Migratable User Interface Descriptions in Component-Based Development

Kris Luyten, Chris Vandervelpen, Karin Coninx

Expertise Centre for Digital Media
Limburgs Universitair Centrum
Wetenschapspark 2
B-3590 Diepenbeek-Belgium
{kris.luyten, chris.vandervelpen, karin.coninx}@luc.ac.be

Abstract. In this paper we describe how a component-based approach can be combined with a user interface (UI) description language to get more flexible and adaptable UIs for embedded systems and mobile computing devices. We envision a new approach for building adaptable user interfaces for embedded systems, which can migrate from one device to another. Adaptability to the device constraints is especially important for adding reusability and extensibility to UIs for embedded systems: this way they are ready to keep pace with new technologies.

1 Introduction

The market of embedded systems and mobile computing devices is a fast evolving market. New technologies are introduced at a very high rate. One of the consequences of this evolution is the constant reinvention of user interfaces (UIs) for these devices. They lack the adaptability and flexibility to be deployed for new devices (possibly using new interaction techniques) without reprogramming them. One of the results of the SEESCOA[1] [13] project is a common software platform, using components for embedded systems on a Java Virtual Machine. Using this specific component-based approach for embedded systems, we can develop a framework for UIs adapting to the environment and device specific constraints as well as encourage reuse. The SEESCOA method is a component-based development approach combined with ideas of contract-based specification for software objects.

This paper presents our ongoing research on the possibility of creating a framework that will allow for runtime migratable UIs, which are independent of the target software platform, the target device and the interaction modalities. These UIs are merely considered as a presentation of a single service or of more

[1] Software Engineering for Embedded Systems using a Component-Oriented Approach, `http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/SEESCOA/`

functionally grouped services. We try to extend the work presented in [3, 14, 10] which all focus on how to abstract a UI for a platform- and device-independent usage. Like work presented in [9, 6, 2], we also use markup languages to describe UIs. However, our work goes a step further by allowing runtime generation of UIs using a markup language. These ideas are combined with a component-based approach allowing the designer to design UIs for particular components, which can be merged automatically at a later stage. This enables UI designers to concentrate on what is important for multi-device UIs: how to present the UI in a structured and logical manner. Unlike approaches like described in [10], we try to develop a truly distributed component-based approach, without relying on a client-server architecture.

Throughout the text we will use an example case study: a small camera surveillance system using 4 cameras. Each camera will be represented by a component. It will be possible to combine the four cameras by using a Mosaic component. This should make it possible to observe four cameras at the same time. Each camera has its own properties: some cameras can zoom in and out, other also allow to change the framerate,. . .

The next section, section 2, takes a look at how UIs for embedded systems or mobile computing devices can be described with a UI description language. An overview of related work is provided. Continuing with section 3, we show how these descriptions can be combined with software components in general, and SEESCOA components in particular. The case-study is presented in more detail to show the results of the approach proposed in this paper. In section 4, we consider how using markup languages and a component-based approach contributes to flexibility, adaptability and migratability of UIs. In particular attention is given to automatic layout management and multi-modal rendering possibilities. Finally conclusions with regard to the current work and possible extensions are formulated in section 6.

2 Describing User Interfaces For Embedded Systems

2.1 Abstracting the User Interface

When designing UIs for embedded systems, we should not take a widget-based approach, but an interaction- or task-based approach. We should be interested in how a user can interact with the offered service and how this can be instantiated afterwards using a concrete widget set. This kind of approach is thoroughly examined in [11] and is important in particular for embedded devices. Too much time is spent reinventing UIs for accessing the same services as technology evolves. One of the major enhancements we envision is the separation of UI design and low-level programming. Until now, embedded systems programmers have a dual task: implementing the actual embedded system and designing and implementing the UI for this system. The main reason for this way of working is the required technical knowledge and background of the system to provide a UI for it. Therefore we use a markup language to describe the UI for embedded systems and mobile computing devices.

