

# Behavior-sensitive User Interfaces for Smart Environments

Veit Schwartze, Sebastian Feuerstack, Sahin Albayrak  
DAI-Labor, TU-Berlin  
Ernst-Reuter-Platz 7, D-10587 Berlin  
{Veit.Schwartze, Sebastian.Feuerstack, Sahin.Albayrak}@DAI-Labor.de

**Abstract.** In smart environments interactive assistants can support the user's daily life by being ubiquitously available through any interaction device that is connected to the network. Focusing on graphical interaction, user interfaces are required to be flexible enough to be adapted to the actual context of the user. In this paper we describe an approach, which enables flexible user interface layout adaptations based on the current context of use (e.g. by changing the size of elements to visually highlight the important elements used in a specific situation). In a case study of the "4-star Cooking assistant" application we prove the capability of our system to dynamically adapt a graphical user interface to the current context of use.

**Keywords:** Layouting, model-based user interface development, adaptation, constraint generation, context-of-use, smart environments, human-computer interaction.

## 1 Introduction

Interactive applications, which are deployed to smart environments, are often targeted to support the users in their every-day life by being ubiquitous available and continuously offering support and information based on the users' requirements. Such applications must be able to adapt to different context-of-use scenarios to remain usable for each user's situation. Scenarios include e.g. adapting the user interface seamlessly to various interaction devices or distributing the user interface to a set of devices that the user feels comfortable with in a specific situation. The broad range of possible user interface distributions and the diversity of available interaction devices make a complete specification of each potential context-of-use scenario difficult during the application design. Necessary adaptations require flexible and robust (re-) layouting mechanisms of the user interface and need to consider the underlying tasks and concepts of the application to generate a consistent layout presentation for all states and distributions of the user interface. Based on previous work [12], we propose a constraint-based GUI layout generation that considers the user's behavior and her location in a smart environment. Therefore we concentrate on the user's context and identify several types of possible layout adaptations:

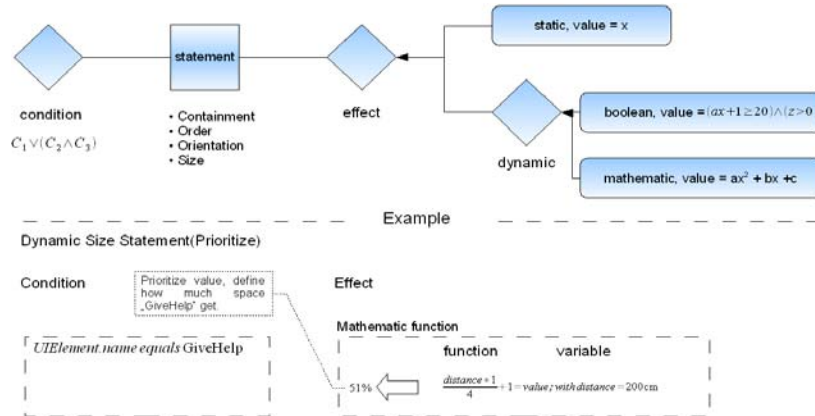
1. *Spot-based adaptation*: In a smart environment, such as our SerCHo Living Lab, different places identify various situations. Applications can consider these spots to adapt their user interface layout to focus on those parts of the UI that are identified as most important for a certain spot.
2. *Distance-based adaptation*: The distance of the user to a certain interaction device, such as a wall-mounted display or a mobile phone, can be used to adapt the layout.
3. *Orientation-based adaptation*: The orientation of the user to an interaction device can influence the presentation of the user interface. Thus, for instance the angle of view in which the user looks at a display can be used to enlarge the visual weight of elements on one side of the user interface presentation.

These adaptations can be done either by discretely or continuously modifying the user interface layout and can be combined for a more comfortable interaction experience. Different to the spot-based adaptation that requires the application developer to explicitly specify, which user tasks are most relevant for a certain user location, the distance- and orientation-based adaptations can be performed without any effort of the designer. In the following, we illustrate the definition and usage of layouting statements to create constraint systems that evaluate runtime context information to adapt the user interface layout accordingly.

