Visual Design of User Interfaces by (De)composition

Sophie Lepreux[1,2], Jean Vanderdonckt[1], and Benjamin Michotte[1]

[1] IAG/ISYS, Université catholique de Louvain, Place des Doyens 1,
B–1348 Louvain-la-Neuve (Belgium)
[2] LAMIH – UMR CNRS 8530, Université de Valenciennes et du Hainaut-Cambrésis,
Le Mont-Houy, F-59313 Valenciennes Cedex 9 (France)
{lepreux, vanderdonckt, michotte}@isys.ucl.ac.be,
sophie.lepreux@univ-valenciennes.fr

Abstract. Most existing graphical user interfaces are usually designed for a fixed context of use, thus making them rather difficult to modify for other contexts of use, such as for other users, other platforms, and other environments. This paper addresses this problem by introducing a new visual design method for graphical users interfaces referred to as “visual design by (de)composition”. In this method, any individual or composite component of a graphical user interface is submitted to a series of operations for composing a new interface from existing components and for decomposing an existing one into smaller pieces that can be used in turn for another interface. For this purpose, any component of a user interface is described by specifications that are consistently written in a user interface description language that remains hidden to the designers’ eyes. We first define the composition and decomposition operations and individually exemplify them on some small examples. We then demonstrate how they can be used to visually design new interfaces for a real-world case study where variations of the context of use induce frequent recomposition of user interfaces. Finally, we describe how the operations are implemented in a dedicated interface builder supporting the aforementioned method.

1 Introduction

In most commercial interface builders (e.g., Macromedia DreamWeaver, Microsoft Visual Studio) and research interface editors (e.g., Glade, TrollTech), the predominant method for visually building a Graphical User Interface (GUI) consists of dragging widgets from a palette, dropping them on a working area, and editing their properties until the results are satisfactory. This method makes sense since the GUI is visual by nature and direct manipulation of constituting widgets remains natural, flexible, and modifiable [1,2]. However, when it comes to reusing parts or whole of an existing GUI to design another one, most interface builders force the designer to produce an incessant sequence of “copy/paste” operations, if supported, with little or no support for recomposing a new GUI from these elements. In particular, the designer should copy widgets one by one and perform relayouting operations (e.g., resizing, realignment, rearrangement) individually. This situation frequently occurs when an

G. Doherty and A. Blandford (Eds.): DSVIS 2006, LNCS 4323, pp. 157 – 170, 2007.
© Springer-Verlag Berlin Heidelberg 2007

existing GUI needs to be adapted for a new context of use, which the GUI was not designed or thought for. If the context of use is considered as the combination of a user (or a user stereotype) working on a given computing platform in a specific environment [3], any variation of one or many of these aspects may lead to a GUI redesign. In the case of multi-platform GUIs [4,5,6], it is impossible to copy/paste GUI elements from one interface builder to another one, unless the interface builder is itself multi-platform. Even in that case, little or no support is provided for reforming a new GUI from fragments coming from existing GUIs. In the case of multi-language GUIs, existing tools prevent designers from just translating the resources in one language and obtain a new GUI for another language.

On the method side, reusability of existing GUIs is often promoted as a desirable method for ensuring consistency, reducing development effort, fitting a particular GUI to the purpose of a given task. In particular, users frequently report that they need to constantly switch from one application to another to fulfill a given task when it was not possible to re-assemble existing components of existing GUIs to form a new one. Again, little or no methodological guidance exists in current development methods to help designers reusing parts or whole of their design to initiate a new development process.

This paper addresses the lack of support for reusing existing developments of GUIs by introducing a visual design method based on three concepts: *decomposition* disassembles an existing GUIs into individual or composite elements that can be further reused for other designs; *composition* assembles individual and composite elements to form a new GUI that fits the purpose of a given task; *recomposition* performs a suite of decompositions and compositions to support re-design of existing GUIs for new contexts of use.

Various simplified forms of decomposition and composition already exist as reported in Section 2 devoted to the state of the art, but we are not aware of any integrated method that is intended to support reusability at a high level of design that does not force people to constantly apply physical and lexical operations. Section 3 presents a reference framework that will be extensively used in the rest of the paper: any GUI will be described in the terms defined by this framework to maintain editable specifications of the GUI of interest. Section 4 defines a series of operators for decomposition and composition: each operator is logically defined, explained, motivated and exemplified with a simple example. Section 5 validates the method by applying these operators on a real-world case study in an interface builder implemented for this purpose. Section 6 concludes the paper by reporting on the main advantages and shortcomings of the work and suggesting some avenues for future work.

2 Related Work

Due to the nature of our problem, the following state of the art is decomposed into two categories: decomposition and composition.