2.2 An XML-Based User Interface description

To describe a UI on a sufficiently abstract level the eXtensible Markup Language (XML)[5] is used. Listing 1.1 provides an example of how a UI can be described in XML. There are already several propositions and real world examples of the usage of XML to describe UIs: [10, 1]. A list of advantages is given in [8]. One of the major advantages is that XML does not force any level of abstraction, so this level can be adapted to the requirements of the situation. Note that an XML document can be presented as a tree which turns out to be a great advantage in our approach. There are other approaches for describing User Interfaces, but we believe that an XML-based description offers the best solution in our component-based approach because of it heavily relies on hierarchical structures.

Listing 1.1. An example XML listing for a camera

```
<ui>
<title>Login</title>
<group name="videopanel">
  <interactor>
    <video name="video">
      <text>Camera 2 video stream</text>
      <mediasource>http://twiki.luc.ac.be/camera:8888</mediasource>
    </video>
  </interactor>
  <interactor>
    <range name="zoomrange">
      <text>Zoom</text>
      <min>-100</min>
      <max>100</max>
      <start>0</start>
      <tick>25</tick>
      <action>
        <func service="Mosaic.camera2">setZoom</func>
        <param name="zoomrange"/>
      </action>
   </range>
  </interactor>
  <interactor>
    <range name="focusrange">
      <text>Focus</text>
      ...
      <action>
      <func service="Mosaic.camera2">setFocus</func>
        <param name="focusrange"/>
      </action>
    </range>
  </interactor>
  <interactor>
    <button name="snapshot">
      <text>Take snapshot</text>
      <action><func service="Mosaic.camera2">saveImage</func></action>
    </button>
  </interactor>
</group>
</ui>
```

The example listing (listing 1.1) is *not* simplified: the UI description is meant to be human-readable and machine-processable at the same time. The description allows human users to specify the UI on a high level.

On the other hand, the structured and hierarchical approach by using XML as a notational language to describe the UI allows machines to process and use these descriptions without human intervention. Our notation uses a range of tags that are easy to read and understand for humans. In the current stage, a stable Document Type Definition or XML Schema is not available because we do not consider our specification to be complete. Nevertheless care has been taken to introduce no ambiguities in the specification and to enable easy migration to other specification languages, in case a certain XML-based notation for describing UIs will evolve into a standard.

The following interactors are currently supported by the system: range interactors, single and multiple choice interactors, a text interactor, push interactors (e.g. a button) and a canvas output interactor (e.g. a video stream). These can be composed to represent a new interactor with combined functionality. The available tags are still limited, but a lot of dialog-based UIs can already be implemented using these widgets (e.g. all kinds of web forms). There are two tag types which are of particular importance: **group** tags and **action** tags. The **group** tags allow to group objects which have no meaning when they are separated. An example of this is a "date interactor": the interactors involved for filling in a date should not be separated (listing 1.2). Groups can be nested: they can be hierarchically structured. This enables us to reuse groups of interactors, and make new composed groups. The **action** tags allow a user to specify which action to fire if the interactor (which is the parent node) is manipulated. The action tag specifies the target (this can be a class name, a server,...) and the functionality that has to be invoked from this target. It is also possible to specify parameters and use the names of the interactors or groups for these parameters. Our system will automatically extract the current content out of the interactor or group (to which these parameter identifiers point) and pass it to the invoked functionality. There is no need to indicate the type for the UI designer, the type checking will be done at runtime. This is advantageous for the level of abstraction, but demands a detailed exception handling algorithm, and allows little or no compile-time or design-time checks. Further implementation may be required to reveal more opportunities to check the validity of the description at design- or compile-time.

Listing 1.2. A date group

```
<group name="date">
 <interactor>
  <range name="day">...</range>
 </interactor>
 <interactor>
  <range name="month">...</range>
 </interactor>
 <interactor>
  <range name="year">...</range>
 </interactor>
</group>
```

3 User Interface Descriptions and Components

3.1 The SEESCOA Component Framework

Within the SEESCOA[1] project a component framework for embedded systems is being developed. One of our involvements for this project is merging UI design and component-based development for embedded systems. The component system is asynchronous and uses the Java programming language as a common platform. Components communicate by sending asynchronous messages to each other, and not by using traditional synchronous message calls.

