Preserving Designer Input on Concrete User Interfaces Using Constraints While Maintaining Adaptive Behavior

Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu
Computing Department, The Open University
Walton Hall, Milton Keynes, United Kingdom
{pierre.akiki, a.k.bandara, y.yu}@open.ac.uk

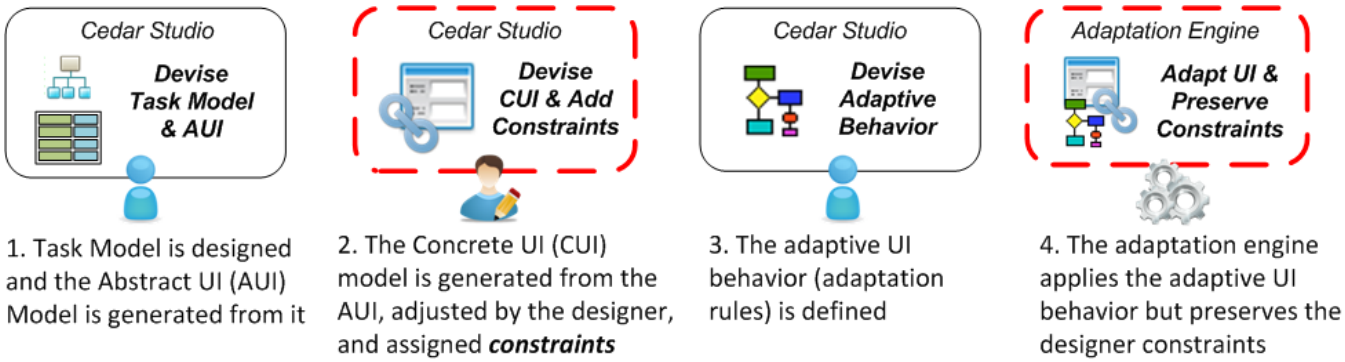

Figure 1. Adding Constraints on the CUI as Part of the Process of Developing Adaptive Model-Driven UIs (Step 2) and Maintaining These Constraints When the Adaptation Engine Applies the Adaptive Behavior (Step 4)

ABSTRACT
User interface (UI) adaptation is applied when a single UI design might not be adequate for maintaining usability in multiple contexts-of-use that can vary according to the user, platform, and environment. Fully-automated UI generation techniques have been criticized for not matching the ingenuity of human designers and manual UI adaptation has also been criticized for being time consuming especially when it is necessary to adapt the UI for a large number of contexts. This paper presents a work-in-progress approach that uses constraints for preserving designer input on concrete user interfaces upon applying adaptive behavior. The constraints can be assigned by the UI designer using our integrated development environment *Cedar Studio*.

Author Keywords
Adaptive user interfaces; Designer input; Constraints; Concrete user interfaces; Model-driven engineering

ACM Classification Keywords
D.2.2 Design Tools and Techniques - User interfaces; [Information Interfaces and Presentation]: H.5.2 User Interfaces – User-centered design

General Terms
Design; Human Factors

INTRODUCTION
User interface (UI) adaptation is applied when a single UI design might not be adequate for maintaining usability in multiple contexts-of-use that can vary according to the user, platform, and environment. UI adaptation is either labeled as adaptable meaning that manual adaptation is required or adaptive indicating that an automatic adaptation is done. By observing the literature we can see that there are a variety of UI adaptation techniques that adopt manual adaptation (adaptable UI) such as "*two interface design*" [14] and "*crowdsourced adaptation*" [17] or automated adaptation (adaptive UI) such as "*Supple*" [13], and "*Personal Universal Controller*" [18].

Some researchers have criticized fully-mechanized UI construction in favor of applying the intelligence of human designers for achieving higher usability [21]. Adaptive UI behavior is also regarded by some as being unpredictable and possibly disorienting for users [11]. Other researchers promote the use of adaptive behavior [5]. The automation provided by adaptive behavior provides advantages in terms of saving development time thereby reducing the cost of adapting user interfaces to multiple contexts-of-use.

The importance of obtaining a predictable outcome is emphasized due to its impact on the success of UI development techniques [16]. Some fully-automated approaches only allow designer input on a high level of abstraction thereby decreasing the control and predictability of the outcome. Other approaches support lower level input such as control over the concrete widgets, nevertheless upon applying adaptive behavior the input made by the human designer will be overridden.

In this paper, we present a work-in-progress technique that allows designers to assign UI constraints that are preserved after applying automated adaptive behavior. The constraints embody the characteristics of the UI that require human ingenuity and are not met by fully-automated techniques.

