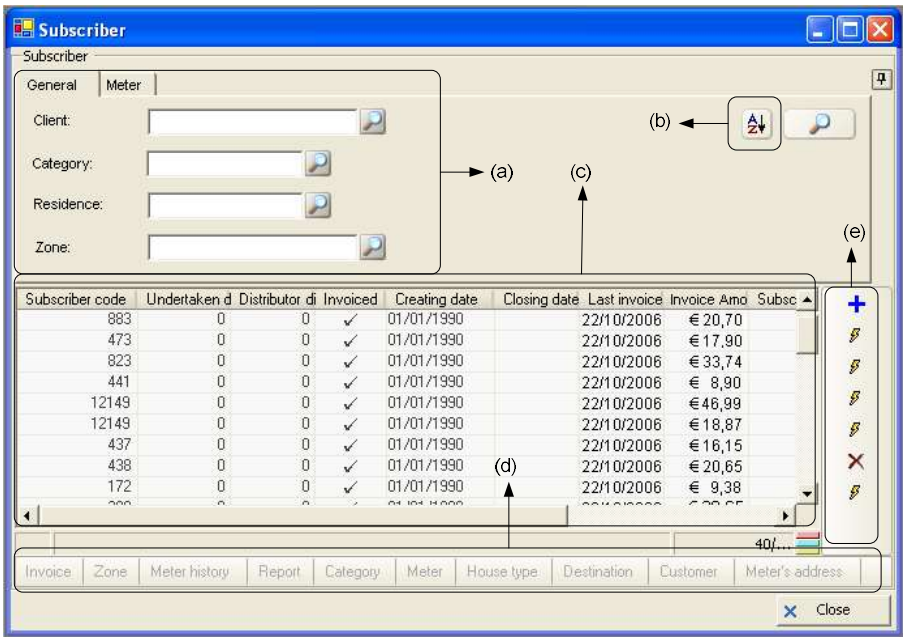

Fig. 4. Our running example as generated: a Population IU attached to a Filter pattern (a), an Order Criteria (b), a Display Set (c), a Navigation pattern (d), and an action pattern (e).

4 The Beautification Process

4.1 Purpose of Beautification Operations

The need for a beautification operation arises as soon as there are user requirements that are unsupported by the MDA approach. For example, for the user interface shown in **Fig. 4**, users complain that the list of objects displayed may become very long since many customers could be selected. To improve the legibility of the list, it was decided that every four lines, the background color should change to a usable background color (e.g., light blue) to avoid interfering with the data. Since this feature is not supported by the MDA approach adopted here there is no modeling operation for it. Therefore, if an aspect is not modeled, the final code does not support that operation and therefore a manual modification is necessary. The goal now is to turn these manual modifications into beautification operations, which will ultimately be considered as modeling operations. To decide which manual modification should be included or excluded from the beautification process, a **statistical analysis** was conducted on three professional interactive applications produced by the MDA-compliant OO-Method and generated by the OlivaNova software:

1. A *small-sized* interactive application: ProAM consists of a golf management application especially designed for golf tournaments, which is generated for both JSP and Visual Basic. A total number of 19 manual modifications were observed from the JSP version, including adding an image, changing the background color, changing a link, changing the login, changing a dynamic menu to a static menu.

2. A *moderately-sized* interactive application: MultiLanguage consists of an application that runs the same user interfaces but in different languages. It was generated for C# and a total amount of 144 manual modifications were reported for this version. Adding a message box, adding a public variable, adding a reference to a library, initializing filter variables and removing or showing a control widget where some of the operations
3. A *large-sized* interactive application: Alligator consists of an invoicing system for multiple inter-related companies with advanced reporting functions, which is generated in Visual Basic. A total amount of 252 manual modifications have been examined for this application. Among others, adding a popup to a grid, adding a function in selection methods and computing the total of items located in a column) where some of those modifications.

The total of 415 manual modifications was divided into two types: those modifications made by the developers and those that were not (171 out of 252 were considered for the large-sized application). This set was then divided into a subset of modifications that were relevant to the *Presentation Model* (PM) (103 out of 171) and those that were not relevant to the PM (58 out of 171) but are relevant to the other models involved in the OO-Method (Fig. 2). A joint analysis of frequency and importance made on the modifications relevant to the PM determined the most frequent modifications and their level of importance in terms of the impact on the generated code. Therefore, a manual modification is included for further consideration as a beautification operation for the following reasons:

- The operation was observed in most applications.
- The operation occurred with a significant frequency.
- The operation was realistic in terms of future implementation support.
- The operation was of at least moderate importance.