A traditional approach, making a static UI as a layer on a service or a data layer, has proven to lack flexibility. We consider components as units that contain logically grouped functionality and data, each living in their own memory space. They should offer an abstract description of how the service or data offered can be presented. Think about components as software units offering a particular service through their interface: their interface is actually a description of their functionality. It is a natural extension to also allow components to describe what they want to offer to a human user.

Each component can provide a description expressed in XML of the functionality it offers. Alternatively, they also could express in which way they could be interacted with. This is not true for all components of course (some just offer basic functionality on a lower level for other components), so only the components directly interested in human interaction should provide an abstract UI description. When building applications out of components a UI is automatically built: each component has its UI in the form of an XML description. These XML descriptions can all be seen as subtrees of the final, composed UI description. I.e. the UI will be automatically composed by connecting the UI descriptions of the components in to a bigger UI description. Figure 3 shows how this works using a small example: the Camera Mosaic component which is described in more detail the next section (section 3.3). Each component can contain a description of their UI: a description of a Camera can be found in 1.3 and of the Mosaic in 1.4. Figure 3 presents how the descriptions can be combined at runtime to create the UI out of the components.

Listing 1.3. UI description of a single camera component

```
<group name="camera2">
  <interactor>
    <videowidget name="video">...</videowidget>
  </interactor>
  <interactor>
    <range name="zoomrange"><action>
      <func service="Surveillance.Controls">setFocus</func>
               <param name="camera2"/>
      <param name="zoomrange"/>
    </action></range>
  </interactor>
  <interactor><range name="focusrange">...</range></interactor>
  <interactor>
    <button name="camera1_onoff"><action>
         <func service="Surveillance.Controls">switch</func>
               <param name="camera2"/>
      <param name="camera1_onoff"/>
    </action></button>
  </interactor>
</group>
```

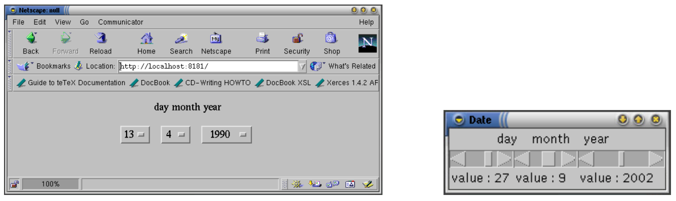
Listing 1.4. UI description of a Mosaic component

```
<ui>
  <title>Camera mosaic</title>
    <group name="mosaic">
      <group name="camera1">&CAMERA1</group>
      <group name="camera2">&CAMERA2</group>
      <group name="camera3">&CAMERA3</group>
      <group name="camera4">&CAMERA4</group>
    </group>
</ui>
```

Notice this approach allows components to migrate and offer their services in other places. The UI integrates smoothly in the new system the component is used on. The component-based approach supports a distributed view on assembling applications out of components and generating their UI: parts of the UI are allowed to migrate together with the functionality the components offer. Finally, the UI description can be submitted to a “renderer” component in the form of an XML document.

3.2 The Rendering Component

As we take a component-based approach for designing UIs for embedded systems, there is one “basic” component: the UI renderer component. This can be compared to a web-browser: a description for an interface can be submitted to the component and it will take care of rendering this description. Nevertheless, there are some differences: the component can receive a description of a UI and render it to different kinds of output devices and widget sets. The state of the UI can be “serialised” back into XML and relocated, which makes the component approach suitable for distributed systems or remote UIs. The SEESCOA



(a) In a browser (b) Java AWT

Fig. 1. Two different views on listing 1.2, both automatically generated

component system takes care of the communication and makes it network transparent. Notice the rendering engine is also embedded in a component, so this component can also have a UI description of its own functionality. To show its UI the rendering component can send its UI description to itself.