The model-driven approach to user interface development has been promoted by many research works such as the well-established CAMELEON reference framework [6]. CAMELEON represents user interfaces on multiple levels of abstraction: (1) *Task Models* can be represented as ConcurTaskTrees [20] and *Domain Models* as UML class diagrams, (2) *Abstract User Interface* (AUI), represents the UI independent of any modality (e.g., graphical, voice, etc.), (3) *Concrete User Interface* (CUI), represents the UI as concrete widgets (e.g., buttons, labels, etc.), and (4) *Final User Interface* (FUI), is the running UI rendered in a presentation technology. The model-driven approach to UI development can serve as a basis for devising adaptive UIs due to the possibility of applying different types of adaptations on the various levels of abstraction [2]. Out of the levels of abstraction presented by CAMELEON, the CUI will be given particular attention in this paper since it embodies the designer's ingenuity. Designer input on the CUI is particularly promoted by indicating that it would be better if the designer can manipulate a concrete object rather than its abstraction [9]. By following such recommendations, we can say that the designer should be allowed to create a CUI rather than completely generating it from an abstract model. Yet even though some approaches might offer designers with the ability to create CUIs, upon applying the adaptive UI behavior the designer's choices are bound to change according to the adaptive UI behavior particular to a given context-of-use. Nevertheless, in certain cases designers would like to keep some UI characteristics intact. We think this could be achieved by providing non-technical UI designers with a simple technique for assigning constraints on the CUI. These constraints could be taken into consideration and preserved at a later stage when the UI is being automatically adapted to a particular context-of-use.

The steps illustrated in Figure 1 show where our proposed technique fits in the process of developing adaptive model-driven UIs. We can see that the constraints are added by the designer in Step 2 after adjusting the CUI design. Later, in Step 4 when the adaptation engine applies the adaptive behavior it preserves the designer's constraints.

The remainder of this paper is structured as follows. The next section briefly describes the related work. Then, an example is given to highlight the importance of preserving designer input on the CUI. Later, our approach to applying CUI constraints is described. Finally, the conclusions and future work are given.

RELATED WORK

By observing the literature we can categorize UI adaptation approaches under the following categories:

- *Adaptable* UIs allow interested stakeholders to manually adapt the desired characteristics
- *Adaptive* UIs automatically react to a change in the context-of-use by changing one or more of their characteristics using a predefined set of adaptation rules
- *Truly Adaptive* UIs can automatically react to a change in the context-of-use but are also capable of reacting to contexts-of-use that were previously unknown

Adaptable UIs fully support manual designer input, which provides an advantage in terms of applying the knowledge of a human designer but has a downside in terms of high development time. Both *Adaptive* and *Truly Adaptive* UIs provide a higher level of automation through the ability of adapting the UI using generic rules but even though the rules are meant to produce an optimal UI based on the context-of-use, in some cases the input of the human designer can be essential (e.g. widget size, position, etc.).

Raneburger et. al. presented an approach to automated generation of WIMP style UIs. They attempt to enhance the quality of the generated UIs by using a graphical tree editor to add hints to the transformations (e.g., the alignment of a widget) [22]. One problem is that UI designers might only work on the CUI level and the specification of the model transformations would be left to the developers. Also, the authors state that a graphical "*what you see is what you get*" (WYSIWYG) editor similar to the one presented by the Gummy [15] system would improve on their approach.

Supple is primarily capable of automatically generating UIs that are adapted to each user's motor abilities by treating UI generation as an optimization problem [13]. Yet, although the authors mention that Supple is not intended to replace human designers, the system only relies on a high level model to generate its final UI thereby preventing designer input from being made on the CUI level.

DynaMo-AID [7] is presented as part of the Dygimes UI creation framework. It incorporates a design process for the development of context-aware UIs that are adaptable at runtime. Like Supple this system focuses on a high level UI representation (task models), which is used for automatic generation of the CUI.

MASP [10] provides designers with a graphical design tool to support the creation of layout models, which are later interpreted at runtime for supporting adaptive UI behavior. Although the tool supports designer input, no mechanism is offered for maintaining this input after the adaptive behavior is applied.

Smart templates are proposed for improving automatic generation of ubiquitous remote control UIs on mobile devices [19]. Although these templates improve the ability of preserving designer input, specifying the various template variations could be time consuming and would be classified under adaptable rather than adaptive behavior.

