Splitting Rules for Graceful Degradation of User Interfaces

Murielle Florins
IAG/ISYS
Université catholique de Louvain
Place des Doyens, 1
B-1348 Louvain-la-Neuve, Belgium
florins@isys.ucl.ac.be

Francisco Montero Simarro
Escuela Politécnica Superior de Albacete
Universidad de Castilla-La Mancha
Campus Universitario s/n
02071 Albacete, Spain
fmontero@info-ab.uclm.es

Jean Vanderdonckt, Benjamin Michotte
IAG/ISYS
Université catholique de Louvain
Place des Doyens, 1
B-1348 Louvain-la-Neuve, Belgium
{vanderdonckt,michotte} @isys.ucl.ac.be

ABSTRACT

This paper presents a series of new algorithms for paginating interaction spaces (i.e.; windows, dialog boxes, web pages…) based on a multi-layer specification in a user interface description language. We first describe how an interaction space can be split using information from the presentation layer (Concrete User Interface). We then demonstrate how information from higher levels of abstraction (Abstract User Interface, Task model) can be used to produce a pagination that is more meaningful from the task's viewpoint than other techniques. The pagination relies on a set of explicit splitting rules that can be applied as the first step in a graceful degradation. These splitting rules are implemented as an interface builder plug-in which automatically generates code under the designer's control.

Categories and Subject Descriptors

D.2.2 [**Design Tools and techniques**]: User Interfaces

General Terms

Design, Human Factors

Keywords

Design, graceful degradation, multiplatform systems, pagination, splitting rules

1. INTRODUCTION

The task of designing user interfaces for multiple platforms simultaneously is difficult and time consuming: user interfaces have always represented an important part of the software development and maintenance effort, even for traditional applications designed with a single target platform in mind. The designer's task is made more difficult when several devices like desktops, mobile phones or PDA's have to be taken into account, for two reasons:

- *Usability*: producing usable user interfaces for each platform-specific version of a system is difficult, due to differences between the capabilities of the different devices and softwares

- *Cross-platform consistency*: users expect to be able to employ their knowledge of a given version of the system when using the same service on another platform, thus the different versions should not be too different

A lot of design methods and adaptation techniques that produce multiple UIs simultaneously for different platforms have been investigated (see, inter alia, [1, 2, 3, 6, 7, 10, 18, 20]). Our approach to the design of multiplatform user interfaces, known as 'graceful degradation' of UIs [9], consists in specifying one source interface, designed for the least constrained platform, and to apply transformation rules to this source interface in order to produce specific interfaces targeted to more constrained platforms (which is sometimes referred to in the literature as a "single authoring" method [7][8]). The transformation process can be decomposed into five steps ranging from the highest level of granularity to the lowest one:

1. *Splitting rules* are applied to split the initial UI into chunks that are logically or semantically related;
2. *Interactor and image transformation rules* (e.g., widget substitution) are applied to transform the widgets of the initial UI into smaller widgets, while supporting the same functionalities;
3. *Moving rules* are applied to reshuffle widgets to obtain a UI that consumes less screen space;
4. *Resizing rules* are applied to shrink widgets and images or to re-align them after they have been moved;
5. *Removal rules* are applied to delete unnecessary or less useful widgets while preserving the main purpose of the initial UI.

Pagination is perhaps the most difficult and significant step of the whole graceful degradation process. Splitting generates important changes into the very structure of the UI, has an important influence on the quality of the final results, and is appreciated by users that consider it as one of the most useful GD rules [12].This paper presents a full implementation of splitting rules (also called pagination rules) and their underlying concepts and related algorithms. Automatic pagination is a complex problem that has been partially addressed in the existing studies that are described in Section 2. Section 3 presents a reference framework that will be extensively used in the rest of the paper. Splitting will be examined at two levels of abstraction: concrete UI (Section 4) and abstract UI (Section 5) with information contained in the task model. Section 6 will demonstrate how software developed for graceful degradation supports splitting rules. Section 7 concludes the paper by reporting on the main advantages and shortcomings of the work and suggesting some avenues for future work.

2. DISCUSSION OF RELATED WORK

2.1 State of the art

Pagination is an important adaptation technique which minimizes the need for scrolling on small displays. Excessive depth of vertical scrolling has been demonstrated to remain an obtrusive usability problem [10]. Pagination for redistributing web pages (e.g., forms, text, images) among several pages has been widely researched, but less attention has been paid to graphical user interfaces (GUIs) as a whole general problem for any platform.

The Covigo library of special tags for HTML [15] implements pagination of web pages at run-time, using simple heuristics such as breaking every fifth <tr> or breaking by size. RIML [19] defines additional mark-up for specifying the layout and pagination capabilities of web pages. The new mark-up delimitates sections, which are the UI building blocks, and associated containers. Each container can be specified as a paginating container. After pagination, the sections that belong to a paginating container can be distributed over different pages, while the content of non-paginating containers will be repeated on each resulting page.