## 2. User Interface Layouting

Different to other layout generation approaches [11], we create the constraint system at runtime. In our layout model a user interface is described using four basic characteristics: the containment, the orientation, the order and the size of user interface elements (UI elements). The *containment* characteristic describes the relation of elements as a nested hierarchy by abstract containers that can contain other abstract containers or UI elements. All UI elements are in an *order* that can be defined by relations like “before” or “after”. The *orientation* distinguishes between elements that are oriented horizontally or vertically to each other. Finally the *size* specifies the width and height of containers and UI elements relative to other UI elements or abstract containers. To create a constraint system from these characteristics, we use a set of statements to express the building process. A statement has conditions combined with conjunctions and disjunctions to define the scope of the statement. Conditions can also use additional information about the UI elements to define application independent statements. The formal description of a statement is shown in figure 1 top. If the conditions are fulfilled, the statement is used and the effect modifies the constraint system. At runtime this set of statements is evaluated and creates a constraint system solved by a Cassowary constraint [1] solver. This constraint solver supports linear problems and solves cycles. To generate a flexible constraint system, it also supports a constraint hierarchy using *weak*, *medium*, *strong* and *required* priorities. The Effect is split into *dynamic* and *static*, static statements use only a static value for adaptations; in opposite dynamic statements use a function depending on dynamic information. Dynamic functions are divided into logical- and

mathematical functions. Mathematical functions describe the behavior of their value in dependency to external information<sup>1</sup> like the distance to the screen. Logical functions use external information<sup>2</sup> to create a logical value to come to a decision. This kind of function for instance is used to generate the initial orientation for the elements of the user interface.



**Fig. 1** Statement format and example

The example shown in figure 1 describes a “Prioritize Statement” changing the space allocation for a specific node, in this case for the element “GiveHelp”. The effect contains a mathematic function with the variable “distance”. If the distance between the user and the screen changes, the function recalculates the prioritized value that describes how much space the element “GiveHelp” additionally gets from other UI elements.

### 2.1. Statement Evaluation

The result of a successful layout calculation is a set of elements, each consisting of the location (an absolute x, y coordinate) and a width and height value. The layout generation is performed in three phases:

1. First an initial layout is automatically generated by a set of predefined algorithms that interpret the design models like the task- and abstract user interface model to generate an initial layout that is consistent for all platforms. The result of the containment statement is a tree structure representing the graphical user interface organization. The orientation statement at first allocates the vertical space and after a designer modifiable threshold value is reached, it uses an alternating orientation. After the definition of the orientation the size statement defines the initial space usage for the user interface elements. Basic constraints assure that all additional constraints added do not corrupt the constraint system.

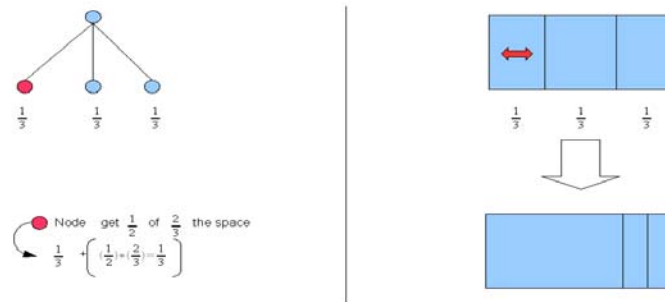
<sup>1</sup> Numerical data from different information sources like the context model.

<sup>2</sup> Comparable data like numerical and textual information.

2. A designer can manipulate the pre-generated layout to match his aesthetical requirements by adding statements that relate information of the design models with a layout characteristic of a UI element.
3. Finally, the user behavior in a smart environment can be considered by adding generic statements that can weight individual UI elements based on the actual context of the user at system runtime.

