Model-based Layout Generation

Sebastian Feuerstack, Marco Blumendorf, Veit Schwartze, Sahin Albayrak

DAI-Labor, TU-Berlin

Ernst-Reuter-Platz 7, D-10587 Berlin

{Sebastian.Feuerstack, Marco.Blumendorf, Veit.Schwartze, Sahin.Albayrak}@DAI-Labor.de

ABSTRACT

Offering user interfaces for interactive applications that are flexible enough to be adapted to various context-of-use scenarios such as supporting different display sizes or addressing various input styles requires an adaptive layout. We describe an approach for layout derivation that is embedded in a model-based user interface generation process. By an interactive and tool-supported process we can efficiently create a layout model that is composed of interpretations of the other design models and is consistent to the application design. By shifting the decision about which interpretations are relevant to support a specific context-of-use scenario from design-time to run-time, we can flexibly adapt the layout to consider new device capabilities, user demands and user interface distributions. We present our run-time environment that is able to evaluate the relevant model layout information to constraints as they are required and to reassemble the user interface parts regarding the updated containment, order, orientation and sizes information of the layout-model. Finally we present results of an evaluation we performed to test the design and run-time efficiency of our model-based layouting approach.

Categories and Subject Descriptors

H.5 [**Information Interfaces and Presentation**]: User interfaces; D.2.2 [**Software Engineering**]: Design Tools and Techniques- *User Interfaces*; H.1.2 [**Models and Principles**]: User/Machine Systems-*Human factors;* H.5.2 [**Information Interfaces and Presentation**]: User Interfaces-g*raphical user interfaces, interaction styles, input devices and strategies, voice I/O*.

General Terms

Design, Human Factors

Keywords

Layouting, model-based user interfaces, constraint generation, context-of-use, human-computer interaction.

1. INTRODUCTION

Interactive applications that are deployed to smart environments must be able to support different context-of-use scenarios. Such 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. Such 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. 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 during the application design impossible.

Specifying the interdependencies between the user interface components using constraints is a common approach to address these issues and nowadays constraint solvers can calculate hundreds of constraints in a reasonable amount of time. To our knowledge there is still an approach missing that supports designers of a user interface in generating these constraints based on the design specifications. A manual constraint setup has two disadvantages: first the pure amount of constraints that is required even to address small interactive systems is hard to handle, and second, the fault tolerance of the constraint setup is complex to attain. Even one single constraint that is not properly specified can destroy the complete layout in a specific situation that has not been considered by the designer during the development process.

This paper introduces an approach for a model-based user interface layouting that differs from previous approaches in two general aspects:

1. We interpret the information from already existing user interface design models, such as the task tree, the dialog model, the abstract user interface model (AUI), the concrete user interface model (CUI), the domain model and the context model for deriving the user interface layout. Therefore we propose an interactive, tool-supported process that reduces the amount of information that needs to be specified for the layout. The tool enables designers to comfortably define design model interpretations by specifying statements and subsequently applying them to all screens of the user interface.
2. We shift the decision about which of the statements are applied from design-time to run-time to enable flexible context-of-use adaptations of the user interface layout. This allows us to describe new context-of-use adaptations of the layout without the need to change the application itself just by describing the layout characteristics of a new platform or a new user profile.

The next section discusses the related work that has been considered to support our approach. Section 3 presents the layout model and its relation to the other user interface design models. Section 4 describes our approach to generate constraints based on the interpretation of design models by an interactive, tool-supported process. Section 5 presents our implementation, integrating a layout model agent into our run-time-environment (MASP) [2,5]. Section 6 discusses results of an evaluation we did to test the efficiency of the model-based layout process at design-time by measuring the performance of the constraint generation and constraint solving at run-time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AVI'08, 28-30 May , 2008, Napoli, Italy

Copyright 2008 ACM 1-978-60558-141-5...$5.00

Finally Section 7 summarizes the paper and outlines future work.

2. RELATED WORK

Nichols et al. lists in PUC [10] a set of requirements that need to be addressed in order to generate high-quality user interfaces. 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 [7] 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. Recent research has been done by Vermeulen [16] 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 [8], 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 [13] 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 [9]. The Interface Designer and Evaluator (AIDE) [14] and Gadget [6] 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).

In the next chapter we present our approach for a layout model, that is designed to be part of a model-based user interface design approach [15] like proposed by the Cameleon Reference Framework [3]. Following a model-based user interface development involves a developer specifying several models using a model editor (such as a task model, a domain model, and a dialog model). Each abstract model is reificated to more concrete models until the final user interface has been derived.