Unlike the two first approaches, Watters and Zhang's [21] approach can process any pre-existing HTML form, not only newly created page specifications. Their algorithm segments forms into a sequence of smaller forms, using partition indicators such as horizontal lines, nested lists and tables. Of course, grouping directives induced from the 'partition indicators' within the code are less accurate than they would have been in an explicit specification. Complex layout relationships (e.g. use of tables for layout purpose) probably constitute a bottleneck for such approaches. This algorithm is restricted to HTML forms only. Splitting pre-existing Web pages is also the concern of Chen & al. [6]. Their technique consists of three steps. First, the high-level content blocks typical in current Web site designs (header, footer, sidebars, and body) are identified by analysing the position and dimension of the nodes in the HTML DOM tree. Afterwards, each block can be further partitioned by detecting "explicit separators" i.e., tags such as <HR>, <TABLE> or <DIV>. The last step consist in finding "implicit separators" i.e., blank spaces. Once the page split into fragments, an index page linking to each subpage is produced by generating a thumbnail image of the original Web page, with the appropriate hyperlinks.

To overcome the language restriction, another group of approaches relies on a generic UI description in a user interface description language (UIDL) that is at a higher level than the markup and programming languages. Göbel et al. use an XML-based dialog description language (DDL) especially aimed at describing web-based dialogs in a device-independent way. A DDL dialog is composed of containers and other elements such as controls and images. 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 [22] apply similar heuristics to an XUL UI description by exploiting 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 [1] also relies on a UIDL, which is converted into multiple device-specific representations. This conversion includes a splitting process. Like other approaches, PIMA's algorithm uses grouping constraints as well as information on size constraints. PIMA also takes navigation into account and the possibility of applying distinct navigation policies between screens resulting from a splitting process.

While the fragmentation methods enumerated so far mostly work on a hierarchy of interface components (i.e. on elements related to the *presentation* of the UI), the splitting algorithm of the Roam system [5] takes as its input a tree structure combining a task model, which is only a hierarchy of tasks without temporal constraints, and a layout structure. The nodes of the tree can be annotated as splittable or unsplittable. Roam's algorithm does not attempt to find the best place to split, but merely places the extra widgets that do not fit in a page onto a new page (no scrolling is allowed). Navigation between the new pages is also generated, although without a lot of flexibility.

2.2 Requirements

To overcome the shortcomings identified in existing pagination techniques, our splitting approach should satisfy the conditions:

- *Be language-independent*, not tied to a given technology: we do not want to write a separate set of algorithms for HTML pages [6, 15, 19, 21] and another one for AWT/Swing windows.
- *Not introduce any additional construct*, no need for yet another mark-up language specially designed to support pagination, nor any additional language constructs (unlike [19]).
- *Be fully controlled by the designer*. With the exception of RIML [19], which allows some customization of the size limit applied to generated pages, all the approaches described above are fully automatic; no human control is envisaged.
- *Use semantic information* to help the designer determine where to split: information on 'breakable' or 'splittable' fragments is useful, but not very rich semantically. When higher-level specifications, especially task models, are available (as in [5]), these specifications must be used to refine the splitting process. In particular, the temporal relationships between tasks must be used, which is not the case in [7].
- *Be able to adapt the dialog* (i.e., the transitions between pages or windows) in a flexible, customizable way. Most of the splitting approaches do not fulfil this requirement.

3. THEORETICAL FRAMEWORK

Our approach, like many others, relies on a high level description of the initial user interface. This description will be expressed in the user interface description language UsiXML. The principles set out, however, are generally applicable, and other mark-up languages could be used alternatively.

UsiXML is structured in four abstraction levels; following the reference framework introduced by Calvary et al. [5] and known as the "CAMELEON framework" (see fig.1)

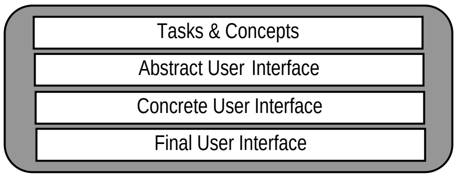


Figure 1. The four abstraction levels in the CAMELEON framework

The *Final User Interface* (FUI) refers to the actual user interface, which will be interpreted or rendered on a given computing platform. It is expressed in source code (e.g. Java or HTML).

The *Concrete User Interface* (CUI) abstracts the Final user Interface into a definition that is independent of any programming language. The CUI contains a detailed description of the user interface in terms of widgets (Concrete Interaction Objects in UsiXML), layout, navigation and behaviour. Fig.2 illustrates the structure of the CUI of a simple information retrieval system.