In order to consistently estimate the importance of a modification, a classification of these modifications was introduced, and is described in the next section.

4.2 Classification of Beautification Operations

In order to classify a manual operation and, therefore, a subsequent beautification operation, Nielsen's linguistic model of interaction [13] was selected for the following reasons: it decomposes a human-computer interaction in terms of seven inter-related, but independent, levels with a communication protocol between them; it has already been successfully used to classify usability guidelines according to their level of importance; it allows identification criteria to univocally locate each modification to one and only one level; and it is possible to find a manual operation for each of the seven levels. Table 1decomposes a simple goal (i.e., delete a paragraph in a letter) into subsequent units of interaction for each level. We do the same here with a user's goal attached to the Population Interaction Unit Pattern shown in Fig. 4:

- **Level 1 (Goal)**: expresses a user's mental goal, such as "search for a particular customer having a water meter in a specific region".
- **Level 2 (Pragmatic)**: translates this mental goal into a task to be carried out in the system according to the system concepts, such as "search for a subscriber having at least one water meter in zone *x*" (Fig. 4).

- **Level 3 (Semantic)**: translates the real-world objects into system objects and functions, such as "search for a subscriber with a code region filled in" (Fig. 4).
- **Level 4 (Syntactic)**: structures the semantic into an ordered sequence of operations in time and space, such as "select a zone code from the list and launch a query".
- **Level 5 (Lexical**): decomposes each operation into the smallest possible pieces of information, such as "a zone code".
- **Level 6 (Alphabetic)**: specifies the unit of information (e.g., a lexeme, a metric) for each information item, such as "an integer for representing the zone code".
- **Level 7 (Physical**): specifies the physically-coded information in terms of light, sound, color, etc., such as "display the integer in black on white for input".

Table 1. Definition of the seven levels of Nielsen's linguistic model of interaction [13].

Level	Title	Units	Definition	Example	World
1	Goal	Concepts of real world	Mentalization of a goal, a wish in the user's head	Delete a paragraph from my letter	Conceptual
2	Pragmatic	Concepts of system	Translation of a goal into system concepts	Delete 6 lines of the current paragraph in the edited text	
3	Semantic	Detailed functions	Real world objects translated into system objects manipulated by functions	Delete a certain amount of lines	
4	Syntactic	System sentences	Time & space sequencing of information units	DELETE 6	Perceptual
5	Lexical	Information units	Smallest elements transporting significant information: word, figure, screen coordinates, icon	[DELETE] command, [6] number	
6	Alphabetic	Lexems	Primitive symbols: letter, numbers, columns, lines, dots, phonems, ...	D, E, L, E, T, E, 6	Physical
7	Physical	Physically coded information	Light, sound, physical moving	Pressing [CTRL]+[D] followed by [6]	

4.3 Definition of a Beautification Operation

According to Table 1, any beautification operation can be classified into one and only one level. If any ambiguity persists after an initial classification, it surely means that the beautification operation should be decomposed into smaller operations which can then be classified unambiguously. In order to show a significant, and yet reasonably small, set of operations, we would like to execute five beautification operations belonging to five different levels for our running example (Fig. 4) as follows:

1. **Level 7 (Physical)**: *Specify (rowHighlightingType)* specifies that for every *n* number of lines in a table, the background color of this line should be set to a color that is different from the foreground color to ensure contrast. The end users need to have a more legible table to be able to find data in it easily. For instance, in Fig. 4, one in every four lines of the "Subscriber table" should be highlighted. This operation belongs to the Physical Level because it affects the physical appearance, not the contents.

2. **Level 6 (Alphabetical)**: *Convert (inputMetricUnit, outputMetricUnit)* converts data expressed according to one metric unit into another one. For instance, in Fig. 4, the currency of a price displayed in the column "Invoice amount" should be converted from the Euro (€) currency into the United States Dollar (U$D) currency. This operation belongs to the Alphabetical Level because it only changes the numerical value of prices with another symbol to support internationalization.
3. **Level 5 (Lexical)**: *Specify (buttonPresentationType)* specifies whether a push button should be presented with one label only (l), with an icon only (i) or with both (i+l), according to the usability guideline saying that combining an icon and a label better addresses the users' variation in terms of function recognition. For instance, in **Fig. 4**, a push button of the navigation pattern (e) could be presented with an icon and label together. This operation belongs to the Lexical Level because textual and/or graphical information is presented for the same object; we are dealing with the lexis of the User Interface description language.
4. **Level 4 (Syntactical)**: *Substitute (widgetType)* replaces a widget of a given type by a widget of another type by transferring its properties from the initial one to the substituted one. For instance, in **Fig. 4**, the edit box attached to "Category" may be substituted by a drop-down combo box because the amount of categories remains fixed and will no longer be changed. This operation belongs to the Syntactical Level because it changes the sequence of actions that the user has to do in order to select a category.
5. **Level 3 (Semantic)**: *Specify (conditionalDisplay)* changes the value of a widget property depending on whether a semantic condition is satisfied or not. For instance, in **Fig. 4**, the "Invoiced" flag should be changed to another symbol depending on whether the invoice has been issued or not. This operation belongs to the Semantic Level because the presentation only changes according to a semantic change of the object (that is, the values of its attributes).