There are several possible output formats and for each kind of output a different rendering component can be supplied. For example: there could be rendering components for a PDA (e.g. Palm, see figure 5), for Java Swing (suitable for use on a desktop PC) and a rendering component for speech synthesis. The date group presented in listing 1.2 is rendered using two different rendering components in figure 1: a HTML rendering component and a Java AWT rendering component. The rendering components are “self-contained”: they do not rely on other components and are suitable to migrate individually to a particular system or software platform. Their internal working relies on the same code nevertheless.

3.3 A Case Study: a Camera Surveillance system

To illustrate how components can deliver their own UI description, we developed an example case study in the context of the SEESCOA project: a surveillance system. The example surveillance system consists of 4 cameras, each camera is represented by a component. The system also contains a Mosaic component, combining the controls for each camera in a combined control. The Mosaic component communicates with a rendering component which renders a UI to an output device. The setup is presented in figure 2. Notice each camera component has its own UI description (shown in listing 1.1) presented as an XML notation. This is shown by the trees attached to the camera components in figure 2. Each camera may offer different possibilities so they can all have different UI descriptions (The camera component is a component which abstracts the hardware and presents a real surveillance camera).

Because of the possibility to specify hierarchical groups, the Mosaic component can take the four individual controls and add them as subtrees in a new

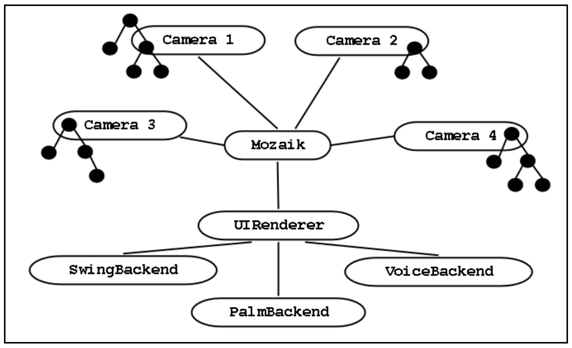


Fig. 2. Component composition example: a simple camera surveillance system

tree. The Mosaic component only needs to add a new root with 4 groups as the children of the root node. Each control can be attached to a group node (figure 3, the group nodes are coloured gray). The UI description tree produced by the Mosaic component is passed to the rendering component and rendered according to the chosen back-end. This illustrates how combining components to access their provided functionality in one application automatically results in a combined UI of these components. Notice several hierarchies can be mixed if desired: a subtree can be attached to an “open” node on another level in a new tree. This should be done with care: the chances of illogical and unusable generated UIs can increase by doing this. Our current system does not link the several subtrees across hierarchies, so no further support for mixing hierarchies is provided.

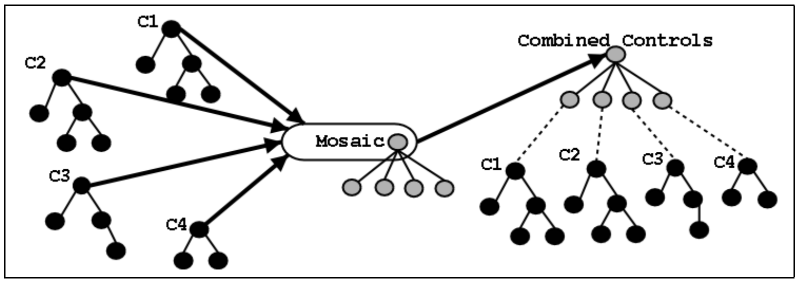


Fig. 3. The Mosaic component combining several UI descriptions

Depending on the target device the UI for the Mosaic component will be different. Suppose for example we want to access the Mosaic component using a traditional desktop computer: the rendering component for a desktop PC will load the available Concrete Interaction Objects (CIOs)[14] and try to map the