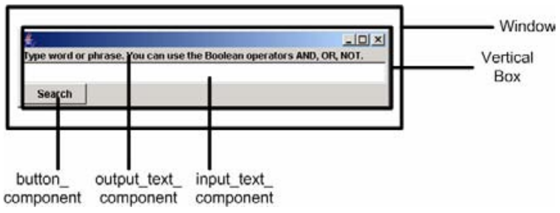


Fig.2 Some elements of a CUI model

The *Abstract User Interface* (AUI) abstracts a Concrete User interface into a UI definition that is independent of any modality of interaction. The AUI is an expression of the UI in terms of *interaction spaces*, i.e. the grouping of tasks that have to be presented together, in the same window or page for example. In UsiXML, the 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 (input, output, control, navigation). Fig.3 shows a representation in the IdealXML environment [16] of the AUI level corresponding to the Concrete user interface exemplified on fig.2.

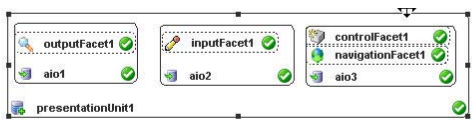


Fig.3 Graphical representation of a AUI model in the IdealXML environment

At the top of the framework, the *Tasks and Concepts* level describes the interactive systems' specifications in terms of the user tasks to be carried out and the domain objects manipulated by these tasks. The Tasks and Concepts level in UsiXML reuses existing formalisms: CTT [17] for the representation of tasks and class diagrams for the domain model. One peculiarity of UsiXML is that this level may be embedded into the specification and is not considered as a separate artefact only used in the requirement analysis development stage. Fig.4 shows the task model corresponding to the same user interface described above at the concrete and abstract level.

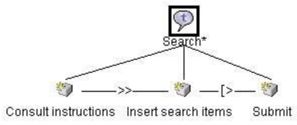


Fig.4 Graphical representation of the CTT-based task model in UsiXML

4. SPLITTING AT THE CUI LEVEL

Not all the layers listed above are mandatory in a user interface specification in UsiXML. In the simplest case, we suppose that the designer has just produced a description of the Concrete User Interface (CUI). The CUI may have been build by hand, using any XML editor or text editor, or with a graphical editor (GrafiXML), or recovered from existing code using reverse engineering tools (Vaquita [4]).

Different constructs in the CUI model of UsiXML can be used for pagination purposes:

− The layout of each container (window, dialog box…) is specified using embedded *boxes*. Those boxes are declared as *splittable* or not, which is the basic ingredient for pagination.

− Each container and each component of the CUI is marked as *pageable* or not. *Pageable* components can be distributed between the graphical containers created during the splitting process, while *non pageable components* have to be present in each fragment. For example, a menu bar or a widget permitting to logout from the system could constitute non pageable components, because their presence in each container is useful.

− *Transitions* can be specified between each pair of containers.

Implementing splitting rules starting from such a model is quite a straightforward process: the splittable attribute indicates where to split, and the pageable attribute indicates which elements will be duplicated. When several boxes are splittable, we choose the most-outer box.

Each execution of our splitting rules is fully controllable and configurable by the designer. The parameters taken into account by the algorithm are:

− The number of interactive spaces at output. By default, we take the number of boxes directly embedded into the main container (level 1–boxes).

− The content of the n interactive spaces at output. By default, we take the content of each level 1–box, but the designer is allowed to select content by drag-and-drop.

− The names assigned to each interactive space at output, which will be used as windows titles and for widgets pointing to these interactive spaces. By default, names are automatically generated by suffixing the original name.

One last parameter, namely the type of transitions generated between the new interaction spaces, deserves a little more explanation.

As the dialog model of UsiXML does not possess a graphical representation yet, we will represent the behavioural aspects as statecharts. We have considered four types of transitions between the interaction spaces generated by the splitting algorithm (hereafter target interaction spaces):

- *Linear navigation* (fig.5) establishes transitions between one target interaction space and another interaction space considered as its successor. It is typically realized with "next-previous" links or buttons. This type of navigation offers the most guidance to the user. Linear navigation is unidirectional (for example, "next" links only) or bidirectional (going backward is allowed).
- *Indexed navigation* (fig.6) establishes transitions between a newly created interaction space, the index, and each target interaction space. Unidirectional indexed navigation provides only transitions from the index to the other interaction spaces; while bidirectional indexed navigation offers transitions in both directions.
- *Mixed navigation* (fig.7) is a combination of linear and indexed navigation.
- *Fully-connected navigation* (fig.8) links each pair of target interaction spaces. This type of navigation is the least restricting for the user. It is typically rendered as a tabbed panel.