Decomposition. The Covigo library (http://www.covigo.com) supports a simple form of decomposition called *pagination*, where a web page is decomposed into smaller

pieces to be used on a smaller screen: special tags are inserted in a HTML web page at run-time to decompose it into smaller pieces. Simple heuristics such as breaking every fifth <tr> or breaking by size are used. Here, the pagination is fully automated, with the attendant risk that it does not break the UI logically. On the other hand, RIML [7] supports manual pagination, thus leaving the decomposition quality under the designer's control and responsibility: it defines additional mark-up for specifying the layout and pagination capabilities of web pages that are then rendered through a dedicated Web adaptation engine. Watters and Zhang [8] segment HTML forms into a sequence of smaller forms, using partition indicators such as horizontal lines, nested lists and tables. Complex layout relationships (e.g., use of tables for layout purpose) probably constitute a bottleneck for such approaches.

To overcome the language restriction, another group of approaches relies on a generic GUI description in a User Interface Description Language (UIDL) that is at a higher level than the markup and programming languages. Major UIDLs such as UIML [4], SunML [9], XIML [10] support decomposition as their UI description can be split into logically related chunks. Again, the designer is responsible for this operation without any support. Göbel *et al.* [6] describe web-based dialogs in a device-independent way through "DLL dialog", which is a composition of containers and elements. Containers whose elements must appear together are called atomic. Elements are assigned weights indicating their resource requirements in terms of memory and screen size. Fragments with similar weights are generated, while respecting the integrity of atomic containers. Navigation elements are added to permit navigation between dialog fragments. No indication is given on how weights should be assigned to leaf elements, which is a difficult task, especially for multiplatform rendering. Ye & Herbert [11] apply similar heuristics for decomposing a XUL UI description by relying on the hierarchy of widgets and containers, while respecting the value of a 'breakable' attribute attached to each component, which has to be explicitly provided by the designer. PIMA [12] also relies on a UIDL, which is converted into multiple device-specific representations, including a decomposition process. Like other approaches, PIMA's algorithm uses grouping constraints as well as information on size constraints. MORALE [13] is a suite of tools for assembling GUIs with their associated definitions, but all (de)composition operations are restricted to cut/-copy/paste primitives.

While the aforementioned decomposition methods mostly work on a hierarchy of GUI widgets, ROAM [5] consider a tree structure combining a task hierarchy and a layout structure. The tree nodes are annotated as splittable or unsplittable depending on the decomposition possibilities. ROAM's does not really decomposes an existing GUI as it merely moves the extra widgets that do not fit onto a new GUI. Graceful degradation [14] addresses the decomposition problem, but only for the purpose of obtaining GUIs for more constrained platforms, one dimension of the context of use, but not the only one. AUIT [15] automatically generate code generation for JSP and servlet implementations depending on parameters from any platform/user/task combination. A set of XSLT transformation scripts convert the XML-encoded logical screen design into several GUIs.

Composition. Several environments attempt to compose a new GUI by assembling fragments coming from the same or different GUIs. They only differ by the level

where the composition is performed. Scalable Fabric [16] is a smart environment where documents associated with interactive applications are grouped depending on their semantic relationships in the user's task. Haystack [17] is a platform for personalizing information spaces and applications for a particular user depending on her tasks. WinCuts [18] recompose GUIs by duplicating parts or whole of a GUI into a new one that corresponds to the users' task. Similarly, Composable UIs [19] define viewports on GUIs to form a new UI by putting the viewports side by side. A detachable UI [20] is a GUI portion that can migrate from one computing platform to another one with re-assembling on the target.

In summary, we observed that major approaches for (de)composition are often language- or platform-dependent to some extend, do not identify independent high-level design primitives for recomposition, are usually supported at the physical level (e.g., as in [18,19,20]) or the application level without any flexibility, are typically considering decomposition merely for screen constraints or multi-platform support. Little or no methodological guidance is provided for this purpose, although it is identified as a major design activity [1,2]. We are not aware of any research that provides a systematic set of (de)composition primitives applicable to any GUI.

3 Reference Framework

To allow high-level design operations on any GUI, we should rely on a high level description of the initial user interface. This description will be expressed in the UsiXML (User Interface eXtensible Markup Language – http://www.usixml.org [21]) UIDL. The principles set out below are, however, generally applicable to any UIDL such as UIML [4], SunML [9] or XIML [10]. UsiXML is structured according to the four abstraction levels of the 'CAMELEON reference framework' [3] for multi-target UIs (Fig. 1).

