Task-Based Design and Runtime Support for Multimodal User Interface Distribution

Tim Clerckx, Chris Vandervelpen, and Karin Coninx

Hasselt University, Expertise Centre for Digital Media,
and transnationale Universiteit Limburg
Wetenschapspark 2, BE-3590 Diepenbeek, Belgium
{tim.clerckx, chris.vandervelpen, karin.coninx}@uhasselt.be

Abstract. This paper describes an approach that uses task modelling for the development of distributed and multimodal user interfaces. We propose to enrich tasks with possible interaction modalities in order to allow the user to perform these tasks using an appropriate modality. The information of the augmented task model can then be used in a generic runtime architecture we have extended to support runtime decisions for distributing the user interface among several devices based on the specified interaction modalities. The approach was tested in the implementation of several case studies. One of these will be presented in this paper to clarify the approach

Keywords: Task-based development, model-based user interface development, distributed user interfaces, multimodal user interfaces.

1 Introduction

In the last decade users are increasingly eager to use mobile devices as an appliance to perform tasks on the road. Together with the increase of wireless network capabilities, connecting these mobile *assistants* to other computing devices becomes easier. As a result we are at the dawn of the era of context aware computing. Context is a fuzzy term without a consent definition. In this work we define context as the collection of factors influencing the user's task in any way, as described by Dey [9]. Factors such as available platforms, sensor-based environmental context, the user's personal preferences, and setup of interaction devices appertain to this set. When we pick out context factors such as available platforms and interaction devices, we are discussing the area of Ubiquitous Computing [19] where users are in contact with several devices in their vicinity.

In previous work we have been concentrating on model-based development of context-aware interactive systems on mobile devices. We created a task-based design process [5] and a runtime architecture [6] enabling the design, prototyping, testing, and deployment of context-aware user interfaces. The focus in our previous work was to create context-aware applications where context factors such as sensor-based context information or information from a user model can be associated with a task model in order to enable the generation of prototypes and to use a generic runtime

architecture. However, in our approach the user interface was always centralized on a mobile device. In this work we describe how we have extended our framework, DynaMo-AID, in order to support the shift towards Ubiquitous Computing. We will discuss how a task model can be enriched with properties that are used (1) at design time to specify how tasks should be presented to the user according to the platform and (2) at runtime to distribute the tasks among the available interaction resources (definition 1). Devices may support several distinct interaction techniques. E.g. on the one hand editing text on a PDA can be accomplished by using a stylus to manipulate a software keyboard. On the other hand speech interaction can be used provided that the PDA is equipped with a microphone. As a result, at runtime has to be decided which interaction resources are at the user's disposal and a usable distribution among interaction resources has to be chosen.

Runtime distribution requires meta data about the tasks in order to realize a usable distributed user interface. This is in particular the case when we are considering ubiquitous environments because at design time it is impossible to know what the environment will look like regarding available interaction resources. E.g. the user walks around with his/her mobile device and comes across a public display that can be communicated with through a wireless connection. When this is the case decisions regarding user interface distribution have to be taken at runtime to anticipate on the current environment. Furthermore, a mapping of abstract information about the user interface to more concrete information is required to construct the final user interface due to the unknown nature of the target device(s).

The remainder of this paper is structured as follows. First we give a brief overview of the DynaMo-AID development process (section 2.1). We focus on the parts relevant for this paper. Next we elaborate on the changes we have applied to the process to enable the support for modelling multimodal and distributed user interfaces (section 2.2). Afterwards the ontology constructed to support the modelling of the modalities and devices is discussed (section 2.3). Section 3 discusses the runtime architecture: first an overview is presented (section 3.1), then we focus on the rendering engine (section 3.2), finally we discuss the approach used to decide how to distribute the tasks among the available devices. In the following section we will discuss related work and compare it to our approach. Finally conclusions are drawn and future work is discussed.

2 Overview of the extended DynaMo-AID development process

In this section we first introduce the DynaMo-AID development process for context-aware user interfaces. We emphasize the changes we have made to support the development of multimodal and distributed user interfaces. We focus on the part of the design process where a task specification is enriched with interaction constraints. Finally we elaborate on the environment ontology used for defining the interaction constraints.