3. THE LAYOUTING MODEL

Like illustrated by figure 1 our layouting model is part of such a model-based user interface design process. To derive a layout model the designer has to specify interpretations of the design models by defining layout statements. In general two different statements are possible: First, layout statements that are explicitly specified for one user interface and second, layout statements that are defined independent of the user interface. The latter interprets pre-defined context information to address layout adaptations for specific devices and users or specific environments. Currently we are focusing on interpreting the context, task tree, AUI and dialog models to derive layout information.

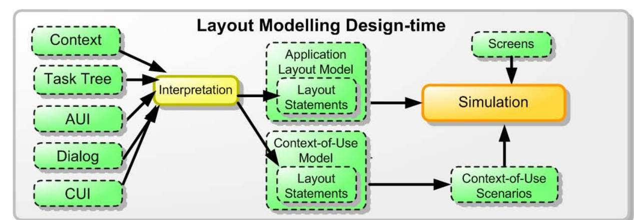


Figure 1: The layouting process is embedded into a model-based user interface design process.

For each new layout statement that is written into the layout model, the designer can initiate a simulation to preview the result. The simulation positions the individual user interface elements based on the specified layout model statements for all screens and context-of-use scenarios that are known at design-time.

Our layout model basically consists of a list of ordered statements. Like illustrated by figure 2, each statement is composed of six properties: the characteristic of the resulting layout primary addressed (containment, orientation, and size) (3.1), the design models used for the constraint generation (3.2), the context-of-use information (3.3), the addressed scope (3.4), the type of condition (3.5) and finally the priority value (3.6). In the following sections we describe these constituent parts of a layout statement in greater detail.

3.1 Layout Characteristics

We identified four of these characteristics that can be used to specify the layout of a graphical user interface: The *containment*, the *order*, the *orientation* and the *size* of the user interface elements.

Like illustrated by figure 3, the *containment* describes the relation between two basic types of entities: Containers (like *c1*) consist of a set of nested containers (*c2*+*c3*) and nested elements (*c2* contains *e1* and *e2*). Elements can present information to the user or enable the user to enter data to the application and cannot be decomposed any further. Additionally a layout describes an *order* of elements (e.g. from left to right and from top to bottom: *e1* before *e2* and *c2* before *c3)*. The *orientation* distinguishes between elements that are oriented horizontally or vertically to each other (e.g. *e1* vertical to *e2*). Finally the *size* specifies the width and height of containers and elements (e.g. the width of *e3* is ½ of the width of *e4*).

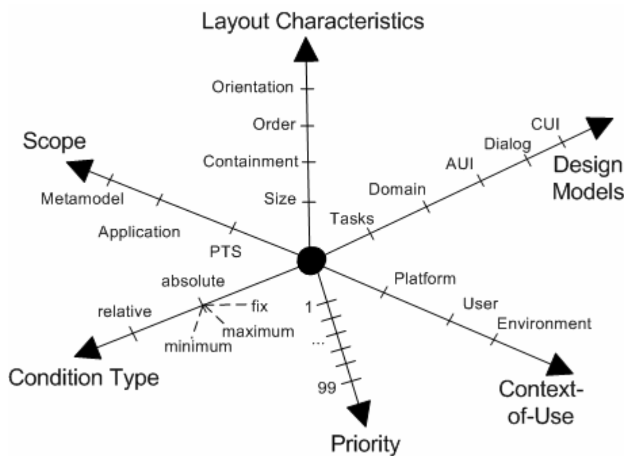


Figure 2: The six axes of the space of properties of the layout statements.

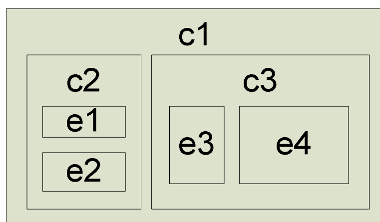


Figure 3: Exemplary sketch of a user interface layout.

3.2 Design Models Interpretation

Design Models are used to specify the interactive system on different levels of abstraction. We interpret the information of these models to derive the user interface layout. A task model, a domain model, an abstract user interface, and the dialog model are typically part of a model-based user interface design. Beneath the task model's hierarchical structure that can be used to derive a basic containment structure for the layout [5] other information can be derived: For instance, the sum of all atomic tasks related to the task tree depth or related to its width can be used to balance the presentation size of the tasks. For instance the CTT notation [12] categorizes interaction tasks into "edit", "control" and "selection" tasks. This task information can be addressed differently related to the context-of-use, for instance by prioritizing those tasks that require user input.