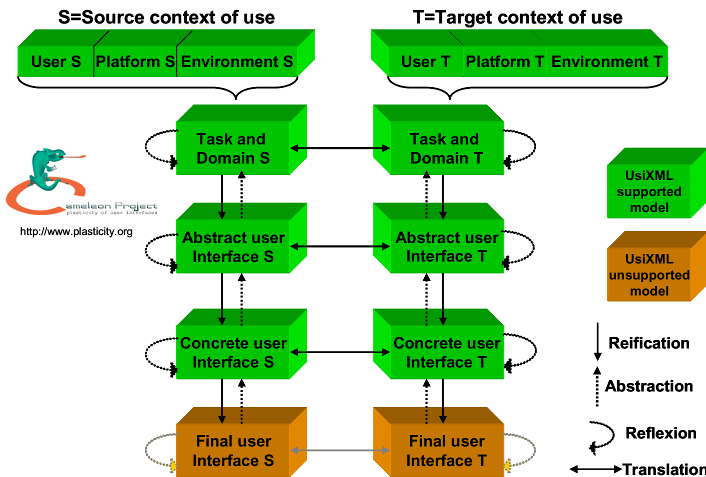

Fig. 1. The four abstraction levels used in the framework

A *Final User Interface* (FUI) refers to an actual UI rendered either by interpretation (e.g., HTML) or by code compilation (e.g., Java). A *Concrete User Interface* (CUI) abstracts a FUI into a description independent of any programming or markup language in terms of Concrete Interaction Objects, layout, navigation, and behavior. An *Abstract User Interface* (AUI) abstracts a CUI into a definition that is independent of any interaction modality (such as graphical, vocal or tactile). An AUI is populated by *abstract components* and *abstract containers*. Abstract components are composed of facets describing the type of interactive tasks they are able to support (i.e., input, output, control, navigation). The *Tasks & Concepts* level describes the interactive system specifications in terms of the user tasks to be carried out and the domain objects of these tasks. As (de)composition operations will be defined independently of any context of use (including the computing platform), the CUI level is the best candidate for a formal definition. Therefore, this level is more detailed in the subsequent paragraphs.

A CUI may be obtained by forward engineering from the T&C level, the AUI level or directly. A CUI is assumed to be described without any reference to any particular computing platform or toolkit of that platform [21]. For this purpose, a CUI model consists of a hierarchical decomposition of CIOs. A *Concrete Interaction Object* (CIO) is defined as any UI entity that users can perceive such as text, image, animation and/or manipulate such as a push button, a list box, or a check box. A CIO is characterized by attributes such as, but not limited to [21]: *id*, *name*, *icon*, *content*, *defaultContent*, *defaultValue*.

4 (De)composition Operations

In this section, (de)composition operations are first defined based on the UsiXML concepts of a Concrete User Interface. Since the UI is represented in UsiXML terms and since it is a XML-compliant language, operations could be defined thanks to tree algebra, with which operations could be logically defined on the XML tree and directly performed. El-bekaï et al. defined a set of operators to comparison (similarity, equivalence and subsumption) and others operators adapted to database [22]. We adapt their notation presented in the next part to decomposition and composition goal in the second part. Then, an implementation is described of a tool that supports a method based on these operations.

4.1 Relation Between UsiXML, XML and Tree Algebra

Since each GUI is described in UsiXML terms as a Concrete User Interface as indicated in the previous section, each GUI is internally represented as a tree of XML elements.

Thus, the correspondence proposed by [22] gives that the basic elements of a UsiXML UI, i.e. a XML tree, could be defined logically:

- XML document → Tree (T)
- Element → Root node (R), parent (P), child (C) node
- Leaf → child (C) node, atomic (A) values

Fig. 2 shows the relationships between a GUI (top left), its UsiXML specifications (top right) and its internal structure as a XML tree in order to perform the operations.

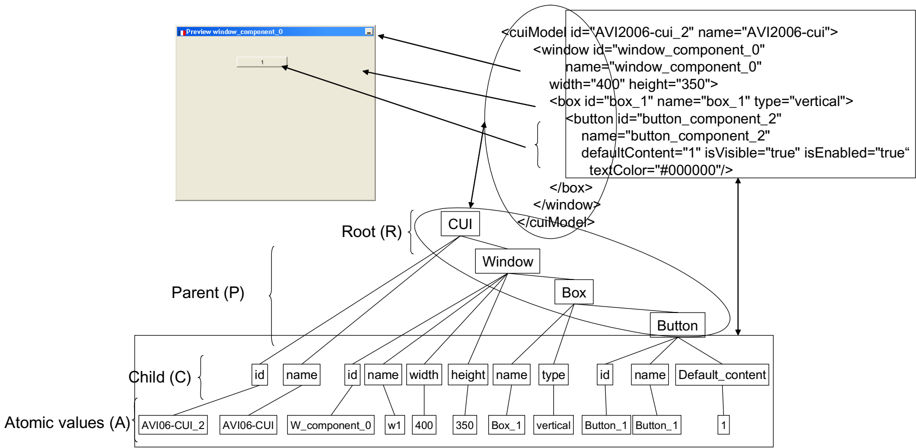

Fig. 2. An UI, its UsiXML and its tree representation

4.2 Presentation of the Operators

A first part presents a few operators associated to the decomposition, whereas a second part presents a few operators linked with the composition.