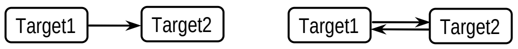


Fig.5 Unidirectional/bidirectional linear navigation

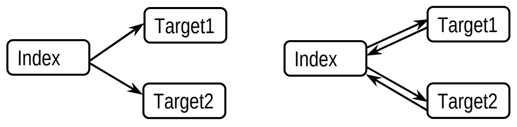


Fig.6 Unidirectional/bidirectional indexed navigation

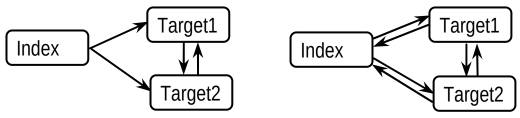


Fig.7 Unidirectional/bidirectional mixed navigation

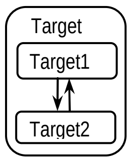


Fig.8 Fully-connected navigation

The splitting algorithm has been integrated in a plug-in for the GrafiXML environment. This plug-in implements a collection of transformation rules, to be applied to a source interface defined at the CUI level in order to produce specific interfaces targeted to more constrained platforms. Splitting rules are only one part of those rules that also include repositioning, interactor substitution, etc. The application of the splitting rule is under the control of the designer. He/she can parameterize the rule, apply it, preview the results, and compare alternatives. He/she is given guidance thanks to a knowledge base of rules linked with the tool. This knowledge base contains examples of the application of each rule as well as usability guidelines, collected in the literature and in a usability study specially dedicated to users' perception of the application of those transformations.

5. SPLITTING AT THE AUI LEVEL

Until now, we have supposed that the designer has just produced a description of the Concrete User interface. Let's now consider the scenario where a task model and an Abstract user interface have been produced. In that case, we can use the high level information from the task model to refine our algorithm.

5.1 Theory and principles

The CTT-based task model of UsiXML is a hierarchy of tasks, where each task can be decomposed into two or more subtasks. A task T can be declared as optional ([T]) or iterative (T*). Sibling tasks, appearing at the same level in the task hierarchy, are connected by temporal/logical operators:

- *Choice* T1 [] T2: exclusive choice between T1 and T2
- *Order independency* T1 |=| T2: T1 and T2 can be performed in any order
- *Independent concurrency* T1 ||| T2 and *Concurrency with information exchange* T1 |[]| T2: T1 and T2 can be performed in any order. We shall call these operators "concurrent operators".
- *Disabling* T1 [> T2 and *Suspend-resume* T1 |> T2: T2 disables/interrupts T1.
- *Enabling* (T1 >> T2) and *Enabling with information passing* (T1 []>> T2): T1 and T2 are executed in sequence. We shall call those operators "sequential operators".

There is a priority ordering between the temporal operators, as shown on fig.9.

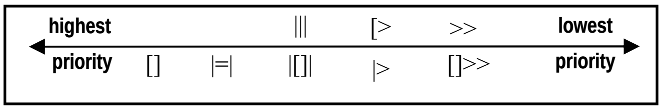


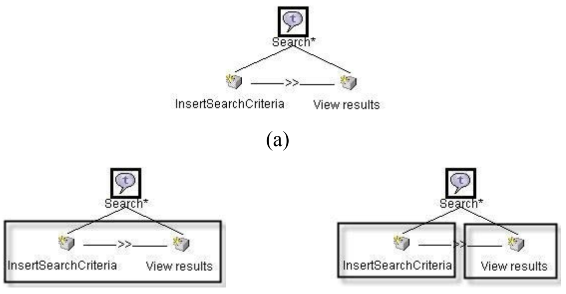
Fig.9 Priority ordering between the temporal operators in the task model

An easy way to cope with the priorities among the temporal operators is the priority tree technique used by Luyten & al. [14]. A priority tree is a view on a task model, with the same semantics as the original task model, but where all the temporal relations at the same level in the task hierarchy have the same priority according to their defined order.
Our algorithm relies on a few principles, detailed hereafter.

►**Principle 1:** An interaction space can be split at the level of a sequential operator.

"Splitting at the level of operator Op" means that all the tasks (and their descendants) to the left of Op which belonged to the source interaction space will be assigned to target interaction space 1 (IStarget1) and that all the tasks (and their descendants) to the right of o which belonged to the source interaction space will be assigned to target interaction space 2 (IStarget2).
Unlike generative approaches such as Paterno's ETS algorithm [17], we do not consider that sequential tasks are automatically assigned to different interaction spaces. This decision is under the responsibility of the designer. Presenting sequential tasks in the same interaction space can make sense, especially when those sequential tasks decompose a higher level task which has to be accomplished iteratively, such as on fig.10. In this example, a simple information retrieval system, a designer could choose to

place the two sequential cases, tasks "insert search criteria" and "view results" into the same interaction space, if the screen space is unconstraint (fig. 10b). If a new version of the user interface has to be conceived for a platform with less display capabilities, pagination could be operated at the level of the sequential operator, creating the two interaction spaces in fig.10c.

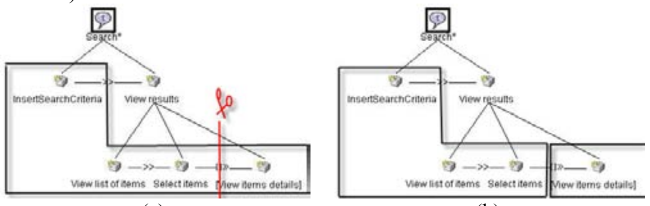


(b) (c)

Fig.10 Task model for a simple information retrieval system, with different distribution of tasks among interaction spaces

►Principle 2: When an interaction space includes several sequential tasks, split before the first optional task in the sequence.