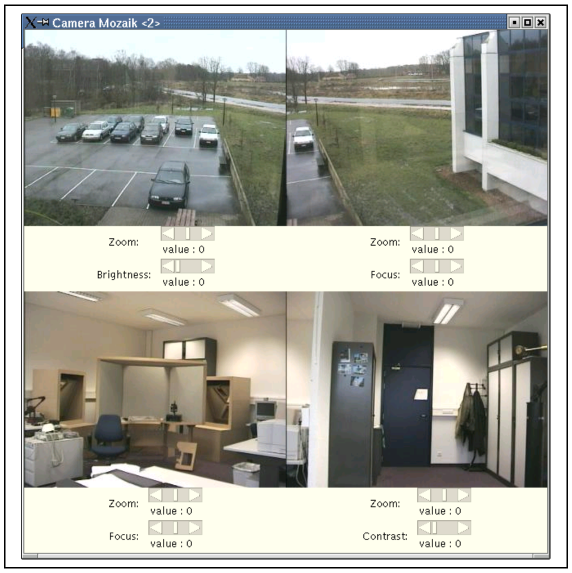
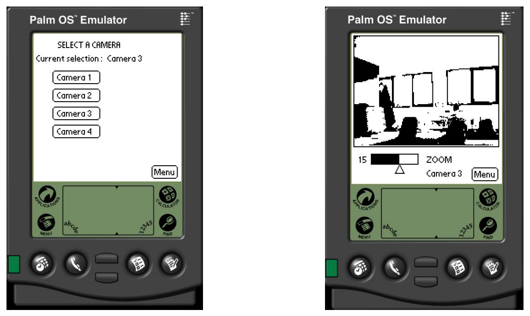


Fig. 4. The Mosaic component on a desktop

Abstract Interaction Objects (AIOs) described in the Mosaic UI description on a widget set suitable for a desktop machine: figure 4 shows this. If we want to access the functionality of the Mosaic component using our PDA, the rendering component for a PDA will do the same thing: load the available CIOs and trying to map the AIOs on this set of CIOs. This time the rendering component knows the PDA has limited possibilities, so it adapts the concrete UI to the screen space constraints. Figure 5 shows the results using a PDA (Palm IIIc). The focus of this work was not data communication but runtime UI migration, so we did not spend time investigating effective data communication between devices. The "videostream" for the PDA was actually implemented by sending separate down-scaled images to the device over its infrared connection. Of course, this can be done much more effective using other techniques or means of communication.

3.4 Extending the case study: decomposing tasks

The case study introduced in section 3.3 is a very simple "interaction session" with a single dialog. We consider an interaction session as the interaction which happens to complete a subtask, like "select camera" in figure 6 for example. Most UIs have more than one interaction session: in a dialog-based UI several dialogs are presented after each other. A design method to take this into account is required at this stage. The design method should enable the designer to decompose tasks hierarchically, and link several interaction sessions to each other



(a) Pick a camera (b) Observe it

Fig. 5. The Mosaic component on a PDA

in order to achieve the postulated goal. This method should support a device independent specification of the UI.

To solve this problem, we combine ConcurTaskTree (CTT) [11] with our component-based description method. One of the advantages of the CTT notation is that we can extend it to model context-sensitive user tasks as described in [12]. Characteristics that determine the context of use include the computing platform, the available interaction devices, available screen space,... When one or more of these characteristics change, a reconfiguration of the UI may be required to adapt to the new context of use. [12] proposes a notation to model context-sensitive user tasks. Their solution consists of a CTT task model with roughly the following parts: a non-context-sensitive CTT part and context-sensitive parts depending on some conditions.

The second advantage is the asynchronous nature of the SEESCOA component system: CTT allows to describe temporal relations, and includes concurrent tasks in its notations. A third advantage is the hierarchical structure it offers: our approach also uses an hierarchical notation to describe the UI in a device independent manner.

Now suppose a human guard has access to a security system using a regular workstation or a PDA. Some tasks he can perform on the workstation are not possible on the PDA. Suppose for example that it's not possible to observe more than 1 camera at the same time on the PDA due to the minor screenspace provided by it. So it depends on the context of use (the device that's being used in this case) whether the operator can pick just one or multiple cameras to observe at a time. Obviously we can say that this is a context-sensitive task. There are also a couple of non-context-sensitive tasks in this case. The operator must logon to the system before he can pick cameras. Also he can choose to stop observing or pick other cameras to observe. While the guard is observing a camera (or cameras depending on the context) the other cameras will continue to record their video streams until the guard logs out again. The enhanced CTT tree is shown in figure 6.