Looking at the abstract user interface model, the interaction object type can be used by a layouting statement to derive an orientation. E.g. navigational elements can be set vertically or horizontally depending on the menu level, whereas selection elements can be oriented vertically for a large amount of elements and horizontally for small amounts by a layouting statement.

3.3 Condition Type

Each statement describes either an absolute condition (minimum, maximum, or fixed) or a relative condition that relates two or more elements. A relative condition targeted to the orientation characteristic is for instance: "e1 over e2", regarding the size a relative condition can specify e.g. "e4 double the width of e3" and finally regarding the containment it has the form of "c3 contains e4". A maximum statement containing an absolute condition can be used to specify a column layout where elements are wrapped to the next row after a specified amount of elements is exceeded. Further on, a maximum statement can restrict the size (regarding its height or width) or the number of elements in a container. If the size limit is exceeded new containers are generated.

3.4 Application Scope

Each statement has a fixed scope to address every application (application independent statements), the whole application, a set of reoccurring elements or a specific screen to handle very fine grained design requirements. Application independent statements are used to characterize context-of-use adaptations that are required to be considered when layouting for a specific device (such as specifying the screen size limitation, or the minimum size of control buttons for a touch screen). Application wide statements help the designer to generalize design decisions and maintain consistency as layouting decisions can be modeled just once and are automatically applied for each reoccurring situation. The more global than local statements have been defined the better is the robustness for context-of-uses changes and the better layout consistency can be expected. Finally a statement can be limited to address a single screen to fine tune the layout for aesthetical reasons or to refine an application wide layout statement.

3.5 Context–of-Use Scope

The context-of-use describes the user, who has preferences and demands for the actual situation, a set of devices that she likes to use in a certain environment. A layout statement can be specified to be relevant for a specific context-of-use situation only. For instance in an environment that supports location tracking, the distance of the user to a device can be used to scale the control elements of the user interface. In the former case the control elements are sized small if the user has no way to control because of his distance to the display, whereas in the latter case the control tasks are sized to meet a pen or a finger print respectively.

3.6 Strict Order by Priority

Specifying the priority of a statement is required on the one hand to support a general-to-specific layouting approach and on the other hand to prevent the generation of conflicting layout constraints. Thus, general layouting principles, as described in style-guidelines or given by a corporate design can be generally defined and overwritten to address more specific situations later on. We address these aspects by specifying a strict order in that the layout statements are evaluated to generate the constraints that we indicate by the priority property.

3.7 Conclusions

Deriving an interface layout based on the design models of a model-based interface development approach results in a consistent layout. Further on, such a layout model derivation reduces the information that has to be specified for the interface layout as a lot of information is already available in the design models. The more global application layout statements can be derived from the design models the better robustness of the interface to unknown context-of-use changes can be expected.

To realize such a model-based layout generation that is based on model interpretation we require (1) an efficient way for the designer to select suitable model interpretations for generating a layout (2) a process that eases the identification of global interpretations to enforce the layout's consistency and robustness against context-of-use changes and finally (3) a model-based run-time system that can evaluate these interpretations in an efficient manner so that layouting adaptation of a user interface is possible at run-time.

We introduce our model-based layout editor in the next section that we implemented to address requirements (1) and (2), and describe how we realized the layouting in our run-time environment, the Multi-Access Service Platform (MASP), to adapt to context changes (3).

4. LAYOUT MODEL GENERATOR

Using the layout model generator the designer has to initially load all design models of an interactive application as well as already known contexts-of-use scenarios that contain device capability descriptions and the preferences of the user.