In the cases where the optional task is not actually carried on, the user will not even have to navigate to the second interaction space. For example, let's consider the extension of the previous task model on fig.11. The "View results" task now consists of three subtasks: the first subtask displays the full list of result generated by the request to the information system; the second subtask is a selection task that puts the focus on one of the displayed results; and the last subtask is an optional task which displays a complete description of the selected item. Again, in a first, unconstrained version of the specification, the designer could choose to present all the tasks in the same interaction space, as on fig.11a. If we now try to determine the best place to operate pagination, we will notice that here, obviously, the best design solution is to split the source interaction space before the optional task "View items details", generating the target interaction spaces shown on fig.11b. (By the way, splitting between two tasks linked with the same component of the AUI is not allowed: it would not make sense to split between "View list of items" and "Select items".)

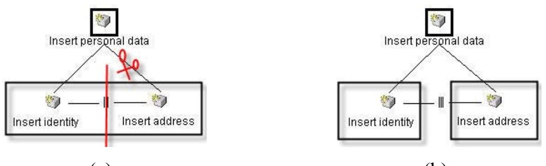


(a) (b)

Fig.11 Splitting an interaction space containing a sequence of tasks, one of them being an optional task

►Principle 3: When it is not possible to split an interaction space at the level of a sequential operator, split at the level of a concurrent, order independency or choice operator (|||, |[]|, |=|, [])

The temporal operators with a lower priority will be considered first. Splitting at the level of an interrupting or disabling task is not allowed. Of course, splitting at the level of one of the four operators above introduces constraints that were not present in the task model, and splitting at the level of a sequential operator should always be preferred when possible. Fig.12 shows the example of a (very) small task model which consists of the higher level task "Insert personal data" and its two concurrent subtasks "Insert identity" and "Insert address". The initial interaction space contains both subtasks (fig.12a). Splitting separates those two subtasks (fig.12b).

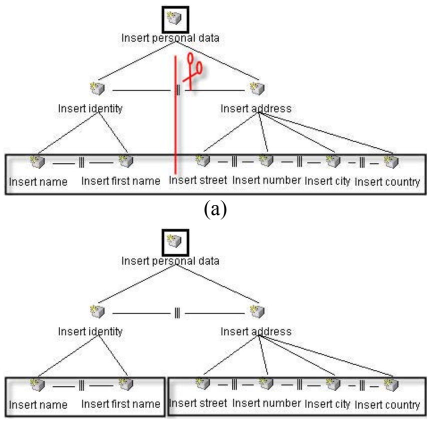


(a) (b)

Fig.12 Splitting an interaction space at the level of a concurrent operator

►Principle 4: When splitting rules can be applied at distinct levels in the task hierarchy, split at the highest level.

The intuition behind this principle is that tasks at a lower level in the task tree will be more closely semantically linked than tasks at a higher level. For example, let's consider the task model on fig.13, a more complete version of the previous example. On the first, less constrained platform, the designer could place all the tasks together (fig.13a). If less space is available, the best place to operate pagination, obviously, is to split the source interaction space at the highest level in the hierarchy, generating the target interaction spaces shown on fig.13b. This transformation preserves the integrity of the "Insert identity" and "Insert address" tasks: their subtasks, which were considered by the designer as tied enough to form concepts, are maintained together.



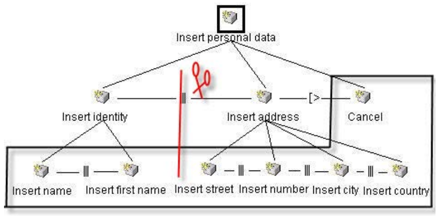
(b)

Fig.13 Splitting an interaction space containing concurrent tasks at different level in the hierarchy

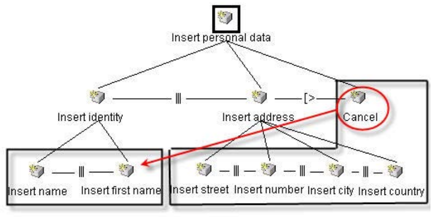
►Principle 5: When splitting in the scope of an operator with a higher level of priority, a distribution of tasks amongst target interaction spaces has to be operated.

Let's consider the small extension to the previous example shown on fig.14. If we naively split at the level of the sequential operator as described above, we will obtain a first interaction space IS1 = (Insert name, Insert first name) and a second interaction space IS2 = (Insert street, Insert number, Insert city, Insert country, Cancel).

Such a transformation introduces a discrepancy between source and target platform since it is not possible anymore to access the disabling "Cancel" task when performing the "Insert identity" task on the target platform: the user has to realize entirely the "Insert identity task", and then he or she can access the second interaction space where the "cancel" task is available. This kind of design is not ergonomic and can be frustrating for the user, especially when long tasks have to be achieved entirely without possibility of interruption. A better transformation rule should distribute the task to the right of the disable operator to each target interaction space, as illustrated on fig.14b, giving a solution with IS1 = (Insert name, Insert first name, Cancel) and IS2 = (Insert street, Insert number, Insert city, Insert country, Cancel). A more formal definition of the distribution principle will be given in section 5.2.