4.2.1 Operators Supporting Decomposition

This part defines two basic operators working on the internal structure of the UsiXML specifications. Other operators such as *Cut*, *Projection*, *and Complementary* are defined with the same principle but are not presented here.

Selection

$$\sigma(T)(E) \rightarrow T$$

Pre : let T a tree and E an Expression

$$\text{Post : } \sigma(T_1)(E) = T_2 \quad \begin{array}{l} T_2(R) = T_1(R) \middle| E(R) = true \\ T_2(P) = T_1(P) \middle| E(P) = true \\ T_2(C) = T_1(C) \middle| E(C) = true \end{array} \qquad (1)$$

The *Selection* operator which works upon tree and an expression is defined in (1). This operator aims at keeping the node which corresponds to the expression. For example, Fig. 3 apply the expression E={output} to an UI and its result. The resulting UI is the same as in the input UI with only the “output” elements.

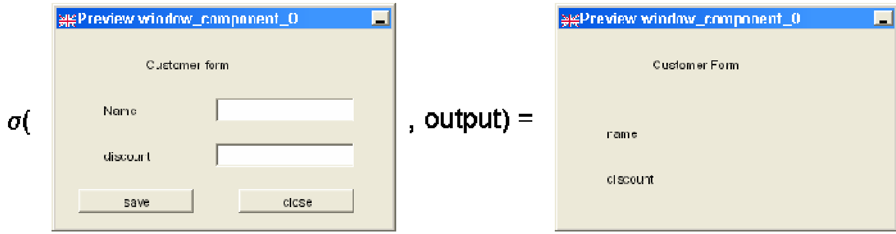

Fig. 3. Example of the selection operator

Intersection

$T \cap T \rightarrow T$

Pre : let $T_1, T_2 trees$

$$\text{Post : } T_1 \cap T_2 = T_3 \quad \begin{array}{l} T_3(R) = T_1(R) \\ T_3(P) = T_1(P) + T_2(P) - 2(T_1T_2(P)) \\ T_3(C) = T_1(C) + T_2(C) - 2(T_1T_2(C)) \end{array} \tag{2}$$

The intersection operator is defined in (2). It is a binary operator; it takes two trees as input. The output is new XML data containing elements, root node, parent nodes and child nodes which are in one of two trees data model. The intersection operator applied on two similar interfaces will give the interface shown in Fig. 4.

In this algorithm, the different elements are compared. We have stated that two elements are identical or similar if they have the same type (i.e. button), the same name in one language (i.e. save) and all the required attributes. As the size and the color are optional arguments, we consider that they can be different. In this case the resulting button keeps only the options which are identical in the two tested button.

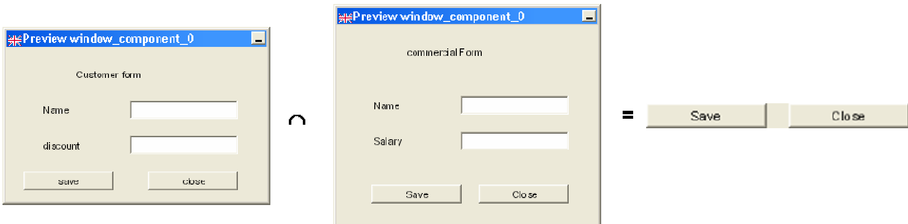

Fig. 4. Example of the intersection operator

4.2.2 Operators Supporting Composition

This part defines two basic operators on the internal structure of the UsiXML specifications. Other operators, such as "Difference" operator are defined with the same principle but are not presented here. This difference operator takes two trees as input and gives a tree as output. The output tree is the very first input tree without the elements which are included in the two input trees.