2.1 Developing context-aware user interfaces

The DynaMo-AID development process (Fig. 1) is prototype-driven with the aim to obtain a context-aware user interface. The process consists of the design of several abstract and concrete models. After the specification of these models, the supporting tool generates a prototype taking into account the models. The prototype can then be evaluated to seek for flaws in the models. Afterwards the models can be updated accordingly and a new prototype is generated. These steps in the process can be performed iteratively until the designer is satisfied with the resulting user interface. Next the user interface can be deployed on the target platform.

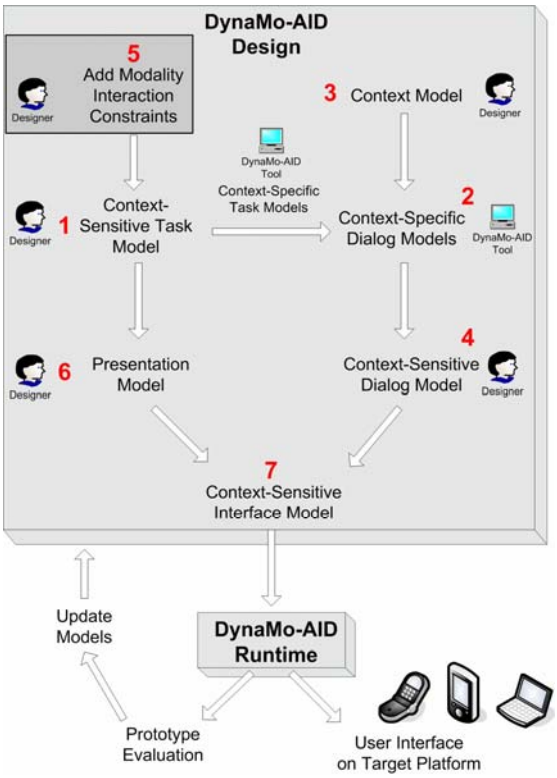

Fig. 1. Overview of the DynaMo-AID development process

The upper part of Fig.1 reveals an overview of the design process. First the designer has to construct a context-sensitive task model (1). To accomplish this, the designer makes use of the ConcurTaskTree notation [12] augmented with extra tasks to introduce context-awareness at the task level [4]. Taking into account this information, the tool extracts a set of dialog models (2) where each dialog model is relevant for a particular context of use. Afterwards these dialog models are connected at those points relevant to apply a context change (4), i.e. a switch from a dialog model relevant in a certain context of use to another dialog model relevant in another

context of use. Furthermore the designer specifies the kind of context information implying the context change (3). The fifth step (5) is an extension and will be discussed in section 2.2. Next the concrete tasks are annotated with Abstract Interaction Objects (AIOs) [17] providing an abstract description about the way the task will have to be presented to the user (6). The aggregate of the models are collected in the interface model (7) which is the input for the runtime architecture in order to either generate a prototype or deploy the user interface on the target platform.

Important for the remainder of the paper is the fact that the dialog model is a State Transition Network (STN). Each state in the STN is an enabled task set, a collection of tasks enabled during the same period of time [12]. This means the tasks should be presented to the user simultaneously, i.e. in the same dialog. The transitions of the STN are labelled with the task(s) initiating the transition to another state. Using this information, a dialog controller can keep track of the current state of the user interface and invoke a switch to another dialog if appropriate (section 3).

Accordingly the dialog model provides the information necessary to decide *which* tasks have to be deployed at a certain moment in time. When several devices and/or interaction modalities are available to the user, the question arises *where* these tasks have to be deployed.

Previous research already tackled the problem of deploying task sets on different devices. Paternò and Santoro [13] for instance described that tasks or domain objects related to a task can be assigned to a selection of platforms in order to decide at runtime whether or not to deploy a task according to the current platform. Our approach also supports this possibility at the task level where it is possible to assign different tasks to different contexts of use (platform is one kind of *context of use*).

However, we argue the approach of enabling tasks for a certain platform and disabling these same tasks for another platform might constrain the user in accomplishing his/her goals. On the one side this can be desirable when the domain objects supporting the performance of this task are constrained by the platform but on the other side the user will not be able to perform all the tasks in the path to accomplish his/her goals. This problem can be tackled by distributing the tasks among different devices in the user's vicinity in a way that all the necessary tasks can be presented to the user. In the next section we propose a method to assign interaction constraints to the tasks in order to make the distribution of tasks among distinct devices and/or interaction modalities possible at runtime.