## 2.2. Context Related-Layout Adaptations

To adapt the interface to specific situations the designer can define context sensitive statements to prioritize specific nodes described in the next section. These statements are only active for specific situations described by context information. Even though our layout model describes the size, order, orientation and containment structure separately, for realizing layout adaptations regarding the user behaviour, we focus on size adaptations, as modifying the other layout characteristics can destroy the user interface consistency, which affects the usability [7]. As we described in the introduction, there are three different statement types: *Spot-based adaptation*, *Orientation-based adaptation*, *Distance-based adaptation*. The basic idea for all adaptations is to highlight the context relevant parts of the user interface for the moment. This is described by a prioritize value characterizing how much additional space an element can use compared to the rest of the interface. The figure below shows an example. The algorithm allocates the space according to the weight (contained elements) so the increase depends on the amount of other elements. In this example we prioritize the red node, the prioritize value of  $\frac{1}{2}$  ensures that the node gets additional space of the other nodes.



**Fig. 2** Result of the prioritizing process

As a result, this statement adds a new constraint with the weight “strong” to the constraint system  $size_{rednode} \geq 2/3 * size_{parentnode}$ . The context based adaptations use static and dynamic statements to recalculate the space allocation for the graphical user interface. The statements, defined by the designer, the prioritize value (static statements) and prioritize function (dynamic statements), used in the next section, are examples and adjustable by the designer. The Spot-based adaptation uses a static prioritize statement for a specific set of nodes and an assigned position of the user. If

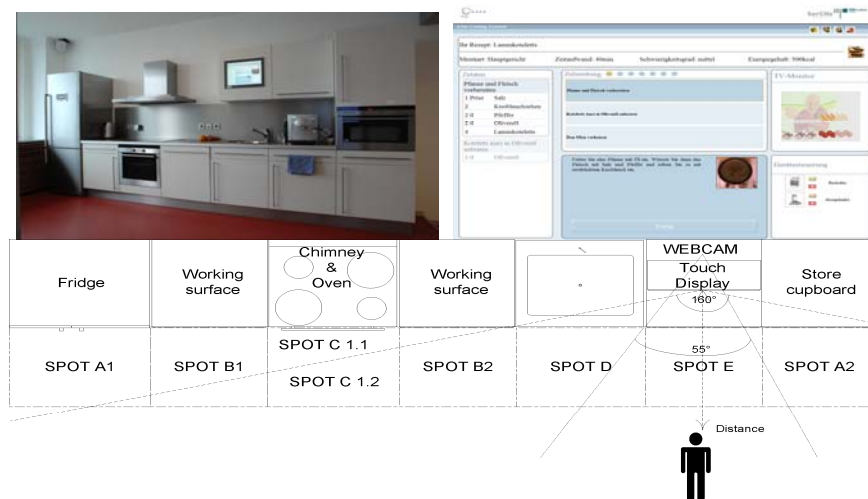
the user reaches the specified position, the statement is used and adds for the affected nodes the amount of space given by the prioritize value.

The orientation-based adaptation uses the Spots “D” and “A2” shown in figure 3 bottom. If the user enters the specified position, this statement is activated and prioritizes a specific node. If the user stands left or right from the screen, this statement prioritizes all nodes with the upper left corner on the opposite side.

The Distance-based adaptation uses the distance from the user to the screen to calculate the prioritize value relative from the distance. If the user moves away from the display, the relevant parts of the interface are enlarged. In the following case study these adaptations are described and discussed.

### 3. Cooking Assistant Case Study

To test the adaptations we deployed the cooking assistant into a real kitchen environment of our SerCHO living lab like depicted by the photo in figure 3 top-left.



**Fig. 3** The kitchen with the cooking assistant running on a touch screen (top-left), the main screen of the cooking assistant (top-right), and the location spots defined by the context model (bottom).