Fusion

$$T + T \rightarrow T$$

$$\text{Pre : let } T_1, T_2 \textit{trees}$$

$$\text{Post : } T_1 + T_2 = T_3 \quad \begin{array}{l} T_3(R) = T_1(R) \\ T_3(P) = T_1(P) + T_2(P) \\ T_3(C) = T_1(C) + T_2(C) \end{array} \tag{3}$$

The fusion operator is defined in (3). It is a binary operator; it takes two trees as input. The output is new XML data containing elements, root node, parent nodes and child nodes which are in the two trees data model. The fusion operator applied on two interfaces, following the algorithm 1, will give the interface shown in Fig. 5.

```
%algorithm1: The two trees T1 and T2 are merge at the %level
R+1 to form the T3 window.

IF (direction = vertical)
Then        Add box (vertical B')
            %Modify the window size:
              T3.height = T1.height + T2.height
              T3.width = T1.width

IF (direction = horizontal)
Then        Add box (horizontal B').
            %Modify the window size:
            T3.height = T1.height
            T3.width = T1.width + T2.width

Add T1(R+1) in box B', Add T2(R+1) in box B'.
```

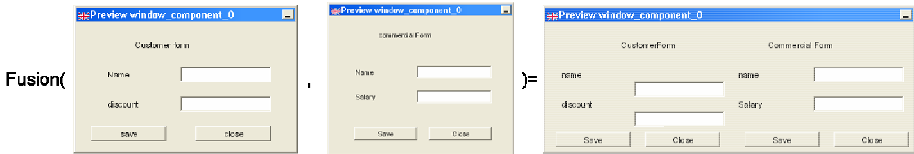

Fig. 5. Example of the fusion operator

Union

$$T \cup T \rightarrow T$$

$$\text{Pre : let } T_1, T_2 \textit{trees}$$

$$\text{Post : } T_1 \cup T_2 = T_3 \quad \begin{array}{l} T_3(R) = T_1(R) \\ T_3(P) = T_1(P) + T_2(P) - (T_1 T_2(P)) \\ T_3(C) = T_1(C) + T_2(C) - (T_1 T_2(C)) \end{array} \tag{4}$$

The output of the union operator consists of new XML data containing elements, root node, parent nodes and child nodes in the two input trees data model without the duplication of any elements such root nodes, parent nodes and child nodes. The union is disjoint: duplicates are purged. This operator is defined in (4). To illustrate this operator, one example of result is shown in Fig. 6. However, if the two “name”

elements are considered as identical then the result could be different. Since the duplicates are purged, the Area text associated to the name which is present in the same structure and content in the two input interfaces will be purged in the output user interface. The result is presented Fig. 7(a). The Union operator does not take into account the place of the element in the interfaces so the result can be as shown in Fig. 7(b). In this case, we could consider some of the operators as '*presentation-independent*', that is they are not sensible to physical aspects of the GUI such as position, size, arrangement, colors, fonts, style. However, if such a need arises, it is still possible to incorporate these constraints as conditions.

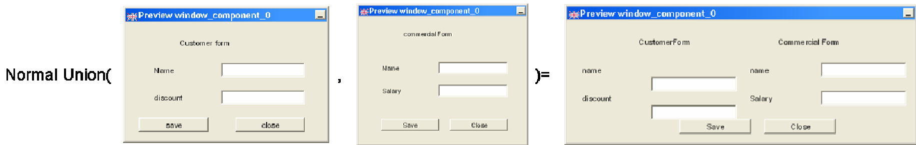

Fig. 6. An example of expected user interfaces from union of the two user interfaces

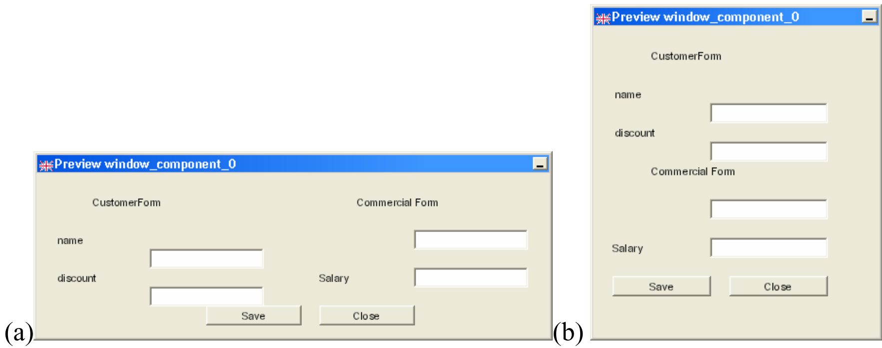

Fig. 7. The other results expected from the Union operator

4.3 Implementation

Some of the above operations have been directly implemented in GrafiXML, a graphical interface builder that automatically generates UsiXML specifications as opposed to final code for other builders. GrafiXML has been implemented in Java 5.0 and today consists of more than 90,000 lines of Java code. It can be freely downloaded from http://www.usixml.org as it is an open source project regulated by Apache 2.0 open licence and available on SourceForge. GrafiXML is able to automatically generate code of a UI specified in UsiXML into (X)HTML or Java. For the purpose of the examples below, we will rely on the Java automated code generation.