2.2 Supporting the design of distributed and multimodal user interfaces

As we have explained in the previous section each state in the dialog model consists of a set of tasks. When the user interface is deployed in a highly dynamic environment with different interaction devices and/or modalities the system has to decide which tasks are deployed on which interaction device supporting the appropriate modalities. Some additional abstract information regarding task deployment is necessary to make these decisions. Therefore we define the following terms based on the definitions in [18]:

Definition 1 An *Interaction Resource (IR)* is an atomic input or output channel available to a user to interact with the system.

Definition 2 An *Interaction Resource Class (IRC)* is a class of Interaction Resources sharing the same interaction modalities.

Definition 3 *An Interaction Device (ID)* is a computing device that aggregates Interaction Resources associated with that particular computing device.

Definition 4 An *Interaction Group (IG)* is a set of joined Interaction Devices necessary for a user to perform his/her tasks.

An example of an ID is a traditional desktop computer that aggregates the IRs keyboard, mouse and display screen. The QWERTY-keyboard attached to the desktop computer is an Interaction Resource belonging to the Interaction Resource Class of keyboards. An example of an IG is the collection of TV ID with a set-top box and a PDA ID, where the PDA is used as a remote control to interact with the TV system.

The goal of the research described in this paper is to find a way to distributed tasks among the available Interaction Resources, given the setup of an Interaction Group. To accomplish this the designer will have to provide additional information for each task about the types of Interaction Resources that can be used to perform the task. Because a task might be performed by several distinct Interaction Resource Classes (e.g. editing text can be done with a keyboard and/or speech) the designer will have to specify how these IRCs relate to each other. This can be expressed using the *CARE* properties introduced by Coutaz et al. [8]. The CARE properties express how a set of modalities relate to each other:

- **Complementarity**: all the modalities have to be used to perform the task;
- **Assignment**: a single modality is assigned to the task in order to perform the task;
- **Redundancy**: all the modalities have the same expressive power meaning the use of a second modality to perform the task will not contribute anything to the interaction;
- **Equivalence**: the task can be performed by using any one of the modalities.

The CARE properties are an instrument to reason about multimodal interactive systems. We use the CARE properties in our approach to indicate how the different modalities assigned to the same task relate to each other. Therefore we define:

Definition 5 A Modality Interaction Constraint (MIC) is a collection of modalities related to each other through a CARE property.

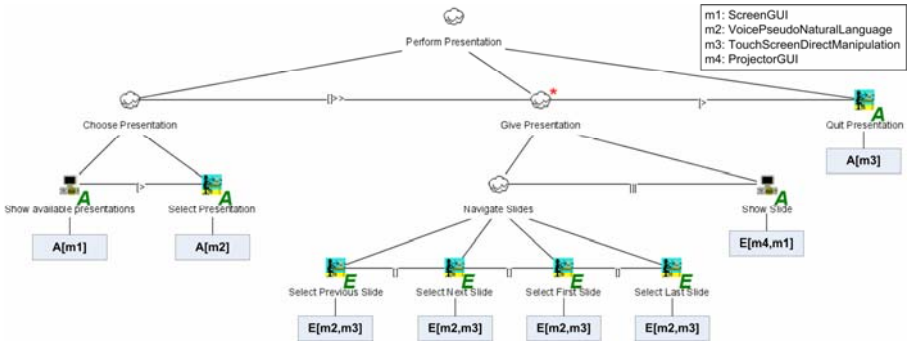

Fig. 2. Example task model with interaction constraints appended to tasks

The information provided by the Modality Interaction Constraint associated with a task can then be used at runtime to find an Interaction Resource belonging to an Interaction Resource Class supporting the appropriate modalities. The relation between modalities and IRCs will be explained in section 2.3.

Fig. 2 shows an example of a task model annotated with Modality Interaction Constraints. The task model describes the task of performing a presentation. First the presentation has to be selected. To accomplish this the available presentations are shown to the user on a device supporting the *ScreenGUI* output modality. The task to select the desired presentation is assigned to the *VoicePseudoNaturalLanguage* modality. This implies the task can only be performed using speech input. Afterwards the presentation can commence. The presenter can navigate through the slides by using a device supporting *VoicePseudoNaturalLanguage*, *TouchScreenDirectManipulation* or both in which case the user chooses the modality.Meanwhile the slide is shown on a device using either a *ProjectorGUI* or a *ScreenGUI.*

The presentation can only be switched off using *TouchScreenDirectManipulation* to prevent someone in the audience to end the presentation prematurely.