These five examples show that a beautification operation is executed depending on the widget types involved, the interaction unit concerned, and the elementary patterns present. Therefore, a *beautification operation* is now formally defined as a *State-Pair Action* (SAP) $\mathcal{Z} = \langle s, a \rangle$ where

s = a state of a IU where the beautification operation could be applied.
a = an action to be performed on the state s when it is found.

A SAP consists of a representation of the Interaction Units (IU) contents prior to executing the action (the *state*) and a description of this action at an appropriate level of abstraction (the *action*). This SAP notion has been selected here because of its common usage in various disciplines such as Reinforcement Learning, Human-Robot Interaction, and Artificial Intelligence and because of its capacity to combine several pairs together to address a state space. Consequently, these five examples of beautification operations could be formally expressed as follows:

$\mathcal{Z}_1 = \langle$ *table in: DisplaySet*, *Specify (rowHighlightingType)* $\rangle$
$\mathcal{Z}_2 = \langle$ *cell in:table in: DisplaySet*, *Convert (inputMetricUnit, outputMetricUnit)* $\rangle$
$\mathcal{Z}_3 = \langle$ *button in:Navigation in: PopulationIU*, *Specify (buttonPresentationType)* $\rangle$
$\mathcal{Z}_4 = \langle$ *inputText in:*, *Substitute (widgetType)* $\rangle$
$\mathcal{Z}_5 = \langle$ *cell in:*, *Specify (conditionalDisplay)* $\rangle$

Note that, in the above examples, an action could be performed independently on any context (e.g., *table in:*); and in the context of a particular IU (e.g., *table in: PopulationIU*), in the context of particular elementary pattern in an IU (e.g., *table in: DisplaySet in: PopulationIU*). If the same beautification operation can be applied on different widgets considered in different contexts, the beautification operation is repeated with the same action. Depending on the scope of the action and depending on what needs to be beautified, the action could be applied on a particular widget, on a particular widget in a container, or to a series of widgets). Therefore, we define a beautification operation as follows:

- A beautification operation is said to be *single-source* when one widget belongs to the action domain. It is said to be *multiple-source* when many widgets belong to the action domain.
- A beautification operation is said to be *single-target* when one widget belongs to the action co-domain. It is said to be *multiple-target* when many widgets belong to the action co-domain.

Now that a beautification operation has been properly defined, the next section describes the beautification process and then decomposes this process into three steps, each being detailed in the respective subsections.

4.4 The Steps of the Beautification Process

Fig. 2 shows that the OO-Method, as supported by OlivaNova, produces a conceptual model, which, in turn, gives rise to the final interactive application (including its UI) after performing mode-to-code (M2C) transformation. Thanks to the concept of beautification, this process can be improved through beautification (Fig. 5). This process is decomposed into three steps which are detailed in the following subsections.

Step 1: Derivation of a Concrete User Interface Model from the Presentation Model. Since the Presentation Model contains an abstract definition of the future UI in terms of IUs and attached elementary patterns, it is considered to be the best candidate to apply a M2M transformation in order to derive a Concrete User Interface Model from it. This model needs to fulfill at least two requirements:

1. In order to apply any beautification operation, it is necessary to know which widget must be replaced depending on the context.
2. In order to manipulate a working model, an internal UI representation that is subject to the beautification operations must be maintained.