AN EXAMPLE OF USER INTERFACE CONSTRAINTS

We developed a mechanism called Role-Based User Interface Simplification (RBUIS) [2] for simplifying UIs by minimizing their feature-set and optimizing their layout based on the context-of-use (user, platform, environment). We define a minimal feature-set as the set with the least features required by a user to perform a job. An optimal layout is the one that maximizes satisfaction of the constraints imposed by a set of aspects such as computer skills, culture, etc. An optimal layout is obtained by adapting the properties of concrete widgets (e.g., type, grouping, size, location, etc.). In RBUIS, the feature-set is minimized by applying roles to task models and the layout is optimized by executing adaptive behavior workflows on the CUI. The workflows can embody visual and code-based constructs. RBUIS is based on the CEDAR architecture [1] and uses interpreted runtime models for the adaptation. Nevertheless, the designer can still create an initial fully-featured CUI. The feasibility of adapting a least constrained UI design was shown in a previous research [12]. RBUIS follows a similar approach by adapting an initial UI that is without constraints in terms of the feature-set and least constrained in terms of the layout (e.g., least constrained screen size).

Adaptive UI behavior such as removing and adding widgets could leave gaps and deformations in the layout, which are not esthetically desirable and could increase the navigation time according to Fitts's Law. A mechanism is needed for maintaining plasticity, denoting the UI's ability to adapt to the context-of-use while preserving its usability [8]. Hence, we can consider layouting as one example of UI constraints that could be influenced by choices made by a human designer rather than merely automated choices. The example illustrated in Figure 3 is that of a sales invoice UI, usually common in enterprise applications such as enterprise resource planning systems. Let us consider that we would like to apply RBUIS to this UI in order to minimize its feature-set for a role that does not require all the initial features. The examples shown in Figure 4 and Figure 5 are two possible layouting alternatives that could be produced after eliminating the undesirable features from the UI. The differences between the two versions are the layouting choices related to group boxes "a" and "b" on one hand, and data grid "c" and text box "d". In Version 1, shown in Figure 4, the width of group box "b" is increased in order to prevent scrolling but this is at the expense of the width of group box "a", whereas in Version 2 shown in Figure 5 the opposite is done. Also, in Version 1 the width of text box "d" is increased at the expense of the height of data grid "c" whereas in Version 2 an opposite choice is made. In both cases there are no absolute right and wrong choices. Such choices depend on what the human designer thinks is more appropriate. Is giving more room for data entry in the fields of group box "a" and the text box in group box "d" more important than showing additional items on the screen in the radio button groups of group box "b" and data grid "c"? When an algorithm makes the choice between Versions 1 and 2 without providing any rationale, critics are going to deem adaptive UIs as being unpredictable. Empowering human designers could strike a balance between automation and human intelligence to increase adaptive UI predictability.

CONCRETE USER INTERFACE CONSTRAINTS

In many cases UIs are designed by non-technical designers. Also, in another work we have highlighted the possibility of engaging end-users in the UI adaptation process [3]. Therefore, we think that the constraints we are proposing should be kept simple in order to be implementable by the non-technical stakeholders. We devised a basic meta-model, illustrated in Figure 2, to reflect such constraints.

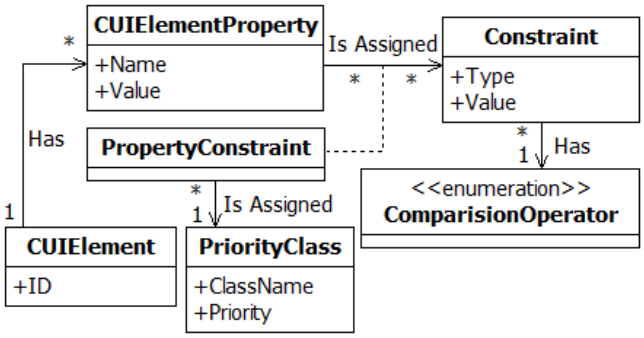

Figure 2. Simple CUI Constraints Meta-Model

Since each *CUIElement* has *Properties*, *Constraints* can be attached to these properties in order to reflect designer related choices regarding their values. A *Constraint* simply has a comparison operator (e.g., ">", "<", "=", etc.) and a value for comparison. In order to have a practical approach that promotes easier constraint assignment, a constraint's value should not necessarily be exact. It can be absolute or relative, quantitative or qualitative. For example, a constraint on the width of a widget could be "> 100" or it could be "= Large". It is possible to define ranges for such values or leave the decision to the adaptation engine to be made according to a given context and UI. Let us consider group boxes "a" and "b" presented in both Figure 4 and Figure 5. If the designer specified that the width of group box "a" should be "Medium" whereas that of group box "b" should be "Large" then the version in Figure 4 would be chosen and vice-versa. The same could work for data grid "c" and text box "d". The designer also has the ability to allocate each *Constraint* to a *Priority Class* in order to indicate which constraint would get eliminated in case a conflict occurs between two or more constraints. If conflicts still exist even with the priority classes, the system will then have to eliminate one at random. A *Constraint* can be one of two types: *Strict* or *Lenient*. For example, a lenient equality constraint indicates that the original value can be changed to close values whereas if it were strict it would mean that the value should be exactly the same but it can still be dropped in case of a conflict. The coming section explains how we distinguish explicit and implicit constraints and our proposition for applying them in practice.