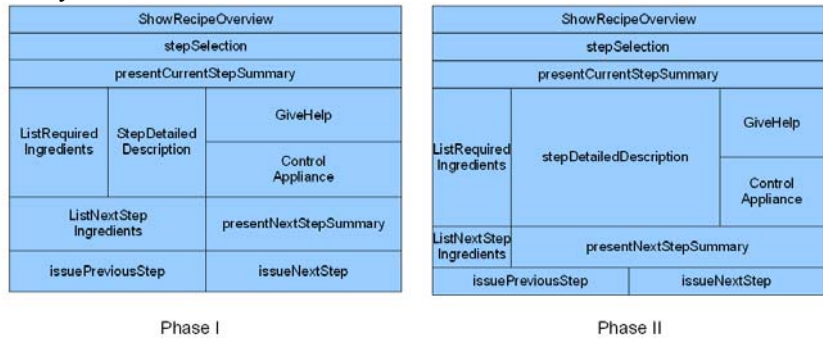
This multimodal application assists the user during the cooking process. The main screen, shown in figure 3, top-right, guides you through the cooking steps and provides help if needed. The figure 3, bottom, illustrates several spots corresponding to the different working positions and user tasks in the kitchen. Since the touch screen supports a view angle of 160 degrees, the user cannot observe the screen from all spots. For the spot-based layouting, we therefore focus on the spots listed in table 1.

Spot	User context	Relevant tasks ordered by Priority.
A2	Looking for ingredients.	1. listRequiredIngredients

C1.2		2. listNextStepIngredients
B2	Preparing ingredients while following the cooking advices and controlling the kitchen appliances.	1. stepDetailedDescription 2. listRequiredIngredients 3. selectAppliance 4. giveHelp
D	Learning about next steps while cleaning dishes after a step has been done.	1. presentNextStepSummary 2. listNextStepIngredients 3. stepSelection
E	Concentrating on the video or getting an overview about the recipe steps.	All tasks same priority

**Tab. 1** An excerpt of the user contexts that are supported by the application. The second column lists the most relevant application tasks for each user tasks.

Figure 4 depicts the box-based preview of our layout editor from which the main screen of the cooking assistant has been derived. By a preceding task analysis, we identified the most relevant interaction tasks. Deriving an initial layout model from a task hierarchy structure has the advantage that related tasks end up in the same boxes and will be laid out close to each other since they share more parent containers the closer they are related.



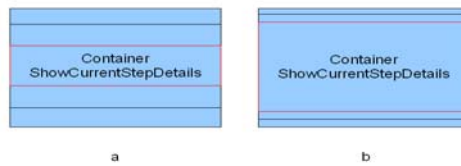
**Fig. 4** Changes from automatic generated layout to designer adapted layout

The starting point for all adaptations is the constraint system generated by the automatic statements shown in figure 4 Phase I and adapted by the designer to adjust the space allocation to his wishes. The result of this process is shown in figure 4 Phase II. To adapt the constraint system to a specific situation, we describe three examples below.

### 3.1. Statements for spot-based adaptation: (B2)

While using the cooking assistant (CA), the user is preparing ingredients, following the cooking advices and controlling the kitchen appliances. Because it is difficult to look at the screen from this position, shown in figure 3 bottom, the statement highlights the important information (Task: *stepDetailedDescription*, *listRequiredIngredients*, *selectAppliance*, *giveHelp*). The condition of the Spot

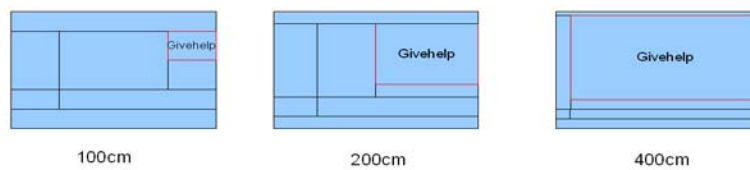
statement is characterized by an environment condition, the position of the user and relevant interactions tasks, as the interface structure is derived from the task model. Because the container “showCurrentStepDetails” contains the most relevant elements it is prioritized. Additionally, the statement use a static prioritize value, defined by the designer. For this study we use a fraction of 4/5(80%) because the prioritization is high enough to support the user, but low enough to follow the changes in the user interface and not confuse the user. The effect of this statement for the case B2 is shown in figure 5.