The Concrete User Interface (CUI) model of the USer Interface eXtensible markup Language (UsiXML – http://www.usixml.org) has been selected because it satisfies these two requirements and allows us to provide the following definitions:

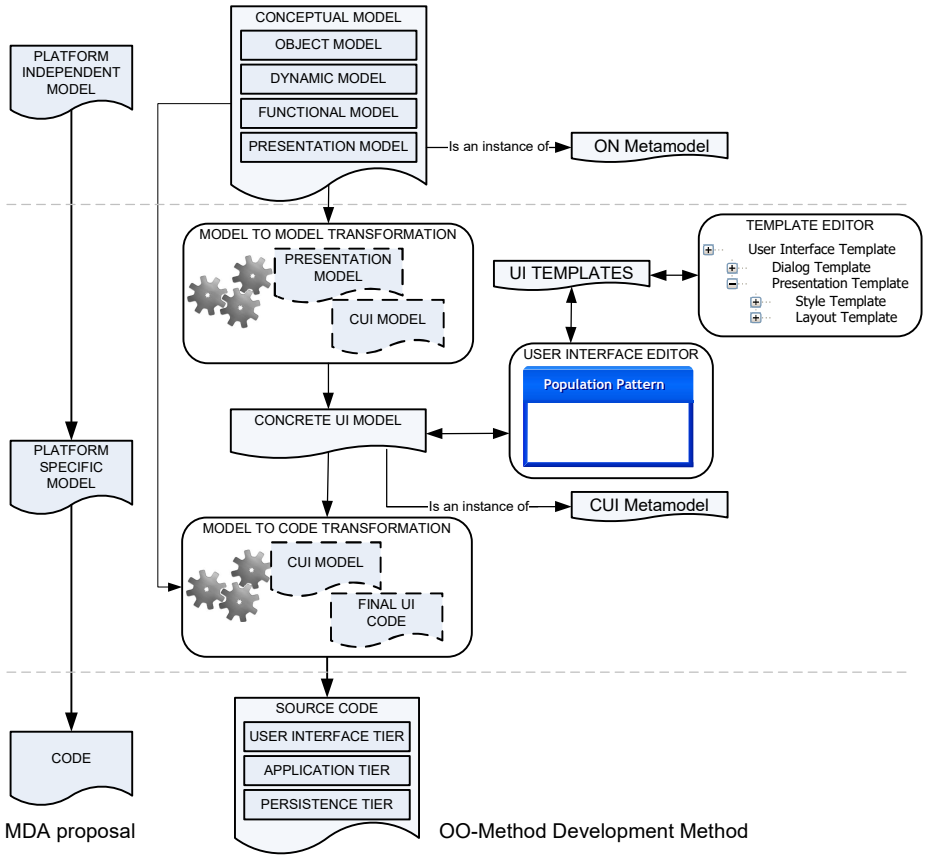

Fig. 5. Update of the OO-Method as defined in Fig. 2 with support for beautification.

- A *Concrete User Interface* (CUI) consists of an abstraction of a final UI independently of the peculiar widgets used in a particular computing platform, toolkit, window or presentation manager, thus resulting in a characterization of a UI in terms of *Concrete Interaction Objects* (CIOs). In UsiXML, there are classes for, graphical CIOs, vocal CIOs, etc. Only graphical CIOs will be considered here.
- Let $\mathscr{C}$ be the set of all graphical CIOs to be considered here.
- A *graphical CIO*, or a CIO for short here, is formally defined as a couple $c = \langle t, A \rangle$
  - where t = type of the CIO A = *decorator* iff c is non-interactive, *graphicalIndividualComponent* iff c is interactive and $\nexists\, c' \in \mathscr{C}$ such that $c' \subset c$, *graphicalContainer* iff c is interactive and $\exists\, c' \in \mathscr{C}$ such that $c' \subset c$, respectively.
  - where A is a set of triple (a_i, t_i, v_i): A = { (a_i, t_i, v_i) }, of cardinality $|A| = n$ where
    - i. a_i (i=1,…,n) = ith attribute of c
    - ii. t_i (i=1,…,n) = data type of the ith attribute of c : $t_i \in$ {boolean, time, date, integer, string}
    - iii. v_i (i=1,…,n) = value of the ith attribute of c = *null* if a_i is empty

- Therefore $\mathcal{C}$ = {decorators, graphicalIndividualComponent, graphicalContainer}
- UsiXML includes several CIOs for these different types, such as: a separator (decorator), inputText, outputText, radioButton, checkBox, listBox (graphicalIndividualComponent), dialogBox, window, and tabbedDialogBox (containers). Fig. 6 illustrates the attributes of a list box that falls into four categories depending on their level of inheritance. Thus, according to the above definition, we can state: listBox = { (id, string, null), (name, string, null), …, (isVisible, boolean, null),…, (mnemonic, string, null),…, (maxLineVisible, string, null),…) to define the internal representation of a list box.
- A CIO is said to be *totally instantiated* when all its attributes a_i have been assigned to a value v_i: c is totally instantiated $\Leftrightarrow \forall$ i=1,…,n: $v_i \neq$ null.
- A CIO is said to be *partially instantiated* when some attributes a_i have been assigned to a value v_i: c is partially instantiated $\Leftrightarrow \exists$ i=1,…,n: $v_i \neq$ null.
- A CIO is said to be *uninstantiated* when all attributes a_i have not been assigned to any value v_i: c is uninstantiated $\Leftrightarrow \forall$ i=1,…,n: $v_i =$ null.