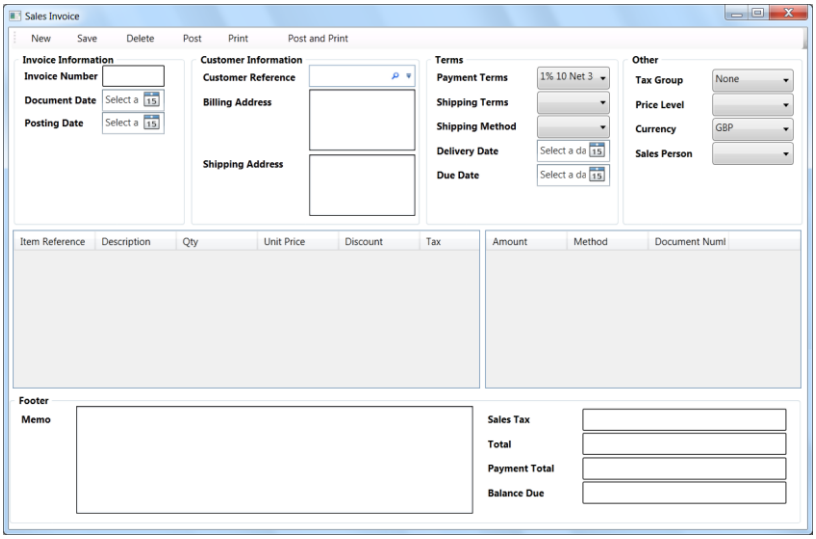

Figure 3. Initial Sales Invoice User Interface

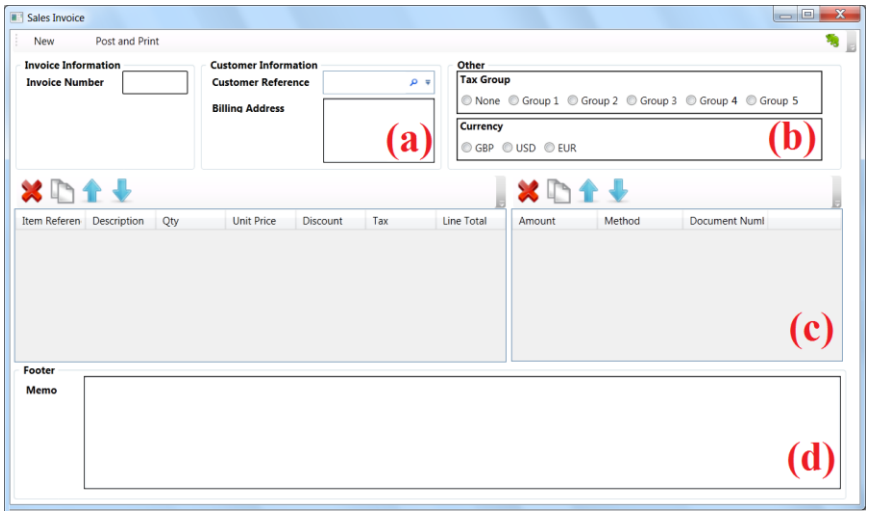

Figure 4. Adapted Sales Invoice User Interface Version 1

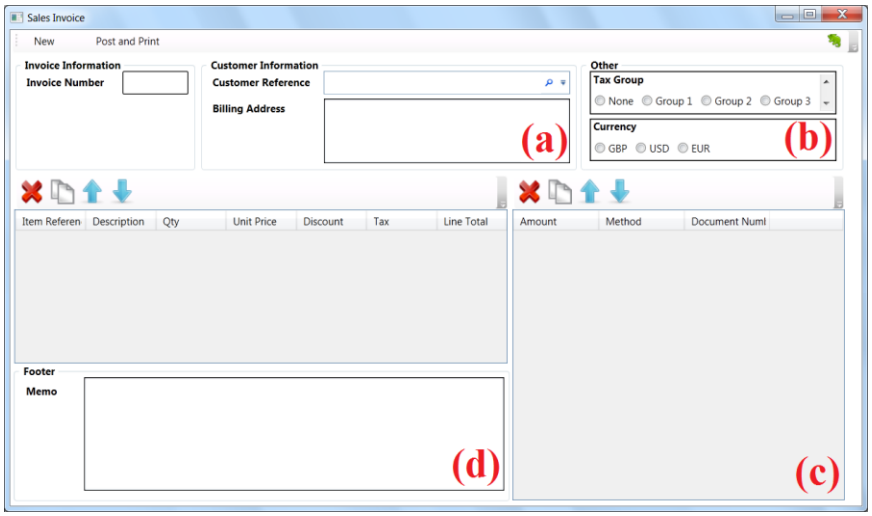

Figure 5. Adapted Sales Invoice User Interface Version 2

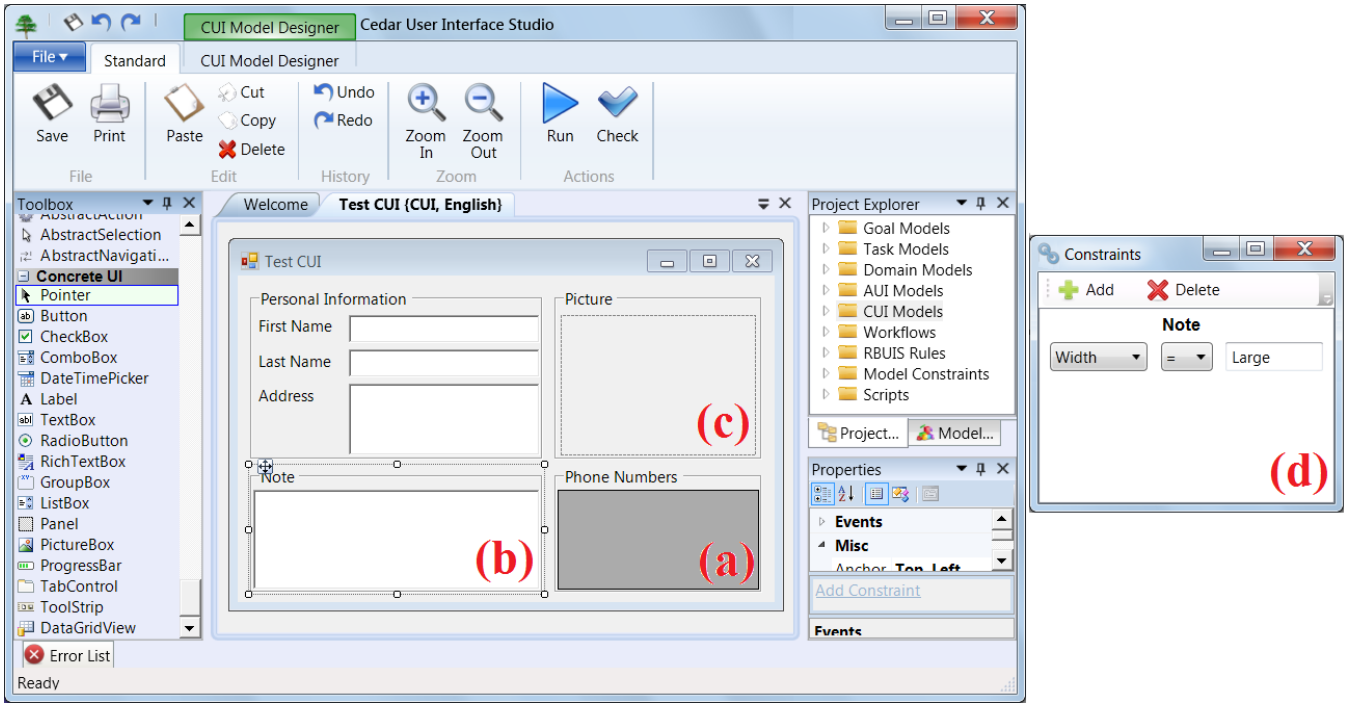

Figure 6. Assigning Concrete User Interface Constraints in *Cedar Studio*

APPLYING CONCRETE UI CONSTRAINTS

Cedar Studio is our integrated development environment (IDE) for supporting the development of adaptive UIs based on a model-driven approach [4]. We consider that designer constraints can be explicitly or implicitly specified. Explicit constraints are specified by the designer on the CUI properties whereas implicit constraints can be deduced from the design made on the canvas itself such as widget ordering and positioning relative to other widgets.