**Fig. 5** B2 prioritize “ShowCurrentStepDetail” with elements stepDetailedDescription, listRequiredIngredients, selectAppliance and giveHelp

### 3.2. Statements for distance-based adaptation:

While cleaning dishes after a step has been done, the user wants to learn more about the next step. A video helps to understand what has to be done. Because the focused task is specified in the AUI model, the layout algorithm can prioritize the task containing the specific element. The distance statement is characterized by a function calculating the prioritize value depending on the distance to the screen. This function is expressed by  $\text{prioritize value} = ax^2 + bx + c$ . The constants a,b,c can adapted by the designer to match the function to the maximum distance. For our case study we use this linear function:  $\text{prioritize value} = 4/3000^3 * \text{distance}$ . The user interface prioritizing “giveHelp” depending to the distance is shown in figure 6.

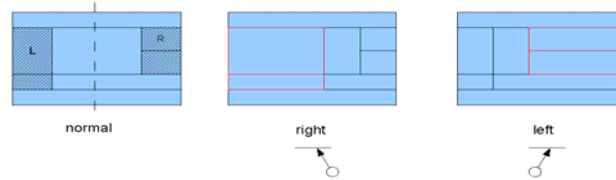


**Fig. 6** Distance based adaptation, shown for 100, 200 and 400cm

### 3.3. Statements for orientation-based adaptation: (A2), (D)

If the user has something to do at the spots A2 and D shown in figure 3 bottom, the view angle to the screen is inappropriate.

<sup>3</sup> This fraction is calculated by the assumption that the interaction space maximum of 600cm, the prioritization for this distance is 4/5(80%)



**Fig. 7** Orientation based adaptation for left- and right side

Depending from the angle of view to the screen shown in figure 7, elements with the upper left corner at the affected side rendered broader than half width of the screen. If the user enters Spot D (left) and leaves the normal angle of view (shown in figure 3 bottom) the width of the elements “giveHelp” and “controlAppliance” is growing to half of the screen width. The same happens if the user enters Spot A2 (right) with the elements “listRequiredIngredients”, “listNextStepIngredients”.

#### 4. Related Work

Nichols et al. list a set of requirements that need to be addressed in order to generate high-quality user interfaces in PUC [5]. As for layout information they propose to not include specific layout information into the models as this first tempts the designers to include too many details into the specification for each considered platform, second delimits the user interface consistency and third might lower the chance of compatibility to future platforms. Different to PUC we are not focusing on control user interfaces, but end up in a domain independent layout model that specifies the containment, the size, the orientation and the order relationships of all individual user interface elements. Therefore we do not want to specify the layout manually for each targeted platform and do not rely on a set of standard elements (like a set of widgets for instance) that has been predefined for each platform.

The SUPPLE system [3] treats interface adaptation as an optimization problem. Therefore SUPPLE focuses on minimizing the user’s effort when controlling the interface by relying on user traces to estimate the effort and to position widgets on the interface. Although in SUPPLE an efficient algorithm to adapt the user interface is presented, it remains questionable if reliable user traces can be generated or estimated. While SUPPLE also uses constraints to describe device and interactor capabilities they present no details about the expressiveness of the constraints and the designers effort in specifying these constraints.

The layout of user interfaces can be described as a linear problem, which can be solved using a constraint solver. The basic idea is shown in [12], this approach uses a grid layout to organize the interface and create a constraint system. Our approach instead uses a tree structure and supports more constraint strengths. Recent research has been done also by Vermeulen [8] implementing the Cassowary algorithm [1], a weak constraint satisfaction algorithm to support user interface adaptation at run-time to different devices. While he demonstrates that constraint satisfaction can be done at run-time, to our knowledge he did not focus on automatic constraint generation.

Other approaches describe the user interface layout as a space usage optimization problem [4], and use geometric constraint solvers, which try to minimize the unused space. Compared to linear constraint solving, geometric constraint solvers require plenty of iterations to solve such a space optimization problem. Beneath performance issues an efficient area usage optimization requires a flexible orientation of the user interface elements, which critically affects the user interface consistency.

Richter [6] has proposed several criteria that need to be maintained when re-layouting a user interface. Machine learning mechanisms can be used to further optimize the layout by eliciting the user's preferences [5]. The Interface Designer and Evaluator (AIDE) [7] and Gadget [2] are incorporating metrics in the user interface design process to evaluate a user interface design.