Once a Presentation Model is created for a UI, a corresponding CUI model is therefore derived as a tree of structured and partially instantiated CIOs whose root is a graphicalContainer. Each IU and each pattern contained in the Presentation Model is transformed into a CUI. This transformation is straightforward since each IU and each pattern is described in terms of uninstantiated CIOs.

Step 2: Execution of the beautification operations. Once a CUI Model has been derived, it can be submitted to beautification operations, applying model-to-model transformations. For this purpose, the CUI Model is opened in the constrained GUI editor. Each partially instantiated CIO belonging to the CUI Model is then subject to beautification operations. The GUI constrained editor detects potential SAPs to be applied by examining the states defined in each SAP and matching them to the CIOs of the GUI Model. If a CIO is subject to a particular SAP, the constrained GUI editor allows the designer to apply the corresponding beautification operation through a contextual menu (Fig. 6): when the cursor moves over a CIO subject to beautification (a), a contextual menu appears (b) which could be pulled down (c) so as to select the desired operation and to apply it instantly (d).

Fig. 6. Sequence of user actions to trigger a beautification operation: (a) selection, (b) contextual menu, (c) selecting a beautification operation, (d) applying it.

When it receives a new SAP as input, the constrained GUI editor finds the collection C of action steps in the model that are consistent with this input. An action step is consistent with a SAP if the step is a generalization (abstraction) of the action in the SAP. For each consistent step, for example s, it checks whether the model can be modified to contain a direct path from C to s. By direct path, we mean a path without intervening action steps.

The constrained GUI editor is a program that captures the model defined in the Presentation Model and with the information given in the ON Metamodel, gives a preview of the designed interface. In order to protect the quality and good design defined in the Presentation Model, this editor is constrained by *parameters*. These parameters define which values can be defined and modified for each component in the interface and, therefore, they define a restriction on the editor that only allows the designer edit those predefined parameters. In this way, it is expected that the designer will not destroy the usability which has been generated by the system. Only the beautification operations which still preserve the usability to some extent are displayed in the contextual menu. For instance, no CIO may be deleted and no new widget or extra functionality can be defined in this editor. Each parameter is described by:

- The name of the parameter.
- The data type and the values of the parameters: Integer value in the range [6..22] .
- The widgets where this parameter may be applied: in this case, every widget that contains text.

Defining a parameter like "Label Font Size" for a table enables the designer to specifying which font size is the best one to show to a user in a table. Parameters go beyond simple CIOs since they gather high-level values that are consequently applied to one or many CIOs to complete their instantiation. When the designer modifies those parameters, a visual representation of each configuration is produced depending on the current value of each parameter, as a preview of what will be generated by OlivaNova. This preview is only for providing designers with some guidance and only approximates how the Final UI will look when the application is generated. We call it a *Generation Preview* as it provides a UI preview before its final code is generated. A prototype of the constrained GUI editor has been implemented so far in Java 1.5 with 15 beautification operations based on the SAP notation and the mechanism described.

Step 3: Generation of the final user interface. Once all the beautification operations have been applied by the designer, the CUI Model is completed and sent to the model compiler so as to perform the model-to-code transformation. This transformation transforms all the models defined in the process of designing the system. Traditionally, OlivaNova provides this transformation after defining the Object Model, the Dynamic Model, the Functional Model, and the Presentation Model. With this new Platform-Specific constrained GUI editor, the designer has the opportunity to define into more details how the application should look. With the modifications described previously, the resulting UI is shown in Fig. 7. Fig. 8 reproduces a UI resulting from other beautification operations also applied on the UI of Fig. 4: another visualization scheme is applied for the instances and the action bar is presented in an alternative way with icons.