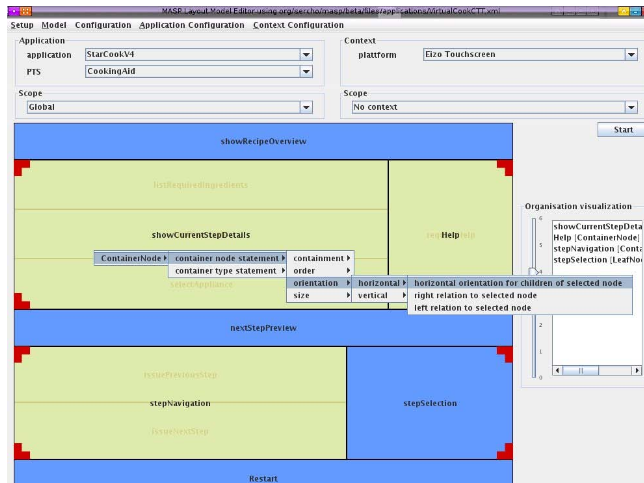


Figure 4: The MASP Layout Model Generator

Figure 4 shows a screenshot of the editor: Using the pull down menu in the upper left corner ("PTS"), the designer can browse through all screens of the application. Each screen consists of a set of elements that should be presented simultaneously to the user on a single device. Predefined interface distributions that allow to one screen to several devices, each containing complementary parts of the interface can be defined as a set of separate screens with the same context-of-use

The result of the layouting process, the layout model is visualized by a box-based layout that represents each individual user interface element that is part of a screen as a box. By the box-based layout the designer gets an impression of the layouting results concerning the individual elements size, containment, order and orientation relationships. Different to the layout result that is calculated during run-time and ends up with absolute coordinates for each box, the simulator linearly scales the preview s but considers the aspect ratio of the targeted device in order to comfortably support layout modeling for large display.

Using the layout editor, the designer specifies all layout statements by using a context menu that is related to the box-based simulation area. The application scope (global, application or screen specific), and the context-of-use of a statement can be set by two separate pull down menus above the simulation area.

The process of deriving layout statement is supported by the tool following several subsequent steps:

1. The designer decides about the layout characterization that the statement should address: the containment structure, the element order, the orientation or the size.
2. The designer defined a new layout statement that interprets one or more
    a. design model information (such as the AUI type: input, output, control, or selection task or the CUI type)
    b. context model information that require a layout adaptation.
3. The designer can visually weight a relational statement. E.g. relate the size-ratio between input and output elements in general or specify size relations between two specific boxes.
4. The Model Generator automatically applies the new statement consistent to the design models to all screens of the applications (limited by the scope of the statement).
5. The Model Generator updates the boxed simulation area to reflect the new layout for all screens and all actually supported context-of-use scenarios of the user interface layout.
6. The designer checks the result and manipulates the order of the statements.

In order to ease the identification of global layout statements to force the layout consistency and robustness against context-of-use changes (requirement 2), we implemented an abstract-to-detail slider, which is depicted to the right in figure 4. The slider allows the designer to browse through the nested boxes by moving the slider up and down starting from the box that contains the whole application, to the atomic elements that describe individual user interface widgets. Following such an abstract-to-detail layout modeling, the designer is supported to start specifying statements on the highest abstraction level possible. The editor visualizes atomic elements in blue and boxes that contain nested elements through a yellow overlay like depicted in figure 4.

To prevent specifying conflicting statements the designer is allowed only to define relational statements between elements that have been specified on the same nesting level (which corresponds to the abstraction level of the task tree if the task model has been used to derive the containment). In the editor, we use the red corners to indicate elements that are located on the same nesting level and thus can be target of a relational statement. For instance in figure 4 the red corners indicate two separate boxes of an exemplary application that are not directly related: The upper one highlights the two boxes "showCurrentStepDetails" and "Help" whereas the lower one consists of one box "stepNavigation" and one individual element "stepSelection". In this case the designer has the option to define an interpretation for the relation between "showCurrentStepDetails" and "Help" but not the option to specify a direct relation containing elements of the upper and the lower box (since such a relation has to be set on a higher level of abstraction which contains both boxes).

Each statement that has been defined is written into the layout model and gets instantly evaluated to a set of constraints that is solved to update the box-based preview. This process happens without any remarkable delay so that we can recalculate the constraints on the fly to give an instant visual feedback.

Figure 5 presents a screenshot of the editor's view of the layout model. The layout statements are grouped by the layout characteristic they are primarily addressing. In case conflicting constraint sets have been generated the last statement that the designer has entered and the one that caused the conflict is highlighted red.