2.3 Interaction Environment Ontology

In order to make it easy for a designer to link modalities to tasks, we have constructed an extensible interaction environment ontology describing different modalities, Interaction Resource, and the way these two concepts are related to each other. The ontology we have constructed is an extension of a general context ontology used in the DynaMo-AID development process [14].

Fig. 3. Structure of the Interaction Environment Ontology

Fig. 3 shows the ontology consisting of two parts. The first part describes the interaction environment using the classes *InteractionDevice* and *InteractionResource*. An InteractionDevice aggregates one or more interaction resources using the *hasInteractionResource* property. An interaction resource can either be an *OutputInteractionResource* or an *InputInteractionResource*. Every class in the ontology has particular properties describing the characteristics of individuals of that class. For example, a Desktop individual contains *hasInteractionResource* properties pointing to individuals of the class *CrtScreen, Keyboard and Mouse*. The Mouse individual on its turn has properties for describing the number of buttons, the used technology…

The second part of the ontology describes the possible modalities based on concepts described in [8]. In this work a modality is defined as the conjunction of an interaction language (direct manipulation, pseudo natural language, gui…) and an interaction device/resource (mouse, keyboard, speech synthesizer…). To model this, we added the classes *InteractionLanguage* and *Modality* to our ontology. A *Modality* individual can be an *InputModality* or an *OutputModality*. A concrete *Modality* individual is defined by two properties. The *usesInteractionLanguage* property points to an *InteractionLanguage* individual. At this time these are *DirectManipulationLanguage, GuiLanguage or PseudoNaturalLanguage.* It is possible for the designer to add new *InteractionLanguage* individuals to the ontology. The second property of *Modality* individuals is the *usesDevice* property. This property points to an *InteracionResource* individual. In this way we created six predefined modalities: MouseDirectManipulation, KeyboardDirectManipulation,

VoicePseudoNaturalLanguage, SpeechOutputPseudoNaturalLanguage, ScreenGui and ProjectorGui. A designer can add new modalities to the ontology as she/he likes. To link a particular *Modality* individual to an *InteractionDevice* individual the property *supportsModality* is used. As shown in fig. 3 using the thick rectangles, an individual desktopPC1 of the *Desktop* class could be linked to a *MouseDirectManipulation* modality using the *supportsModality* property. The modality on its turn is related to a *Mouse* individual using the *usesDevice* property and to the *DirectManipulation* interaction language using the *usesInteractionLanguage* property. Notice that for this to work, the *Mouse* individual has to be linked to desktopPC1 using the *hasInteractionResource* property.

To enable designers to annotate Modality Interaction Constraints to the tasks, we have extended the DynaMo-AID design tool [7]. Fig. 4 shows the dialog box in the tool which inspects the type of task and queries the ontology in order to present the available modalities to the designer. If the task is an interaction task, input modalities will be shown to the designer, if the task is an application task, output modalities will appear in the *Available Modalilties* part of the dialog box.

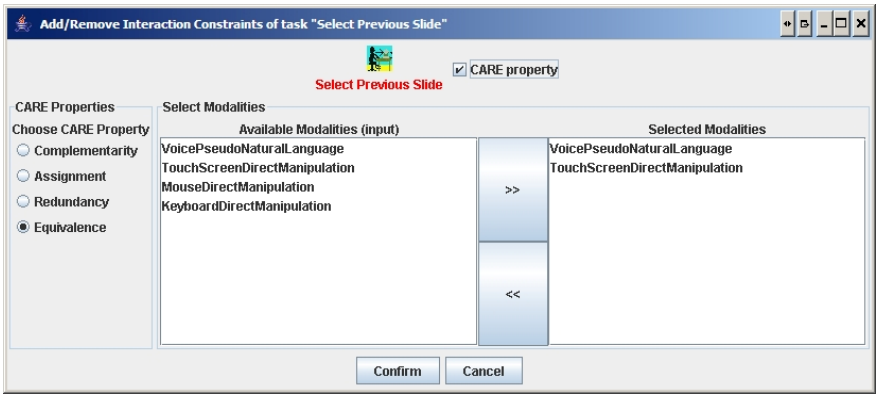

Fig. 4. Screenshot of the dialog box used to annotate a task with an interaction constraint

3 Runtime support: prototyping and deployment of the user interface

In the previous section we have described how designers can add information to a task model to describe which interaction modalities are appropriate to perform the tasks. In this section we discuss how this information can be used at runtime in order to enable runtime distribution of tasks among Interaction Devices.