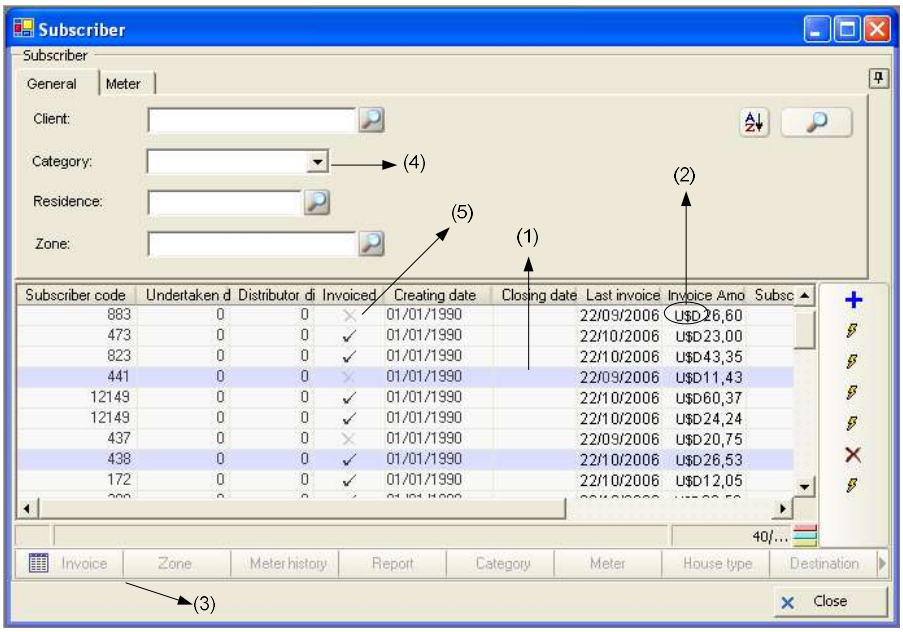

Fig. 7. The running example of Fig. 4. after applying the actions of the five beautification operations: (1) Specify(rowHighlightingType), (2) Convert (inputMetricUnit, outputMetricUnit), (3) Specify (buttonPresentationType), (4) Substitute (widgetType), (5) Specify (conditionalDisplay).

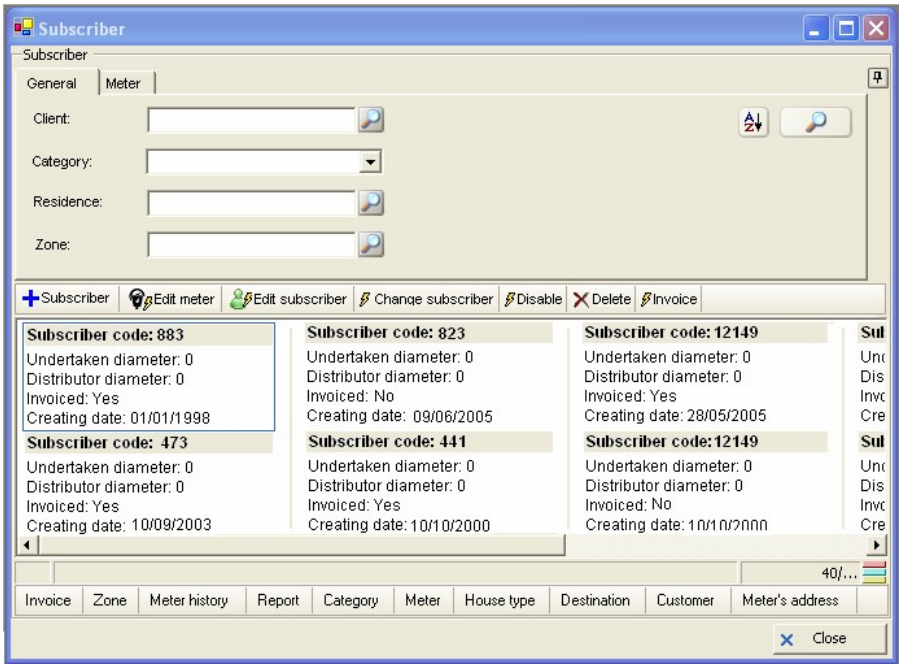

Fig. 8. The running example of Fig. 4 after applying other beautification operations.

4.5 The Parameters, Templates, and the User Interface Model

After presenting the three steps involved in the beautification process, we hereby explain how parameters could be gathered in a UI template, which are in turn organized into a hierarchy of templates. A template could be related to presentation or dialog. A presentation template is decomposed into style and layout templates (Fig. 11). A style template is decomposed into color scheme template and font template. As stated previously, the constrained GUI editor exploits these parameters. Each parameter represents an interface concept and is initially defined on a default value. As each interface concept is conceptualized in a parameter. With the template support, the designer can recurrently apply the same configuration to many projects or can have institutionalized styles for different customers. Here are some examples of such parameters at various levels of templates:

- *flowAlignment* (type=string, status=public, inherited=no, allowed values=optional): specifies how elements should flow in a flowBox: left, middle, right. For example, if flowAlignment="left", respectively, "middle", "right", the elements contained in such a flowBox will be laid out as represented in Fig. 9a, b, and c.
- *labelVerticalAlignment* (type=string, status=public, inherited=no, allowed values=mandatory): specifies how identification labels are their corresponding CIO are aligned vertically, as depicted in Fig. 10.