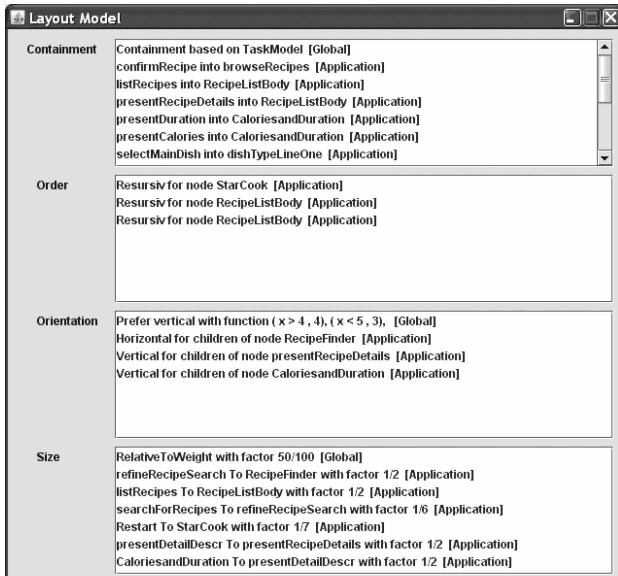


Figure 5: The actual layout model consisting of a set of statements that are grouped by the layout characteristic they are targeting to.

Not all of the four layout characteristics can be handled independently from each other. First, the containment constraints the order, orientation and size characteristics and second, the element order constraints the orientation and the element size. To manage these interdependencies we define a general order in which the statements are processed based on the layout characteristic they are mainly addressing: Like depicted by the screenshot in figure 5 we process the containment-related statements before the order-related ones Thereafter the orientation-related statements and finally the size -related statements are processed.

After a suitable set of constraint generating functions has been identified, the designer can check the resulting layout for its adaptivity to manage certain context-of-use scenarios by browsing through a set of predefined contexts-of-use. Predefined contexts-of-use contain further context-specific layout statements that have been specified independent from a certain application and are reflecting the capabilities of a device or the preferences of a user that are already known at design-time. Like illustrated in figure 1 the layout statements of predefined contexts-of-use are merged to the layout statements of the application to simulate the user interface layout. In the following section we describe how the layout statements are evaluated to constraints in our run-time environment.

5. CONSTRAINT GENERATION AT RUN-TIME

Following the idea of using software agents to coordinate the user interface management system [3] we are using an agent-based run-time environment, the Multi-Access Service Platform (MASP) to generate and adapt user interfaces. But instead of requiring a hierarchical organization to several agents like proposed by PAC-Amodeus [11], the communication flow between the agents in our environment can be flexibly configured based on the requirements of the interactive application. As illustrated by figure 6 the environment is driven by several agents where each interprets one user interface model. In contrast to other approaches [3,15] that refine a user interface model at design-time to end up with a compiled version of the user interface, we keep all of the models alive at run-time. This allows us to more flexibly react to context-of-use changes that have not been desired at design-time by specifying the required adaptation on an abstract model-level.

Each agent is comprised of two parts: a tuple space to store the instantiated model information and a manager containing the semantics and functionality to manipulate the model information. Whereas the manager has complete access to its own tuple space it is not aware of the other agents connected to the system. We connect the agents by using tuple space operations (atomic read/manipulate/write) and the eventing system of a tuple space. The eventing system allows a manager to register for changes of another tuple space. Each agent, handling one user interface model is instantiated once to run a single application, but is able to handle several sessions for different users that are accessing the same application.

The communication processes between all agents are not hard wired but instead configured for each application based on the user interface models that are relevant for the applications domain. Therefore we can easily add the layouting model agent as an additional component to the MASP..

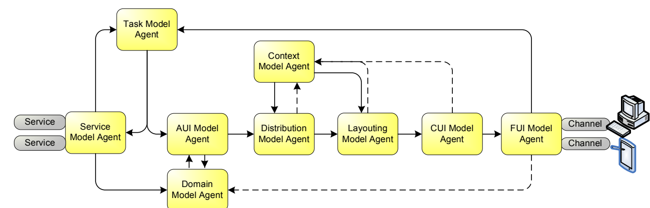
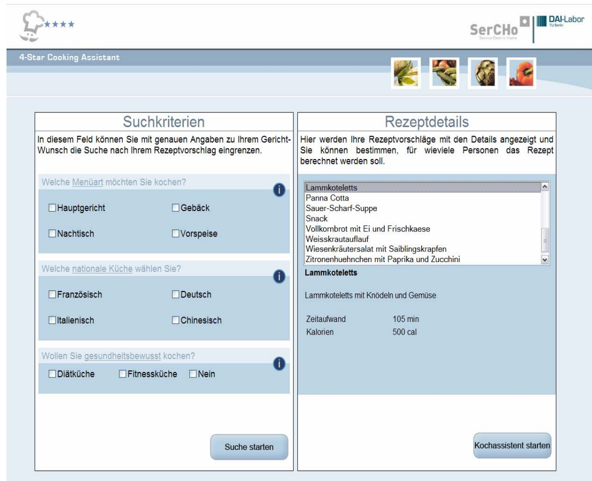


Figure 6: The layouting model is embedded as an agent into our run-time environment.