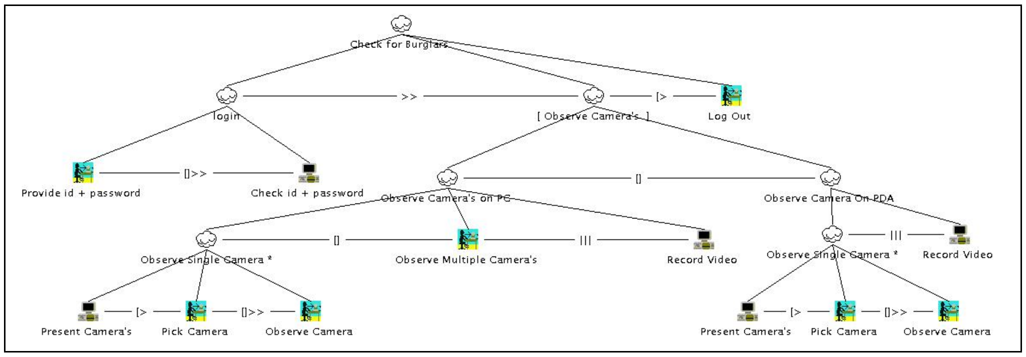


Fig. 6. ConcurTaskTree diagram: checking for burglars with the camera surveillance system in a context-sensitive way

While being a good solution for modeling context-sensitive tasks there are two minor drawbacks to it. The first one is that some subtrees may appear more than once in the model. For example in figure 6 the subtree *Observe Single Camera* appears in the two different contexts of use. [12] solves this by factoring out these subtrees by placing them in the context-insensitive part of the model. The second drawback is that we still have to model every possible context of use: for each device different properties have to be taken into account. In our approach we try to avoid this by using abstract UI descriptions for an interaction task. A CTT description can be saved as an XML document, which allows us to attach our own XML description at the leafs representing an interaction task. These XML descriptions are actually the composed descriptions of the components which are used at that moment. A CTT description becomes a way to describe how we want to interact with a set of given components in a particular stage of the usage of an application. We gain a model-based approach for designing the UI, extending the component-based approach for modelling the software itself. So, instead of using a context-sensitive description as shown in figure 6 we can accomplish the same thing with a non context-sensitive description as shown in figure 7. We recognise that these are just the first steps, and the method has not been tested for a wider range of devices yet. When using totally different ways of interaction (e.g. not dialog-based), we expect we need context-sensitive parts as a consequence of particular other ways to complete the subtasks.

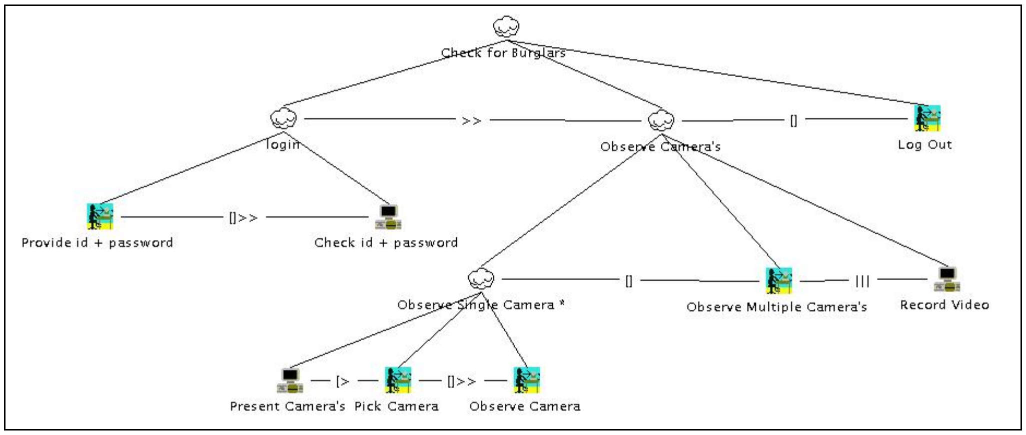


Fig. 7. ConcurTaskTree diagram: checking for burglars with the camera surveillance system