(a)



(b)

Fig.14 An example of distribution of a disabling task

5.2 Description of the algorithm

Our algorithm requires at input a subset of tasks of the task model, viewed as a priority tree. This subset of tasks, or Source interaction space (hereafter: ISsource) contains the leaf tasks that are mapped to the components of an abstract container that the designer has decided to split.

ISsource is a list (T1,…, Tn) where

- T1,…, Tn are leaf tasks in the task model.
- (T1,…,Tn) is a subsequence of the list (Ti,…,Tj) formed by the leaves nodes in the task model, considered as an ordered tree.

Until the interaction space is split and unless there are no more operators to go through:

1. We try to split at the level of a sequential operator

1.1. If there is an optional task in the sequence, we split before this task

1.2. Else

1.2.1. We look for the first suitable sequential operator where to split. "First" means "at the highest level in the task hierarchy" and "suitable" means that splitting at that place would generate non empty target interaction spaces, well balanced in terms of number of tasks. We search the task tree applying a breath-first strategy, starting from the task in the task model that is the lowest common ancestor of the tasks forming ISsource.

1.2.2. If such a sequential operator is found, the source interaction space is then split into two temporary target interaction spaces IStarget1 and IStarget2. Let T1 and T2 be the two tasks in the task model linked by the operator where we have decided to split. IStarget1 will contain the first part of ISsource, delimited by T1 if T1 is a leaf task, its right-most descendant otherwise. IStarget2 will contain the remainder of ISsource.

1.2.3. We then apply distribution rules.

2. When it was not possible to operate sequential splitting, we then try to split at the level of another operator.

2.1. We look for the first suitable operator where to split.

2.2. If such an operator is found, the source interaction space is then split into two temporary target interaction spaces.

2.3. We then apply distribution rules.

Distribution rules are applied when splitting occurs between two tasks T1 and T2 that are in the scope of a temporal operator with higher priority (see fig.9). By construction, splitting always occurs between sister tasks in the priority tree. Let T1 and T2 be two sister tasks, linked by temporal operator Op1. T1 and T2 are in the scope of temporal operator Op2 iff an ancestor of T1 and T2 is linked by a temporal operator Op2 to a given task T3. If Op2 has a higher priority level than Op1 and T3 has descendants in ISsource, then distribution must be applied. Distribution consists in appending to the right of IStarget1 the descendants of T3 that belong to ISsource and appending to the left of IStarget2 the descendants of T3 that belong to ISsource. The algorithm begins by considering distribution at the level of the mother of T1 and T2, and then the upper levels are successively considered, until reaching the level of the lowest common ancestor of all tasks in ISsource.

5.3 Implementation

This algorithm has been integrated to the IdealXML environment [16]. IdealXML is a development environment which permits to specify user interfaces in UsiXML at different abstraction levels. The developer specifies a task model, an AUI model, and mappings between those levels in IdealXML. As explained earlier, the AUI level is composed of containers and components. If the designer decides that a container contains too much components, he might choose to split this container into smaller units. The designer selects this abstract container. The tool retrieves the set of leaf tasks linked with the components inside the container. This set of leaf tasks is given as input to the algorithm described above. The algorithm gives as output two subsets of tasks to be integrated in two separate subsets of containers. The original container in the AUI is replaced by two new containers, each of them containing appropriate components. The mapping between leave tasks and components is kept constant, only the mapping between higher level tasks and containers is modified.

6. Example

To exemplify the pagination algorithm on a more complex user interface, we now consider an on-line hotel booking form (Figure 15).

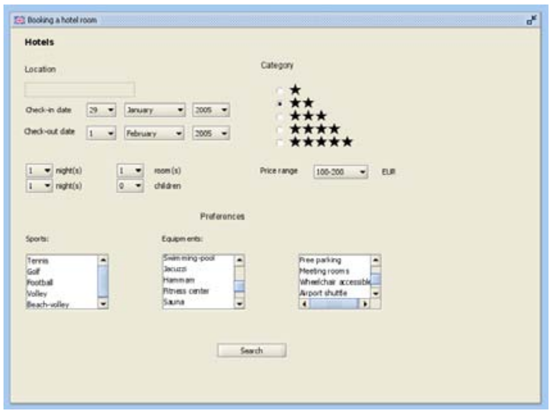
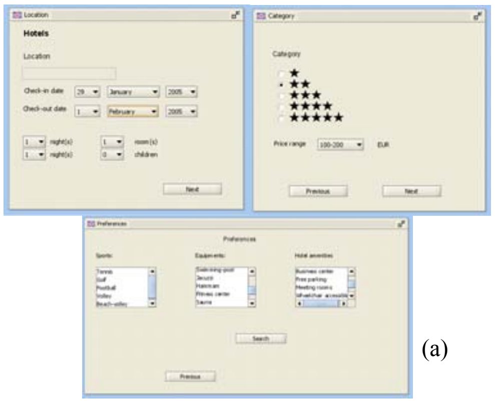
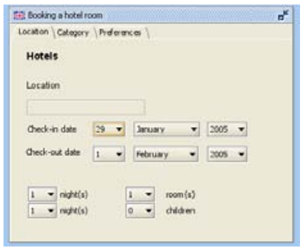