As illustrated by figure 6, the layouting agent registers itself for events from the distribution agent, which calculates the distribution of a presentation task set to all platforms that are connected to the MASP. For each new or updated user interface distribution the layouting agent receives an event containing all the elements of the user interface that should be simultaneously presented on a specific platform. While the distribution agent is required to calculate a reasonable user interface distribution based on the actual context of use, the layout agent has to layout a presentation for all the individual elements it receives from the distribution agent for a single device.



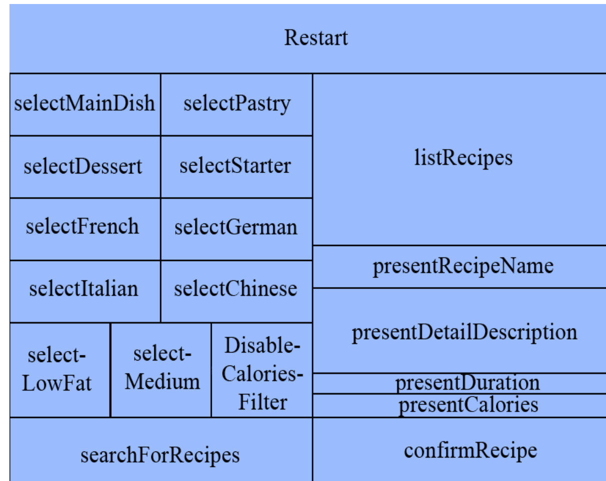


Figure 8: The screen for the recipe search and the final box-based layout result of the layout-model

As soon as such an event from the Distribution Model Agent has been received the Layout Model Agent reads the actual context-of-use and evaluates all of the layouting statements that are relevant for the actual user interface screen.

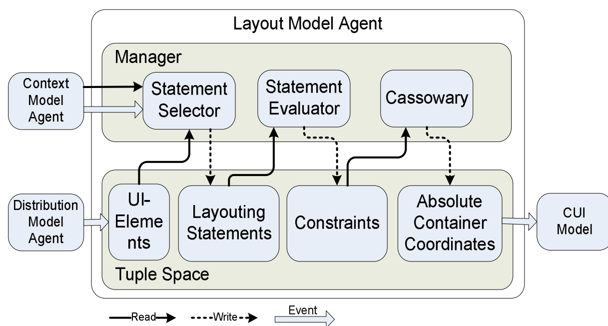


Figure 7: Each Model Agent is comprised of a manager that encapsulates the agent's functionality and a tuple space to store its data.

Figure 7 depicts the internal setup of a Layout Model Agent and its internal as well as its external communication. The agent senses for two external events to happen: First, for a new distribution of the user interface and second, for a change of the context-of-use. Both stimulate the agent to select and assemble the layouting statements. The selection of suitable statements is done by the following way:

1. Retrieve layouting statements for the actual context of use that have been specified independently from the application and that specify layout requirements to address a certain user or a specific device.
2. From the ordered statement list select the statements for a screen *s* that:
    a. address application wide layout interpretation
    b. address reoccurring elements that are used by *s*
    c. directly address the screen *s*
    d. are defined for this application and the relevant context-of-use scenario.

The statements that have been selected and ordered by priority are then evaluated to a set of constraints by the statement evaluator. Thereafter the layout agent finally solves the new constraint setup using the cassowary constraint solver [1]. Solving the constraints results in absolute positions for each element of the user interface that are stored within the layouting agent's own tuple space. The CUI Model Agent is registered for updates to the absolute positions and therefore receives updates for each change of these coordinates that the CUI Model Agent will use to re-position the user interface elements.

Different to other approaches that use a constraint solver to calculate the user interface layout, we introduced an additional level of abstraction for defining the user interface layout by a separate layout model that includes statements that are derived using an interactive and tool-supported process and are consistent to the other user-interface models. Since we decide at run-time which statements to evaluate to generate constraints, we can flexibly address layout adaptations to new contexts-of-use scenarios that can even be independently specified from an application but have been introduced together with a new device or a new kind of user type.

In the next chapter we present first results of an evaluation we did to test the efficiency of our approach. The evaluation has been done as part of a research project where we realized a multi-modal cooking assistant that supports the user in finding recipes, creating a shopping list and guides the user step by step through the cooking process.