4 Flexible and Adaptable User Interfaces for Embedded Systems

4.1 Realising a concrete UI

To transform the abstract UI description into a concrete one it has to pass several stages of processing in our approach:

1. a mapping stage
2. a specialised layout management stage
3. the rendering stage

The UI description, presented as a tree in memory, is passed to a rendering component, which will initiate the mapping stage: it tries to convert the AIOs into CIOs [3]. For each available widget set, the mapping choices are implemented ad hoc: the current implementation does not support user guidance. This is one of the shortcomings in our approach: we tend to solve this problem in a following iteration. The mapping stage will convert the abstract UI description to a "platform specific" description for one or several specific modalities (using XSLT[2] or an agent component).

Once the system has built a concrete representation structure, the actual screenspace needed for this presentation can be calculated. The mapping stage already involves some calculations of the weighted values of the AIOs, and the corresponding space they may require.

The final step is to show the actual UI: this is done by rendering the CIOs on screen. The widget set used to do the mapping is provided by the target platform and therefore it defines how to represent the CIOs visually.

[2] e**X**tensible **S**tylesheet **L**anguage **T**ransformations

4.2 System independent User Interfaces

Every time a new device is used as output device, the specific UI renderer component will use the device profile, containing the device constraints and its ad hoc knowledge of the target system. The renderer changes the UI presentation according to the defined limitations.

One of the consequences of adapting the UI to new device constraints is the need for an automatic layout algorithm when GUI rendering is used. When the UI moves to a new output device, the UI should be laid out in a logical way. One approach achieving this is by using layout algorithms found in diagram rendering (like graphs and state-charts). Due to the hierarchical view on the UI, we try to adopt weighting algorithms especially designed for presenting as hierarchical data like presented in [4]. Every leaf is given a *weight* indicating its complexity (primarily space needs). Recursively every group (i.e. every node that is no leaf) will get the complexity of its children and is added up with a certain constant value in complexity weight. This is a simple attempt to automate the layout algorithm, without taking into account real usability issues which arise when automating this process.

Our architecture allows each subtree of a UI description tree to use a different layout algorithm. For example; we use a layout algorithm that allocates space from left to right in a rectangular space for the first level of subtrees under the root node. The space is allocated according to the weighted complexity of each subtree. On the next level, a more complex layout algorithm is used (like a GridBagLayout in the Java programming language) for each subtree. One of the advantages of this approach is a better support for fragmented UI (several parts of the UI are accessible from several devices), multi-modal UIs and dynamically changing UIs. Currently we are integrating *spatial constraints* for 2D UI in our system, so the UI designer can indicate how AIOs should be placed in relation to each other [7].

5 Summary of our Current Results

Current results include a rendering component, to which an XML document describing an abstracted UI can be submitted. The renderer maps this description to an actual widget set and tries to adapt the layout so the UI fits on screen. When the screen size becomes too small, the renderer will try to split different parts of the UI and put them behind each other. While doing this, logically grouped elements will not be split up. These grouping operators are specified in the abstract UI description: they group user interactions which logically depend on each other. Examples of tested target widget sets are Java *AWT*, *Swing*, *kAWT* and HTML (web pages).

The SEESCOA Components can be combined in order to make a fully functional application and their UI description can be combined automatically. An example of this was described using the Camera Mosaic application. This enables User Interfaces to become migratable: first of all their description can be rendered to other output devices and second the UI can accompany the component it represents when it is sent to another system. We have only tested the system with simple UIs, so no conclusions can be made concerning scalability.

6 Conclusions and Future Work

The new component-oriented approach suggested in this paper has several advantages for developers of embedded systems and mobile computing devices in particular. It is

Flexible : changing the UI can be done by another renderer component or letting components provide another UI description;
Reusable : providing a high level *description* of the UI related to the functionality a component offers, allows easier reusability of previously designed UIs in contrast with hard-coded UIs;
Adaptable : by abstracting the UI, device constraints can be taken into account when rendering the concrete UI.