5 Case Studies

The operators defined here above can be used in two cases. At the *design time,* they can be used by the designers to create the user interfaces. For example, the user

interfaces which are built to one application or to a set of applications of the firm have to respect a graphic charter. With the operator, the designer can reuse some of the elements of the user interfaces. This is already illustrated by the examples associated to the operators. This case is not presented here but is presented in [23]. The second case of use is at *run time*. It is integrated in the reuse issue which has introduced the component idea. The first issue in this domain is the composition of the components. If we consider the business component as a component with user interfaces, one issue in the domain of HCI is to compose the user interfaces of the business components. The using of business components and of their user interfaces brings to the user interfaces composition issue. If we consider that the user interfaces are specified with UsiXML, the union operator is particularly interesting for the composition.

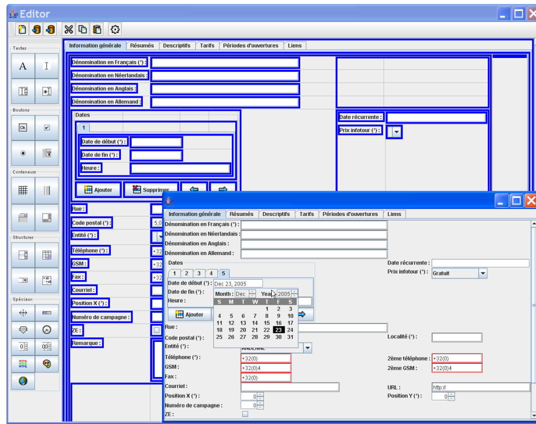

Fig. 8. Initial UI for a tourist application

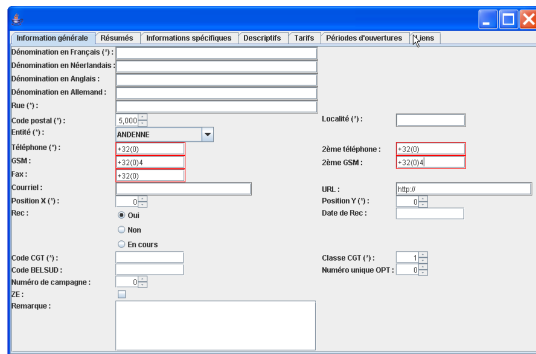

Fig. 9. Initial UI for an event management application

Let us now consider another case study taken in the domain of tourism. In this domain, it happens frequently that some parts of the same information should be reproduced in different UI for different events (e.g., hotel information, tourist trip

including hotel booking, booking a hotel, etc.). Fig. 8 reproduces a screenshot of a Concrete User Interface edited in the editor and its preview in Java (obtained by Java automated code generation). This view is particularly appreciated by designers and developers (and even end users) as it combines the design view and the final view, which is pretty close to the UI as the end user will see. In order to define a precise layout, a matrix of lines and guides could be defined to align objects in lines and columns.

Fig. 9 reproduces another UI for an event management application, also taken in the same domain. The two UIs only differ from a few fields, here the dates of the events in Fig. 8 and the comment in Fig. 9. Therefore, if we want to identify the common part of these two UIs, the intersection operator performs the operation, as defined previously, to identify common parts of both trees and then rebuilds a new tree with the identified elements. This operator re-generates new UsiXML specifications. This intersection is reproduced in Fig. 10. Note in Fig. 10 that the designer did not need to do anything: all common elements were identified, a new layout was produced so as to mimic the initial one and all objects have been laid out and aligned to preserve the initial constraints. Therefore, there was no need to re-position, re-align, or re-arrange the widgets.

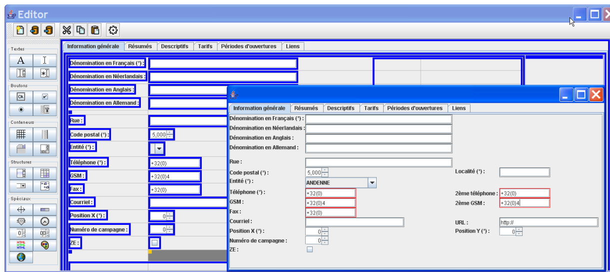

Fig. 10. Intersection of UIs found in Fig. 10 and 11

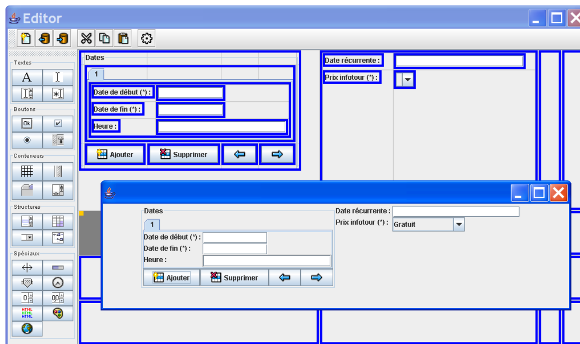

Fig. 11. Difference between Fig. 10 and 11

Similarly, Fig. 11 illustrates the application of the difference operator, in this case Fig. 8 – Fig. 9. Therefore, Fig. 11 only contains those widgets of Fig. 8 that are not