Figure 15. The CUI of the hotel booking system in GrafiXML

Let us assume that we want to transform this UI for display on a smaller screen, such as a PDA. The designer, with the help of the tool, can explore several alternatives, depending on the type of navigation required between the fragments generated, or the number of fragments desired at output. Figure 16a shows the result of applying the splitting rule with the "sequential navigation" parameter. Alternatively, the system could generate a tabbed dialog box with the same contents rearranged (Figure 16b)



(a)



(b)

Figure 16. The CUI of the hotel booking system after applying the splitting rules

In this scenario, the decision on where to split was based on the specification, by the designer, of splittable/unsplittable boxes. In another scenario, we could suppose that the designer has started the development of the hotel booking system by specifying a task model, in a tool such as IdealXML. For the moment, we do not have any implementations of a 'split' CUI starting from a 'split' AUI. However, the two UsiXML-compliant tools GrafiXML and IdealXML could collaborate in this way:

1. An AUI, a task model and mappings are built and edited with IdealXML.
2. A CUI is edited with GrafiXML. The designer does not want to specify by hand which boxes are splittable and which unsplittable.
3. Consider a given group, box *x*. We want to determine whether box *x* is splittable or not. The process is as follows:
   3.1. retrieve abstract container *y* at the AUI level which is mapped to box *x*;
   3.2. retrieve leaf tasks T1,…,Tn in the task model which are executed in container *y*;
   3.3. apply the splitting algorithm;
   3.4. if the result is non-empty, box *x* is marked as splittable; otherwise, it is marked as unsplittable.
4. Apply the splitting rule at the CUI level.

Phases 3.1 and 3.4 have still to be done manually since they are supported by two different software packages. When the splitting process is finished the designer is still free to apply transformation rules that belong to levels exhibiting a lower level of granularity (Section 1).

7. CONCLUSION AND FUTURE WORK

We have described a pagination technique, which relies on a high level description of the UI in the user interface description language UsiXML. When applied at the Concrete level, the algorithm proposed is quite classical. Nevertheless, it goes further than state-of-the art approaches from several points of views. It is generic: once splitting applied at the CUI level, final user interfaces can be generated in Java or HTML for example, so that we do not have to write separate programs to paginate Java or HTML user interfaces. It relies on pre-existing structures of UsiXML. It can be applied automatically, using default parameters, but it can also be fully controlled by the developer, which can choose the number of pages at output, the type of dialog generated and the content of the pages. It proposes a large range of dialog styles, when other approaches often only generate sequential navigation.

However, the originality of the technique proposed is to involve UI description at several abstraction levels As far as we know, there have been no similar tentative to use information from the AUI and task levels in order to improve the splitting process. This multilevel approach is quite new and exciting. Existing model-based tools which generate several versions of a user interface for multiple platforms adopt a totally different approach: either they generate code starting directly from a description at the tasks and concepts level [20][18], which offers little or no control on the layout and structure of the final interface, either they require to specify a distinct CUI for each target platform or each family of target platforms [1], which has the double disadvantage to demand more work to the designer and to offer no guarantee of consistency between the distinct versions of the final interface. In contrast, our approach requires only one specification, that can be given with all the level of detail required (choice of widgets, layout,…) while taking advantage of information specified at higher abstraction levels if this information is available.

The main limitation of our approach is that it is meant to be applied as design time, as a part of a single authoring process (graceful degradation). Investigation of the utility of our algorithms at run-time, on pre-existing user interfaces, remains to be done. For example, a possible future work would be to consider the pagination at run-time of Web pages with an embarked task model, and to compare the results with existing approaches such as [6].

8. ACKNOWLEDGMENTS

We gratefully thank the Salamandre Project, funded by the "Initiatives III" research program of the Ministry of Walloon Region, DGTRE, Belgium and the SIMILAR network of excellence, supported by the IST 6th framework program of the European Commission (project # FP6-507609).

9. REFERENCES

[1] Ali M.F., Pérez-Quiñones M.A. and Abrams M. Building Multi-Platform User Interfaces With UIML. In: A. Seffah & H. Javahery (eds.) *Multiple User Interfaces: Engineering and Application Framework.* John Wiley and Sons, Chichester, UK, 2004.

[2] Artail, H.A., and Raydan, M. Device-aware desktop web page transformation for rendering on handhelds. *Personal and Ubiquitous Computing, 9(6)* (2005), 368-380.