6. EVALUATION

We tested our approach regarding two aspects: first, the efficiency at design-time for the designer to generate the layout model by using the layout model generator. Second we tested the efficiency of the implementation to generate and solve the constraints in our run-time system.

6.1 Design Efficiency

To test the design efficiency of the approach, we asked a designer to realize a layout for an interactive cooking assistant application based on a textual description of a scenario of how the cooking assistant should support the user. The designer created three screens and one user interface distribution scenario where one screen is split to two different devices: The initial screen asks the user to search for a recipe based on several search options. The second screen is about assisting the user to generate a shopping list by asking the user which of the

required ingredients are available and which are not available. This screen could be split into two parts where one part gets distributed onto a PDA that could be taken along during shopping and the other part remains on a touch screen in the kitchen. The last screen assists the user during cooking by offering multi-medial help, controlling the kitchen appliances and by splitting each recipe into a list of steps containing the required ingredients as well as a detailed description about what to do in each step. Figure 8 presents the initial screen for the recipe search as it has been realized by the designer and the result of the model-based layouting using the box-based layout of the editor.

Independently from the designer we asked a developer to follow a model-based development approach. Initially both, the designer and the developer shared the same textual description of a scenario for the cooking assistant. Based on the results of the model-based development approach including a fine grained task model, a domain model and an AUI model, we then derived a model-based layout that should correspond to the screens of the designer as close as possible. Finally we measured the amount of statements that have been required to end up with the same layout as the designer has realized.

Each screen has a different layout complexity consisting of a number of elements that are nested based on the abstraction level of the task model.

Screen	1)	2)	3)	4)	5)	6)	7)
1.Recipe Search	19	7	9	3	3	4	19
2.Shopping List	13	8	0	6	1	2	12
3. Distribution: PDA,Touch	4,9	8	0	1,2	0	0	0
3.Cooking Aid	15	10	0	2	2	4	8

Table 1: Complexity of the screens that need to be layouted and the amount of statements required. 1) Elements to layout, 2) Abstraction levels 3) Number of containment-, 4) orientation-, 5) order- 6) site-related statements 7) total amount of statements

Table 1 lists the level of complexity (number of elements, and the maximum nesting level utilized) for the three screens that have been sequentially layouted and the amount of statements that were required to realize the layout of the designer. After the first screen has been layouted the derived statements have been reapplied to the second screen and finally to the third screen. The second column of table 1 lists the different levels of UI-complexities that we have considered by the three screens: Whereas the RecipeFinder screen has a lot of elements (19) and a less nested structure of 7 levels, the Cooking Aid screen has 15 elements on 10 nesting levels as it is composed of various parts that are not directly related (e.g. the multi-medial help and the appliance control). For the ShoppingList screen two further layouts have been designed that are reflecting a distribution scenario where parts of the screen get distributed to a PDA (4 elements) and some parts (9 elements) remain on the screen. By analyzing the amount and type of statements that were required to layout the screens in the same way like the designer did, several observations have been made and are listed in the following paragraphs:

- Containment and order related statements can be derived from a task tree efficiently.
- If the task model is used to derive the containment and atomic tasks are identical to individual widgets, the introduction of further containment-related statements is required (for our application we required 8 containment statements for grouping checkboxes for the recipe search screen).
- Size related statements can be defined very efficiently on an application wide, global level based on the information of the design models (such as weighting input to output tasks, or by giving control tasks that usually end up presented as buttons a global minimum /maximum size restriction).
- The aspect ratio has to be defined pictures that should be presented within a task (using a size relational statement) which can be automatically derived at run-time when loading the picture.
- The orientation related statements can only be very limited specified on a global level but have to be reapplied for most of the individual screens. This is because our design models have no information that can be used to derive an initial orientation. So we applied a heuristic approach that produces elements with a balanced width to height relation by switching the orientation of the elements. Therefore we toggle the orientation horizontal to vertical and vice-versa, for each nesting level that has been derived from the task model.
- The container, order and size related statements of the layout model helped to assemble layouts for user interface distributions that have not been explicitly addressed at design-time. Orientation related statements caused problems as after a distribution has been initiated the re-orientation of the remaining user interface parts were not expected by the users.

6.2 Efficiency at Run-time

In order to check the run-time performance of generating and solving the constraints, we measured the performance of both the statement evaluation and the constraint solving separately.