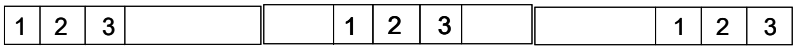

Fig. 9. The three types of flowAlignement in a flowBox (a,b,c).

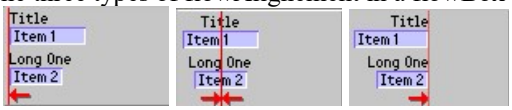

Fig. 10. The three types of labelVerticalAlignement.

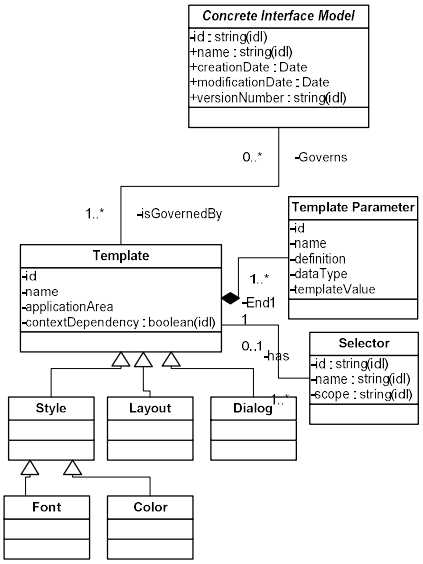

Fig. 11. How a template is attached to a CUI Model.

5 Conclusions and Future Work

This paper has examined in details the process of UI beautification, by analogy with the concept of "beautification" in the field of Computer Graphics. In the context of Model-Driven Engineering, this process turns out to be particularly useful. It consists in modifying a software artifact (a model of the system being built or the final application) so as to address those users' requirements which the given method could not satisfy by means of model-to-model and model-to-code transformations. When applied at the level of the automatically generated application, it supports round trip engineering in order to keep the models consistent with the tweaked code.

We have restricted the scope to UI beautification to only those beautification operations which preserve some usability. A review of existing methods and tools shows that there is still a lack of a sound solution to this thorny problem. Between the two extreme and commonly taken positions, resignation to the generated interface or unrestricted tweaking of the UI, we chose to adopt an intermediate and sensible approach that consists shifting the beautification process to a more abstract level. We have tackled this issue by following several steps:

1. Identification the most frequently demanded UI modifications.
2. Definition of a Concrete User Interface (CUI) model that allows refining UI appearance and behavior.
3. Definition of operations over the elements of the CUI model; these operations are formally defined as a SAP performed on partially instantiated CIO and will guarantee the fulfillment of the user's needs selected in step 1.
4. Construction of Constrained GUI Editor, a tool that allows editing the CUI model via the previously defined beautification operations.

The paper not only defines the process formally. It also presents a methodological approach to deal with the unsupported requirements of the Model-Driven Engineering that are recurrent in tools under the MDE approach. An example is provided along the paper to illustrate the approach. The first experience gained with this process and tool has been satisfactory and rewarding, although the Constrained GUI Editor supports a limited amount of operations: it significantly reduces the tweaking operations on the final generated code. The designer's effort to cover the unsupported requirements is reduced and the process has been well accepted. As future work, a more extended constrained GUI Editor will be developed and an empirical validation of the proposed modifications in OO-Method will be analyzed. This analysis will involve two aspects: the improvements on the whole production process after the adoption of the new tool and the benefits of its use in terms of final user satisfaction and UI usability.

Acknowledgements. This work is financed by DESTINO with reference TIN2004-03534, supported by the COST European Action n°294 MAUSE (Toward the Maturation of IT Usability Evaluation – www.cost294.org) and the SIMILAR network of excellence on multimodal interfaces (www.similar.cc). The authors would like to also thank Emilio Iborra, Ismael Torres, and José Maria Cubel for their input in this work.