present in Fig. 9. Again, these widgets are identified and re-laid out so as to form an entirely new UI that is ready to test with the end user. Note that it is even possible to define new operators or composition of individual operators. As soon as this intersection is identified, it is possible to submit again this intersection to any other operation (here, a sequence of global copy/paste) so as to define a new operator by composition. The composition of UI operators is inspired from the macro-commands from the domain of command languages where several individual commands applied to some objects could be grouped together into a macro-command. In this way, the designer is able to define her/his own combination of operators and repetitions on demand. At any time, each operator works on the underlying UI model expressed in UsiXML. Without this characteristic, it would have been almost impossible to program these operators in a classical interface builder where all widgets are physically defined. Instead, they are here logically defined, thus allowing logical operations. At any time, the code of the final UI can be produced.

The last example showed Fig. 12 concerns the (vertical) union operator. This operator allows composing two interfaces without repetition. In this case, two parts of information are repeated, the designation and the piece of information. These common parts are viewed in the Fig. 10 which presents the intersection. So these elements are not duplicated by the union operator. All the elements are placed with the respect of the initials UIs. In this case, if the fusion operator is used, then all the elements of each interface are laid-out. The common elements will be presented twice.

Fig. 12. Union of Fig. 8 and 9

6 Conclusion and Future Work

We have described logical operators with which it is possible to manipulate UI portions or whole at a large grain than simply with the widget level that is the most common technique found in classical interface builders. Therefore, instead of manipulating one widget at time for designing a UI (an activity that is time consuming and tedious), it is possible to manipulate UI fragments as such. Then, and only then, cut/copy/paste operations could be applied. The main difference is that these operations are logically applied as opposed to a physical application where all individual widgets need to be re-positioned, re-sized, and re-arranged. Re-positioning,

re-sizing, and re-arrangement are the most frequently executed operations in interface builders, consequently to redesigning a UI or reusing a previously designed UI. This situation also often occurs when UI templates are used.

The operators which have been introduced are logically defined based on the tree algebra and adapted to the domain of user interfaces. These operators were described with an example and more developed in the case study. Using of the operators from the tree algebra is appropriate because the user interfaces are specified in UsiXML and because the XML documents can be processed like trees.

Acknowledgments

We gratefully thank the support from the SIMILAR network of excellence (The European research taskforce creating human-machine interfaces SIMILAR to human-human communication), supported by the 6th Framework Program of the European Commission, under contract FP6-IST1-2003-507609 (http://www.similar.cc). The authors thank also the Nord-Pas de Calais regional authority (Projects MIAOU and EUCUE) and the FEDER (Fonds Européen de Développement Régional, European Fund for Regional Development) for supporting a part of this work.

References

1. Brown, J.: Exploring Human-Computer Interaction and Software Engineering Methodologies for Creation of Interactive Software. SIGCHI Bulletin **29**, 1 (1997) 32–35
2. Morch, A.: Tailoring tools for system development. Journal of End User Computing **10**, 2 (1998) 22–29
3. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A Unifying Reference Framework for Multi-Target User Interfaces. Interacting with Computer **15**, 3 (2003) 289–308
4. Ali M.F., Pérez-Quiñones M.A., Abrams M.: Building Multi-Platform User Interfaces with UIML. In: Seffah, A., Javahery, H. (eds.): Multiple User Interfaces: Engineering and Application Framework. John Wiley, Chichester (2004) 95–118
5. Chu, H., Song, H., Wong, C., Kurakake, S., Katagiri, M.: Roam, a Seamless Application Framework. Journal of System and Software **69**, 3 (2004) 209–226
6. Göbel, S., Buchholz, S., Ziegert, T., Schill, A.: Device Independent Representation of Web-based Dialogs and Contents. In Proc. of IEEE Youth Forum in Computer Science and Engineering YUFORIC´01 (Valencia, November 2001). IEEE Computer Society Press, Los Alamitos (2001)
7. Spriestersbach, A., Ziegert, T., Grassel, G., Wasmund, M., Dermler, G.: Flexible Pagination and Layouting for Device Independent Authoring. In Proc. of WWW'2003 Workshop on Emerging Applications for Wireless and Mobile Access (2003)
8. Watters, C., Zhang, R.: PDA Access to Internet Content: Focus on Forms. In Proc. of the 36th Annual Hawaii Int. Conf. on System Sciences HICSS'03 (Big Island, January 2003). IEEE Computer Society Press, Los Alamitos (2003) 105–113
9. Dery-Pinna, A.-M., Fierstone, J., Picard, E.: Component Model and Programming: a First Step to Manage Human-Computer Interaction Adaptation. In Proc. of 5th Int. Symposium on Human-Computer Interaction with Mobile Devices and Services MobileHCI'2003 (Udine, September 8-11, 2003). Lecture Notes in Computer Science, Vol. 2795. Springer-Verlag, Berlin (2003) 456–460