Screen	1)	2)	3)	4)
1.Recipe Search	25	142	<1ms	14 ms
2.Shopping List	20	107	<1ms	8 ms
3. Distribution PDA,Touch	8,10	56,81	<1ms	8,10ms
4. Cooking Aid	23	130	<1ms	13 ms

Table 2: Complexity of the screens the need to be layouted and the amount of statements required. 1) Number of statements to evaluate, 2) Number of evaluated constraints 3) Measurement for statement evaluation (ms) 4) Duration for constraint solving (ms)

Table 2 shows the results of the performance evaluation for our cooking assistant application. For each screen we have measured the amount of statements that have been selected as relevant for layouting each screen (second column) and the amount of constraints that have been generated by evaluating the selected statements. It could be observed that currently an average of 5 to 7 constraints is generated by one statement. In the last two columns the measured average calculation time (of three runs) for selecting the required statements and the duration

for solving the generated constraints are listed: We could observe that the time to choose between the statements that are relevant for a specific situation was always under 1ms and the amount of constraints (and the amount of selected relevant statements) is related to the solving time. Thus, the bigger the difference between the overall number of layouting statements and the number of selected statements the shorter constraint solving times can be expected.

7. CONCLUSION

The information of the design models of a a model-based interface design approach can be interpreted to derive a layout model. We describe these interpretations by statements that create a layout model that we evaluate at run-time. This approach offers two advantages: First, since the statements are interpreting the design models, they ensure a consistent user interface layout and second, as the statements are evaluated at run-time, they enable flexible context-of-use adaptations even to situations that have not been directly considered during application design.

We are currently investigating further evaluations as the initially evaluation data is based on a relatively small application. Although we initially hoped to identify a set of predefined layout derivations based on preexisting design models that can be generally applied for all applications, we are now trying to classify application types and try to figure out if we can support the layout designer by proposing different statement sets based on the application type.

8. ACKNOWLEDGMENTS

We thank the German Federal Ministry of Economics and Technology for supporting our work as part of the Service Centric Home project in the Next Generation Media program.

9. REFERENCES

1. G. J. Badros and A. Borning; The Cassowary linear arithmetic constraint solving algorithm; In ACM Transactions on Computer-Human Interaction, 2001
2. M. Blumendorf, S. Feuerstack, S. Albayrak; Multimodal User Interfaces for Smart Environments: The Multi-Access Service Platform; Accepted as demo paper for ACM Advanced Visual Interfaces Conference 2008; Napoli, Italy
3. G. Calvary. et all; A unifying reference framework for multi-target user interfaces. In: Interacting with Computers, Vol. 15, No. 3. pp. 289-308, 2003.
4. J. Coutaz; PAC: An object oriented model for implementing user interfaces; In:SIGCHI Bull., vol. 19, no. 2, pp. 37--41, 1987
5. S. Feuerstack, M. Blumendorf, S. Albayrak; Prototyping of Multimodal Interactions for Smart Environments based on Task Model; Workshop on Model Driven Software Engineering for Ambient Intelligence Applications, European Conference an Ambient Intelligence 2007, Darmstadt, Germany.
6. J. Fogarty and S. Hudson; GADGET: A toolkit for optimization-based approaches to interface and display generation, 2003.
7. K. Gajos and D.Weld; SUPPLE: Automatically Generating User Interfaces; In: Proceedings of Conference on Intelligent User Interfaces 2004, Maderia, Funchal, Portugal; pp. 93-100, 2004
8. H. Hosobe (2001), A modular geometric constraint solver for user interface applications, *in* 'UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology', ACM Press, New York, NY, USA, pp. 91—100
9. 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
10. 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, PA, Oct 14-16, 2002a. pp. 377-382
11. L. Nigay and J. Coutaz, Formal Methods in Human Computer Interaction, Ch. Software architecture modelling: bridging two worlds using ergonomics and software properties, Springer Verlag, pp. pages 49-73, 1997
12. F. Paterno: Model-based Design and Evaluation of Interactive Applications. Springer Verlag. Berlin 1999.
13. K. Richter (2006), Transformational Consistency, *in* 'CADUI'2006 Computer-AIDED Design of User Interface V'.
14. A. Sears. Aide: a step toward metric-based interface development tools, pages 101–110, 1995
15. J. Vanderdonckt; P. Berquin, "Towards a very large model-based approach for user interface development," User Interfaces to Data Intensive Systems, 1999. Proceedings , vol., no., pp.76-85, 1999
16. J. Vermeulen, Widget set independent layout management for uiml, Master's thesis, School voor Informatie Technologie Transnationale Universiteit Limburg, 20