References

1. Abrahão, S., Iborra, E., Vanderdonckt, J.: Usability Evaluation of User Interfaces Generated with a Model-Driven Architecture Tool. Chapter 2. In: Law, E., Hvannberg, E., Cockton, G. (eds): Maturing Usability: Quality in Software, Interaction and Value. HCI Series,

Springer-Verlag, Berlin (2007)
2. Antkiewicz, M.: Round-Trip Engineering of Framework-Based Software using Framework-Specific Modeling Languages. In: Proc. of ASE'2006
3. Clerckx, T., Luyten, K., Coninx, K.: The Mapping Problem Back and Forth: Customizing Dynamic Models while preserving Consistency. In: Proc. of TAMODIA'2004 (Prague, November 15-16, 2004). ACM Int. Series, Vol. 86. ACM Press, New York (2004) 33–42
4. da Silva, P.P.: User Interface Declarative Models and Development Environments: A Survey. In: Proc. of 7th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'2000 (Limerick, June 5-6, 2000). Lecture Notes in Computer Science, Vol. 1946. Springer-Verlag, Berlin (2000) 207–226
5. Genova V8.0, Esito AS, Lysaker (2006), http://www.genera.no/default.htm
6. Griffiths, T., Barclay, P.J., Paton, N.W., McKirdy, J., Kennedy, J.B., Gray, P.D., Cooper, R., Goble, C.A., da Silva, P.P.: Teallach: a Model-Based User Interface Development Environment for Object Databases. Interacting with Computers 14,1 (December 2001) 31–68
7. Hall, A., Chapman, R.: Correctness by Construction: Developing a Commercial Secure System. IEEE Software 19,1 (2002) 18–25
8. Igarashi, T., Matsuoka, S., Kawachiya, S., Tanaka, H.: Interactive Beautification: a Technique for Rapid Geometric Design. In: Proc. of the 10th Annual ACM Symposium on User Interface Software and Technology UIST'97. ACM Press, New York (1997) 105–114
9. JaxFront, XCentric Technology & Consulting GmbH, Zurich (2006), http://www.jaxfront.org/pages/
10. Model-Driven Architecture Guide, Version 1.0.1, Object Management Group (December 2006), http://www.omg.org/docs/omg/03-06-01.pdf
11. Molina, P.J., Meliá, S., Pastor, O.: JUST-UI: A User Interface Specification Model. In: Proc. of 4th Int. Conf. on Computer-Aided Design of User Interfaces CADUI'2002 (Valenciennes, May 2002). Kluwer Academic Press, Dordrecht (2002) 63–74
12. Myers, B., Hudson, S.E., Pausch, R.: Past, Present, and Future of User Interface Software Tools. ACM Trans. Computer-Human Interaction 7,1 (2000) 3–28
13. Nielsen, J.: A Virtual Protocol Model for Computer-Human Interaction. International Journal of Man-Machine Studies 24,3 (1986) 301–312
14. Nunes, N.J., Falcao e Cunha, J.: WISDOM - A UML-Based Architecture for Interactive Systems. In: Proc. of DSV-IS'2000 (Limerick, June 5-6, 2000). Lecture Notes in Computer Science, Vol. 1946. Springer-Verlag, Berlin (2000) 191–205
15. OlivaNova Software, Care Technologies, Denia (December 2006), http://www.care-t.com
16. Pastor, O., Gómez, J., Insfrán, E., Pelechano, V.: The OO-Method Approach for Information Systems Modeling: from Object-oriented Conceptual Modeling to Automated Programming. Information Systems 26,7 (2001) 507–534
17. Pavlidis, T., Van Wyk, C.J.V.: An Automatic Beautifier for Drawings and Illustrations. Computer Graphics 19,3 (July 1985) 225–234
18. Plimmer, B., Grundy, J.: Beautifying Sketching-based Design Tool Content: Issues and Experiences. In: Proc. of the 6th Australasian Conf. on User Interface AUIC'2005 (Darlinghurst, 2005). Australian Computer Society, Inc. (2005) 31–38
19. Puerta, A.R., A Model-Based Interface Development Environment. IEEE Software, 14,4 (1997) 40–47
20. Puerta, A.R., Eriksson, H., Gennari, J.H., Musen, M.A.: Beyond Data Models for Automated User Interface Generation. In Proc. of HCI'94 Glasgow, September 1994). Cambridge University Press, New York (2004) 353–366
21. Sendall, S., Küster, J.: Taming Model Round-Trip Engineering. In: Proc. of Workshop 'Best Practices for Model-Driven Software Development' MDSD'2004 (Vancouver, October 2004).
22. Vanderdonckt, J.: A MDA-Compliant Environment for Developing User Interfaces of Information Systems. In: Proc. of CAiSE'2005. LNCS, Vol. 3520. Springer-Verlag, 16–31.