Both projects focus on criticizing already existing user interface layouts by advising and interactively supporting the designer during the layout optimization process. They follow a descriptive approach by re-evaluating already existing systems with the help of metrics. This is different to our approach that can be directly embedded into a model-based design process (forward engineering).

To adapt user interfaces to a specific situation, in [9] an XSL transformation is used to adapt the abstract description of the interface to the different devices. Our approach follows a model-based user interface design [8]. Following a model-based user interface development involves a developer specifying several models using a model editor. Each abstract model is reified to a more concrete model until the final user interface has been derived. The result is a fine structured user interface, which could be easily adapted to different situations.

An akin approach to create a user interface is presented in [10], the interface structure is derived from the task model and fleshed out by the AUI- and CUI Model. To adapt the interface to mobile devices, different containing patterns are used to organize the information on the screen. Our approach doesn't break the interface structure into small pieces because all information has to be displayed.

## 5. Conclusion and further work

In this paper we presented an approach to adapt the user interface of applications to specific situations. Furthermore our case study "4-Star Cooking Assistant" has shown the relevance to support the user. In the future we have to enlarge the case study to other applications and check more context information about the relevance for GUI adaptations.

*User-interaction-related adaptation:* Based on the user's experiences and his interaction history (tasks completion and referred objects), the most important areas of control can be visually weighted higher to prevent unprofitable interaction cycles or helping the user in cases where he is thinking (too) long about how to interact or to go any further.

*User-abilities-related adaptation:* The layout adapts to the user's stress factor by visually highlighting the most relevant tasks, and takes into account if the user is left or right handed by arranging the most relevant parts of the user interface. Finally his eye-sight capabilities can be used to highlight the most important areas of control.

## 6. References

1. G. J. Badros and A. Borning, The Cassowary linear arithmetic constraint solving algorithm, In ACM Transactions on Computer-Human Interaction, 2001
2. J. Fogarty and S. Hudson, GADGET: A toolkit for optimization-based approaches to interface and display generation, 2003.
3. K. Gajos and D. Weld, SUPPLE: Automatically Generating User interfaces; In: Proceedings of Conference on Intelligent User Interfaces 2004, Maderia, Funchal, Portugal, 2004
4. K. Gajos and D. S. Weld, Preference elicitation for interface optimization, UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology, 2005 New York, NY, USA
5. J. Nichols, Brad A. Myers, Thomas K. Harris, Roni Rosenfeld, Stefanie Shriver, Michael Higgins and Joseph Hughes, Requirements for Automatically Generating Multi-Modal Interfaces for Complex Appliances, IEEE Fourth International Conference on Multimodal Interfaces, Pittsburgh
6. K. Richter, Transformational Consistency, in CADUI'2006 Computer-AIDED Design of User Interface V, 2006
7. A. Sears. Aide: a step toward metric-based interface development tools, pages 101–110, 1995
8. J. Vermeulen, Widget set independent layout management for uiml, Master's thesis, School voor Informatie Technologie Transnationale Universiteit Limburg, 2000
9. Dickson K. W. Chiu and Dan Hong and S. C. Cheung and Eleanna Kafeza, Adapting Ubiquitous Enterprise Services with Context and Views, EDOC '06: Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference, 2006, 391 - 394, Washington, DC, USA
10. Francisco J. Martinez-Ruiz and Jean Vanderdonckt and Jaime Martinez-Ruiz, Context-Aware Generation of User Interface Containers for Mobile Devices, ENC '08: Proceedings of the 2008 Mexican International Conference on Computer Science, 2008, 63 -72, Washington, DC, USA
11. Christof Lutteroth and Robert Strandh and Gerald Weber, Domain Specific High-Level Constraints for User Interface Layout, 2008, 307 - 342, Hingham, USA
12. Sebastian Feuerstack, Marco Blumendorf, Veit Schwartze and Sahin Albayrak, Model-based layout generation, Proceedings of the working conference on Advanced visual interfaces, 2008, Napoli, Italy