Explicit Constraints

We extended the CUI designer of *Cedar Studio* to support the addition of explicit designer constraints. Let us considers a basic example that requires such constraints and propose a technique for applying it in practice. Consider that the "Phone Numbers" grid (Figure 6 – a) should be eliminated for a given context-of-use. The layouting engine will be faced with two choices, either filling the space by increasing the width of the "Note" (Figure 6 – b) or by increasing the height of the "Picture" (Figure 6 – c). If the designer adds a constraint as shown in Figure 6 – d to indicate that the "Note" should have a "Large" width, the system should be able to incorporate this choice in a constraint problem that can be passed to a constraint solver.

Listing 1. Constraint Problem Written in Python on Z3Py

```
1. #variables to hold the final calculated width of the widgets
2. noteWidth, pictureHeight = Reals('noteWidth pictureHeight')

3. #initial width of the note and picture widgets
4. initialNoteWidth,initialPictureHeight = Reals('initialNoteWidth
   initialPictureHeight')
5. initialNoteWidth = 250; initialPictureHeight = 200

6. #the height and width of the canvas holding the widgets
7. canvasWidth, canvasHeight = Reals('canvasWidth
   canvasHeight')
8. canvasWidth = 300; canvasHeight = 200

9. solve (
   #the two possibilities
   (noteWidth == canvasWidth and pictureHeight ==
   initialPictureHeight) or
   (noteWidth == initialNoteWidth and pictureHeight ==
   canvasHeight),
   #constraint based on the designer's input
   noteWidth == max(canvasWidth, initialNoteWidth))
```

The problem shown in Listing 1 is expressed in Python and is relevant to the example demonstrated in Figure 6. It defines two variables "noteWidth" and "pictureHeight" to hold the calculated values of the widget properties. It takes as input the initial property values ("initialNoteWidth" and "initialPictureHeight") and the height and width of the canvas ("canvasHeight" and "canvasWidth") that are the possible values that these properties can take. The two possibilities at hand are either resizing the width of the "Note" widget to fit the canvas width and keeping the height of the "Picture" widget intact or vice-versa. Since the designer specified a constraint stating that the "Note" width should be "Large", the problem was supplied with a constraint "noteWidth == max (canvasWidth, initialNoteWidth)" in order to choose the largest possible value. Running the problem on the Z3Py [24] constraint solver yields the following result: "[*noteWidth* = 300, *pictureWidth* = 200]". The yielded values could be applied to the relevant CUI element properties to obtain an adapted user interface that preserves designer input.

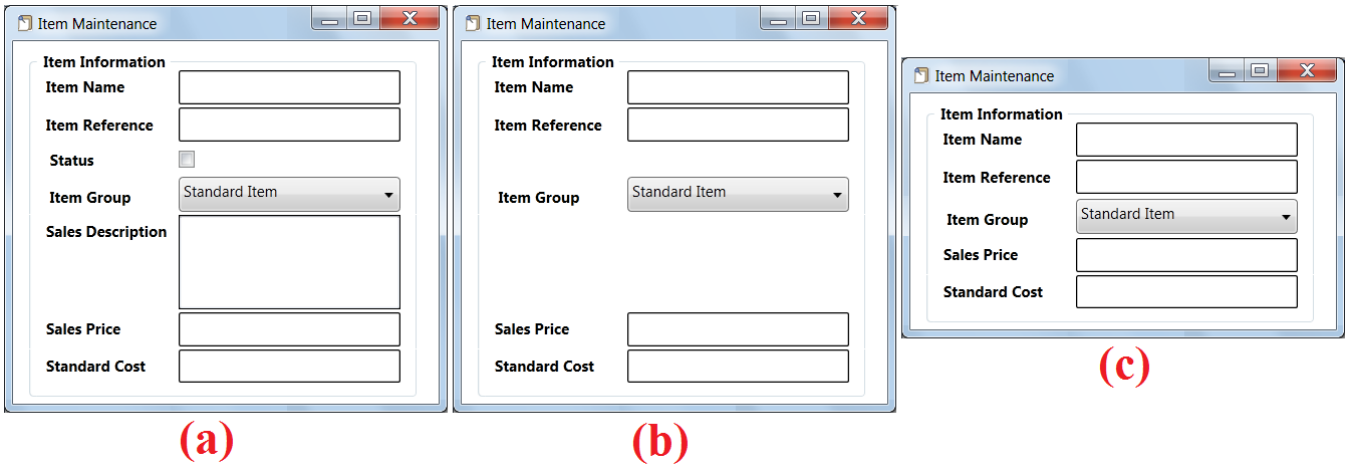

Figure 7. Implicit Concrete User Interface Constraints – A Relative Positioning Example
(a) Initial User Interface Design, (b) Minimized Feature-Set UI that Hides Widgets, (c) Refitted Layout UI Design

Implicit Constraints

An implicit layouting constraint that we worked on as part of the layouting algorithm supporting RBUIS is related to the relative widget positioning and ordering specified by the designer. Upon eliminating parts of the UI in Figure 7 – a to minimize its feature-set for a particular context-of-use as shown in Figure 7 – b, this algorithm would be responsible for refitting the UI by removing the gaps. The example in Figure 7 – c shows how the widgets are pushed upwards beneath the closest widget. Deducing implicit constraints from the design made on the canvas saves the designer the effort of adding these constraints separately.