3.1 Overview of the runtime architecture

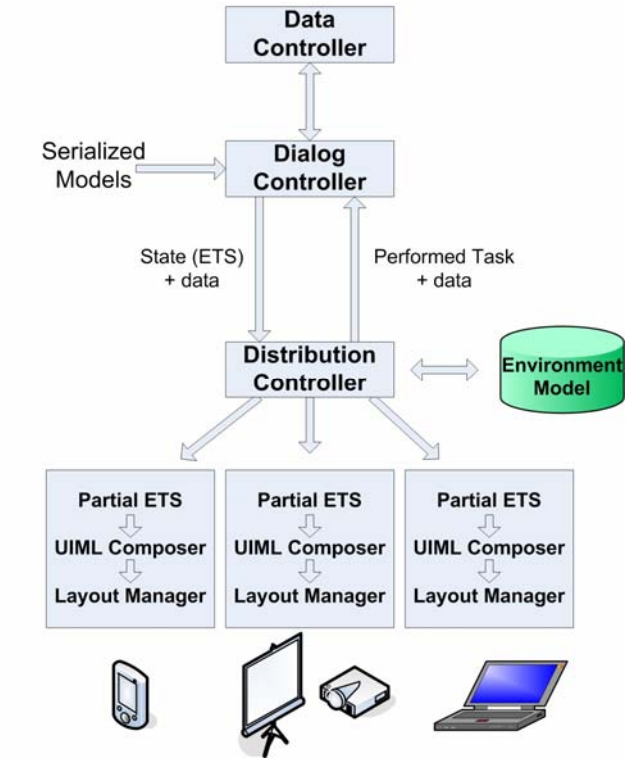

Fig. 5. Overview of the extensions of the runtime architecture

To support distribution we have extended our existing generic runtime architecture supporting model-based designed user interfaces influenced by context changes. Fig. 5 shows an overview of the extensions we have applied. The models described in the previous section can be constructed in the design tool and can be serialized to an XML-based format. These models are the input of the runtime architecture. The *Dialog Controller* takes into account the dialog model to keep track of the current state of the user interface. The current state implies which tasks are enabled at the current time (section 2.1). The *Data Controller* keeps track of the data presented in the user interface and takes care of the communication with the functional core of the system.

When the user interface is started the environment is scanned for devices. The Universal Plug and Play[1] standard is used to discover devices in the vicinity. Each device broadcasts a device profile mentioning the available Interaction Resources supported by the device. Taking this information into account an *Environment Model*

[1] http://www.upnp.org

can be constructed. This environment model contains the whereabouts of the Interaction Devices and the available Interaction Resources for each device. When the environment model is constructed, the dialog controller will load the first state in accordance with the starting state of the State Transition Network. The active state thus corresponds to the tasks that are enabled when the system is started. This information is passed on to the *Distribution Controller* along with the data related to these tasks as provided by the data controller. The distribution controller will then seek for each task an appropriate Interaction Device containing an Interaction Resource that supports the interaction modalities related to the tasks. The distribution controller will then group the tasks by Interaction Device, resulting in Partial Enabled Task Sets (groups of tasks enabled during the same period of time and deployed on the same Interaction Device). Afterwards the Abstract Interaction Objects related to the tasks of the Partial Enabled Task Set are grouped and are transformed to a UIML[2] document. Behaviour information is added to the UIML document to be able to communicate with the renderer and an automatic layout manager will add layout constraints that can be interpreted by the rendering engine.