[3] Banavar, G., Bergman, L.D., Gaeremynck, Y., Soroker, D., and Sussman, J. Tooling and system support for authoring multi-device applications. *Journal of Systems and Software 69(3)* (2004), 227-242.

[4] Bouillon, L.; Vanderdonckt, J. & Chow, K.C. Flexible re-engineering of web sites. In *Proceedings of the 2004 International Conference on Intelligent User Interfaces IUI'04* (Funchal, Madeira Island, Port., January 13-16, 2004)

[5] Calvary G., Coutaz J., and Thevenin D. A unifying reference framework for the development of plastic user interfaces. In *Proceedings of IFIP WG 2.7 Conference on Engineering the User Interface EHCI'2001* (Toronto, May 11-13, 2001), Chapman & Hall, London, 2001.

[6] Chen, Y., Xie, X., Ma, W.-Y., and Zhang, H.-J. Adapting Web Pages for Small-Screen Devices. *IEEE Internet Computing, 09(1)* (2005), 50-56.

[7] Chu, H., Song, H., Wong, C., Kurakake, S., and Katagiri, M. Roam, a seamless application framework. *Journal of System and Software 69(3)* (2004), 209-226.

[8] Ding, Y., and Litz, H. Creating Multiplatform User Interfaces by Annotation and Adaptation. In *Proceedings of the 10th International Conference on Intelligent User Interfaces IUI'2006* (Sydney, January 29 - February 1, 2006).

[9] Florins, M., and Vanderdonckt, J. Graceful degradation of user interfaces as a design method for multiplatform systems. In *Proceedings of the 2004 International Conference on Intelligent User Interfaces IUI'04* (Funchal, Madeira Island, Port., January 13-16, 2004).

[10] Giller, V., Melcher, R., Schrammel, J., Sefelin, R., and Tscheligi, M. Usability Evaluations for Multi-device Application Development - Three Example Studies. In *Proceedings of Mobile HCI'03* (Udine, Italy, Sept. 8-11, 2003).

[11] Göbel, S., Buchholz, S., Ziegert, T., and Schill, A. Device Independent Representation of Web-based Dialogs and Contents. In *Proceedings of the IEEE YUFORIC ′01* (Valencia, Spain, Nov. 2001).

[12] Henry, C. & Henry, K. *Recherche sur les préférences des utilisateurs en ce qui concerne la dégradation des interfaces en vue d'être visionnées sur des plates-formes à petit écran*. Master's thesis, IAG, Université catholique de Louvain, Louvain-la-Neuve, 2004.

[13] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Florins, M. & Trevisan, D. USIXML: A User Interface Description Language for Context-Sensitive User Interfaces. In *Proceedings of the first international Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages* (Gallipoli, Italy, 25 May 2004).

[14] Luyten, K., Clerckx, T., Coninx, K. & Vanderdonckt, J. Derivation of a Dialog Model from a Task Model by Activity Chain Extraction. In *Proceedings of DSV-IS 2003-10th International Workshop on Design, Specification, and Verification of Interactive Systems* (Funchal, Madeira Island, Portugal, June 11-13, 2003).

[15] Mandyam, S., Vedati, K., Kuo, C. and Wang, W., User Interface Adaptations: Indispensible for Single Authoring. In *Workshop on Device Independent Authoring Techniques* (St. Leon-Rot, 15-26 September 2002).

[16] Montero, F., López-Jaquero V., Vanderdonckt J., Gonzalez P., Lozano, M. D. and Limbourg, Q. Solving the Mapping Problem in User Interface Design by Seamless Integration in IdealXML In *Proceedings of DSVIS'05 - 12th International Workshop on Design, Specification and Verification of Interactive Systems* (Newcastle upon Tyne, UK, July 13–15, 2005)

[17] Paternò F. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, London, UK, 1999.

[18] Paternò, F., Mori, G. and Santoro, C. Tool Support for Designing Nomadic Applications. In *Proceedings of 7th Int. Conf. on Intelligent User Interfaces IUI'03* (January 12-15, 2003, Miami), ACM Press, New York.

[19] Spriestersbach, A., Ziegert, T., Grassel, G., Wasmund, M., and Dermler, G. Flexible pagination and layouting for device independent authoring. In *WWW2003 Emerging Applications for Wireless and Mobile access Workshop* (not printed).

[20] Thevenin D. *Adaptation in Human Computer Interaction: the case of Plasticity*. Ph. D. Thesis, Joseph Fourier University, Grenoble, 2001.

[21] Watters, C., and Zhang, R. PDA Access to Internet Content: Focus on Forms. In *Proceedings of HICSS'03, the 36th Annual Hawaii International Conference on System Sciences* (Big Island, Hawaii, January 2003).

[22] Ye, J., and Herbert, J. User Interface Tailoring for Mobile Computing Devices. In *Proceedings of UI4All, 8th ERCIM Workshop « User Interfaces for All »* (Vienna, Austria, 28-29 June 2004).