10. Eisenstein, J., Vanderdonckt, J., Puerta, A.: Model-Based User-Interface Development Techniques for Mobile Computing. In Lester J. (ed.): Proc. of 5th ACM Int. Conf. on Intelligent User Interfaces IUI'2001 (Santa Fe, January 14-17, 2001). ACM Press, New York (2001) 69–76
11. Ye, J., Herbert, J.: User Interface Tailoring for Mobile Computing Devices. In Proc. of 8th ERCIM Workshop « User Interfaces for All » UI4All'04 (Vienna, June 28-29, 2004). Lecture Notes in Computer Science, Vol. 3196, Springer-Verlag, Berlin (2004) 175–184
12. Banavar, G., Bergman, L.D., Gaeremynck, Y., Soroker, D., Sussman, J.: Tooling and System Support for Authoring Multi-device applications. Journal of Systems and Software **69**, 3 (2004) 227–242
13. Rugaber, S.: A Tool Suite for Evolving Legacy Software. In Proc. of IEEE Int. Conf. on Software Maintenance ICSM'99 (Oxford, 30 August-3 Sep. 1999). IEEE Comp. Society Press, Los Alamitos (1999) 33–39
14. Florins, M., Vanderdonckt, J.: Graceful Degradation of User Interfaces as a Design Method for Multiplatform Systems. In Proc. of Int. Conf. on Intelligent User Interfaces IUI'04 (Funchal, January 13-16, 2004). ACM Press, New York (2004) 140–147
15. Grundy, J.C., Hosking, J.G.: Developing Adaptable User Interfaces for Component-based Systems. Interacting with Computers **14**, 3 (2001) 175–194
16. Robertson, G., Horvitz, E., Czerwinski, M., Baudisch, P., Hutchings, D., Meyers, B., Robbins, D., Smith, G.: Scalable Fabric: Flexible Task Management. In Proc. of ACM Conf. on Advanced Visual Interfaces AVI'2004 (Gallipoli, May 25-28, 2004). ACM Press, New York, (2004) 85–89
17. Quan, D., Huynh, D., Karger, D.R.: Haystack: A Platform for Authoring End User Semantic Web Applications. In Proc. of International Semantic Web Conference (2003)
18. Tan, D.S., Meyers, B., Czerwinski, M.: WinCuts: Manipulating Arbitrary Window Regions for more Effective Use of Screen Space. In Proc. of ACM Conf. on Human Aspects in Computing Systems CHI'2004 (Vienna, April 2004). ACM Press, New York (2004) 1525-1528
19. Leventhal, E., Grubis, A.: Composable User Interfaces. The MITRE Corporation, Bedford USA (2004)
20. Grolaux, D., Vanderdonckt, J., Van Roy, P.: Attach me, Detach me, Assemble me like You Work. In Costabile, M.-F., Paternò, F. (eds.): Proc. of 10th IFIP TC 13 Int. Conf. on Human-Computer Interaction INTERACT'2005 (Rome, September 12-16, 2005), Lecture Notes in Computer Science, Vol. 3585, Springer-Verlag, Berlin (2005) 198–212
21. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Lopez, V.: USIXML: a Language Supporting Multi-Path Development of User Interfaces. In Proc. of 9th IFIP Working Conf. on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems EHCI-DSV-IS'2004 (Hamburg, July 11-13, 2004). Lecture Notes in Computer Science, Vol. 3425. Springer-Verlag, Berlin (2005) 200–220
22. El Bekai, A., Nick Rossiter, B.: A Tree Based Algebra Framework for XML Data Systems. In Proc. of the 7th Int. Conf. on Enterprise Information Systems ICEIS'2005 (Miami, May 25-28, 2005) (2005) 305–312
23. Lepreux, S., Vanderdonckt, J.: Toward a support of the user interfaces design using composition rules. In Calvary, G., Pribeanu, C., Santucci, G., Vanderdonckt, J. (eds): Proc. of the 6th International Conference on Computer-Aided Design of User Interfaces (CADUI'2006). (Bucharest, Romania, June 5-8, 2006) Chapter 19, Springer-Verlag, Berlin, (2006)