Algorithm 1. UI Refitting Written in C# (Excerpt)

```
1.  public bool RefitTop(List<ControlInfo> Controls, int StartingTop = 5)
2.  {
3.      List<List<ControlInfo>> lines = this.GetControlLines(Controls);
4.      if (lines.Count == 0) { return true; }
5.
6.      foreach (ControlInfo control in lines[0])
7.      { control.Top = StartingTop;  }
8.
9.      for (int counter = 1; counter < lines.Count; counter++)
10.     {
11.         foreach (ControlInfo control in lines[counter])
12.         {
13.             int reverseLineCounter = counter -1;
14.             var ctrsAbove = new List<List<ControlInfo>> ();
15.
16.             while (ctrsAbove.Count() == 0 && reverseLineCounter >= 0)
17.             {
18.                 ctrsAbove = from l in lines[reverseLineCounter]
19.                             where (l.Left > control.Left - l.Width &&
20.                             l.Left < control.Left + l.Width)
21.                             orderby l.Height descending select l;
22.                 reverseLineCounter--;
23.             }
24.
25.             if (ctrsAbove.Count() > 0) {
26.                 ControlInfo ctrAbove = ctrsAbove.First();
27.                 control.Top = ctrAbove.Bottom + widgetMargin;
28.             }
29.             else { control.Top = StartingTop; }
30.         }
31.     }
32.     return true;
33. }
```

The part of our algorithm that pushes the widgets upwards is shown in Algorithm 1. We implemented the implicit constraints as a layouting algorithm due to its simplicity in comparison to having to generate a constraint problem such as the one shown in Listing 1. For example, the implementation excerpt shown in Algorithm 1 splits the CUI controls into ordered lines and moves each widget beneath the one above it from one of the previous lines. Expressing this algorithm as a separate constraint problem for different contexts would have been more difficult than writing one generic solution.

CONCLUSIONS AND FUTURE WORK

This paper presented a work-in-progress technique that allows designers to supply CUI constraints that would be maintained after applying automated adaptations. We categorized these constraints as explicit and implicit. Explicit constraints are specified by the designer on the CUI properties whereas implicit constraints can be deduced from the design made on the canvas such as widget ordering and positioning. Both types of constraints can be specified using our IDE *Cedar Studio*. We proposed the generation of constraint problems that could be solved by constraint solvers to satisfy explicit constraints. On the other hand we implemented implicit constraints relevant to widget positioning and ordering as a layouting algorithm.

More work is still required to make the proposed technique applicable in practice. A primary point would be devising an algorithm that would convert explicit designer constraints into a constraints problem such as the one shown in Listing 1. This algorithm should then be utilized by the adaptation engine in combination with the algorithm for refitting the UI based on implicit constraints in order to maintain the designer's input upon adapting the user interface. When this part is accomplished, then we can comprehensively test both explicit and implicit constraints in a real-life scenario by measuring the extent to which the usability is preserved and the efficiency of the technique.

Our solution is intended for allowing designers to add any type of constraints that can be applied on the properties of the concrete UI widgets. The incorporation of this solution in a generic IDE like *Cedar Studio* allows extensions to be made in the future. One possible extension would be supplying UI designers with the ability to automatically check the initial design (implicit constraints) based on general ergonomic rules [23] or to add these rules as explicit constraints. Another possibility is to use such ergonomic rules for prioritizing constraints in order to allow the system to make an informed decision when it faces two conflicting constraints that were assigned the same priority by the human designer.

ACKNOWLEDGMENT

This work is partially funded by ERC Advanced Grant 291652.