Besides these advantages we showed how attaching abstract UI descriptions to components helps to compose User Interfaces at runtime without intervention from a programmer. This is especially important when a mobile computing device has to present a new service that it was not aware of. E.g. a PDA comes near to a printer and should be able to present the accessible functionality of this printer. All these advantages make the UIs migratable: they can be easily transported from one device to another, adapting to new environments.

Future work includes adding alternative output rendering components other then a 2D screen renderer like speech output and the implementation of context-sensitive layout algorithms. We acknowledge there is a lack of support for artistic and aesthetic influences in the creation of the UIs employing the approach we presented in this paper. It is our intention to look at alternative interaction methods besides traditional interaction methods. Due to the asynchronous nature of the SEESCOA component system, it is interesting to take time-related HCI patterns into account.

Although the focus is not on the usability of the UIs, introducing these patterns can help us to ensure a minimal usability. For introducing design-time type checks an appropriate editor for this is required. Some checks can be done if a certain amount of information of the application logic is available (the editor should know which arguments can be handled by what kind of functionality). An editor for designing the UI descriptions is not available at this moment.

7 Acknowledgements

Our research is partly funded by the Flemish government and EFRO[3]. The SEESCOA project IWT 980374 is directly funded by the IWT[4]. The *Vrije Uni-*

[3] European Fund for Regional Development
[4] Flemish subsidy organization

versiteit Brussel (Programming Lab) and *Katholieke Universiteit Leuven (Distrinet)* have created the SEESCOA component system.

References

1. Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. *UIML: An Appliance-Independent XML User Interface Language.* World Wide Web, `http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html`, 1998.
2. Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An appliance-independent XML user interface language. *WWW8 / Computer Networks*, 31(11-16):1695–1708, 1999.
3. Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta. Applying Model-Based Techniques to the Development of UIs for Mobile Computers. In *IUI 2001 International Conference on Intelligent User Interfaces*, pages 69–76, 2001.
4. David Harel and Gregory Yashchin. An Algorithm for Blob Hierarchy Layout. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 29–40, May 2000.
5. Elliotte Rusty Harold. *XML; Extensible Markup Language, Structuring Complex Content for the Web.* IDG Books Worldwide, 1998.
6. IBM Corporation. *MoDAL (Mobile Document Application Language).* World Wide Web, http://www.almaden.ibm.com/cs/TSpaces/MoDAL/.
7. Simon Lok and Steven Feiner. A Survey of Automated Layout Techniques for Information Presentations. In *Proceedings of SmartGraphics 2001*, March 2001.
8. Kris Luyten and Karin Coninx. An XML-based runtime user interface description language for mobile computing devices. In *Proceedings of the Eight Workshop of Design, Specification and Verification of Interactive Systems*, pages 17–29, June 2001.
9. Andreas Müll ler, Peter Forbrig, and Clemens Cap. Model-Based User Interface Design Using Markup Concepts. In *Proceedings of the Eight Workshop of Design, Specification and Verification of Interactive Systems*, pages 30–39, June 2001.
10. Dan R. Olsen, Sean Jefferies, Travis Nielsen, William Moyes, and Paul Fredrickson. Cross-modal interaction using XWeb. In *Proceedings of the 13th Annual Symposium on User Interface Software and Technology (UIST-00)*, pages 191–200, N.Y., November 5–8 2000. ACM Press.
11. Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications.* Springer, 2000.
12. Costin Pribeanu, Quentin Limbourg, and Jean Vanderdonckt. Task Modelling for Context-Sensitive User Interfaces. In *Proceedings of the Eight Workshop of Design, Specification and Verification of Interactive Systems*, pages 60–76, June 2001.
13. David Urting, Stefan Van Baelen, Tom Holvoet, and Yolande Berbers. Embedded Software Development: Components and Contracts. In *Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems*, pages 685–690, 2001.
14. J. Vanderdonckt and F. Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *ACM Conference on Human Aspects in Computing Systems InterCHI'93*, pages 424–429. Addison Wesley, 1993.