REFERENCES

1. Akiki, P.A., Bandara, A.K., and Yu, Y. Using Interpreted Runtime Models for Devising Adaptive User Interfaces of Enterprise Applications. *Proceedings of the 14th International Conference on Enterprise Information Systems*, SciTePress (2012), 72–77.
2. Akiki, P.A., Bandara, A.K., and Yu, Y. RBUIS: Simplifying Enterprise Application User Interfaces through Engineering Role-Based Adaptive Behavior. *Proceedings of the fifth ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ACM (2013), Forthcoming.
3. Akiki, P.A., Bandara, A.K., and Yu, Y. Crowdsourcing User Interface Adaptations for Minimizing the Bloat in Enterprise Applications. *Proceedings of the fifth ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ACM (2013), Forthcoming.
4. Akiki, P.A., Bandara, A.K., and Yu, Y. Cedar Studio: An IDE Supporting Adaptive Model-Driven User Interfaces for Enterprise Applications. *Proceedings of the fifth ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ACM (2013), Forthcoming.
5. Benyon, D. Adaptive systems: A solution to usability problems. *User Modeling and User-Adapted Interaction 3*, 1 Springer (1993), 65–87.
6. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. A Unifying Reference Framework for Multi-Target User Interfaces. *Interacting with Computers 15* Elsevier (2003) 289–308.
7. Clerckx, T., Luyten, K., and Coninx, K. DynaMo-AID: a Design Process and a Runtime Architecture for Dynamic Model-Based User Interface Development. *Proceedings of the 2004 International Conference on Engineering Human Computer Interaction and Interactive Systems*, Springer-Verlag (2004), 11–13.
8. Coutaz, J. User Interface Plasticity: Model Driven Engineering to the Limit! *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ACM (2010), 1–8.
9. Demeure, A., Meskens, J., Luyten, K., and Coninx, K. Design by Example of Graphical User Interfaces adapting to available screen size. In V. Lopez-Jaquero, J.P. Molina, F. Montero and J. Vanderdonckt, eds., *Computer-Aided Design of User Interfaces VI*. Springer-Verlag (2009), 277–282.
10. Feuerstack, S., Blumendorf, M., Schwartze, V., and Albayrak, S. Model-based Layout Generation. *Proceedings of the Working Conference on Advanced Visual Interfaces*, ACM (2008), 217–224.
11. Findlater, L. and McGrenere, J. A Comparison of Static, Adaptive, and Adaptable Menus. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2004), 89–96.
12. Florins, M. and Vanderdonckt, J. Graceful Degradation of User Interfaces as a Design Method for Multiplatform Systems. *Proceedings of the 9th International Conference on Intelligent User Interfaces*, ACM (2004), 140–147.
13. Gajos, K.Z., Weld, D.S., and Wobbrock, J.O. Automatically Generating Personalized User Interfaces with Supple. *Artificial Intelligence 174*, 12-13 Elsevier (2010), 910–950.
14. McGrenere, J., Baecker, R.M., and Booth, K.S. An Evaluation of a Multiple Interface Design Solution for Bloated Software. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2002), 164–170.
15. Meskens, J., Vermeulen, J., Luyten, K., and Coninx, K. Gummy for Multi-Platform User Interface Designs: Shape me, Multiply me, Fix me, Use me. *Proceedings of the Working Conference on Advanced Visual Interfaces*, ACM (2008), 233–240.
16. Myers, B., Hudson, S.E., and Pausch, R. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction 7*, 1 ACM (2000), 3–28.
17. Nebeling, M. and Norrie, M.C. Tools and Architectural Support for Crowdsourced Adaptation of Web Interfaces. *Proceedings of the 11th International Conference on Web Engineering*, Springer-Verlag (2011), 243–257.
18. Nichols, J., Myers, B.A., Higgins, M., et al. Generating Remote Control Interfaces for Complex Appliances. *Proceedings of the 15th annual ACM Symposium on User Interface Software and Technology*, ACM (2002), 161–170.
19. Nichols, J., Myers, B.A., and Litwack, K. Improving Automatic Interface Generation with Smart Templates.

Proceedings of the 9th International Conference on Intelligent User Interfaces, ACM (2004), 286–288.

20. Paternò, F., Mancini, C., and Meniconi, S. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, Chapman & Hall, Ltd. (1997), 362–369.
21. Pleuss, A., Botterweck, G., and Dhungana, D. Integrating Automated Product Derivation and Individual User Interface Design. *Proceedings of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems*, Universitat Duisburg-Essen (2010), 69–76.
22. Raneburger, D., Popp, R., and Vanderdonckt, J. An Automated Layout Approach for Model-Driven WIMP-UI Generation. *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ACM (2012), 91–100.
23. Vanderdonckt, J. and Bodart, F. The "Corpus Ergonomicus": A Comprehensive and Unique Source for Human-Machine Interface. *Proceedings of the 1st International Conference on Applied Ergonomics*, USA Publishing (1996), 162–169.
24. Microsoft Z3Py. http://rise4fun.com/z3py.