3.2 Rendering Engine

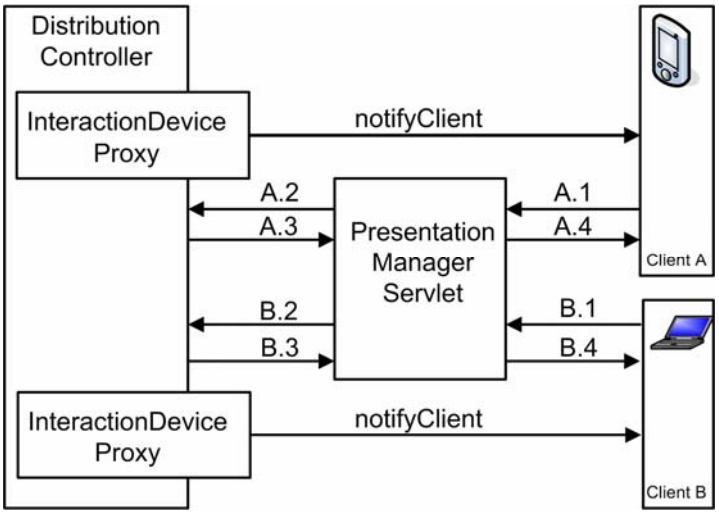

Fig. 6. Overview of the rendering architecture

Fig. 6 shows an overview of the rendering architecture consisting of three layers: the Distribution Controller, the Presentation Manager (a servlet) and the clients. Whenever the presentation of the user interface needs an update, e.g. when a new state has be deployed or when a user interface update occurs, the Distribution Controller sends a *notifyClient* message to one or multiple clients (depending on the

[2] http://www.uiml.org

distribution plan, section 3.3) using the InteractionDevice Proxy that is connected to the Client. As a response to this message, the client browsers are redirected to the URL where the Presentation Manager servlet awaits client requests (A.1 and B.1, HTTP). These requests can be of different types (according to the information in the notifyClient message):

- **requestUI**: requests a newly available presentation for the interaction device. After receiving the message, the Presentation Manager forwards the message to the Distribution Controller (A.2 and B.2) which responds by sending the UIML representation of the user interface and the data for this client to the Presentation Manager (A3 and B3). The Presentation Manager servlet now builds an internal PresentationStructure object for the user interface and stores the object in the current session. Depending on the modalities that should be supported, the presentation manager chooses the appropriate generator servlet, XplusVGeneratorServlet or XHTMLGeneratorServlet, that generates the concrete presentation and sends it as an HTTP response back to the client (A.4 and B.4). The task of the XplusVGenerator is to transform the PresentationStrucure object to XHTML + VoiceXml (X+V[3]). X+V supports multimodal (Speech + GUI) interfaces and can be interpreted by multimodal browsers such as the ACCESS Systems' NetFront Multimodal Browser[4] The XHTMLGeneratorServlet transforms the PresentationStructure object to XHTML for interpretation by normal client browsers;
- **requestDataUpdate**: requests a data update for the current presentation. When the Presentation Manager servlet receives this message from a client it is forwarded to the Distribution Controller (A.2 and B.2) which sends the user interface data as a response (A.3 and B.3). Now the Presentation Manager updates the data in the PresentationStructure object available in the current session and chooses the appropriate generator servlet to generate the concrete user interface and to send it to the client browser (A.4 and B.4);
- **taskPerformed**: when a user interacts with the user interface, e.g. by clicking a button, a taskPerformed message together with the request parameters are sent to the Presentation Manager which forwards the message to the Distribution Controller.

Notice that the system follows a Model-View-Controller architecture. The Presentation Manager Servlet is the controller, the generator servlets are the views and the PresentationStructure is the model.

Fig. 7 shows what happens when generating the user interface for the task model in fig. 2. In (a), the user interface was deployed in an environment without an interaction device that supports the modality *ProjectorGUI*. This implies, the *Navigate Slides* and the *Show Slide* task are all deployed on a PDA using the X+V generator and a multimodal browser that supports X+V. This means we can navigate slides using voice by saying for example `Next Slide` or `First Slide`, or we can use the stylus to interact with the buttons. In (b) the user interface is distributed because we added a laptop attached to a projector to the environment. In this case the *Navigate Slides* tasks are still deployed on the PDA using the X+V generator. The *Show Slide* task

[3] http://www.voicexml.org/specs/multimodal/x+v/12/

[4] http://www-306.ibm.com/software/pervasive/multimodal/

however is deployed on the laptop screen using the XHTML generator and an XHTML browser.

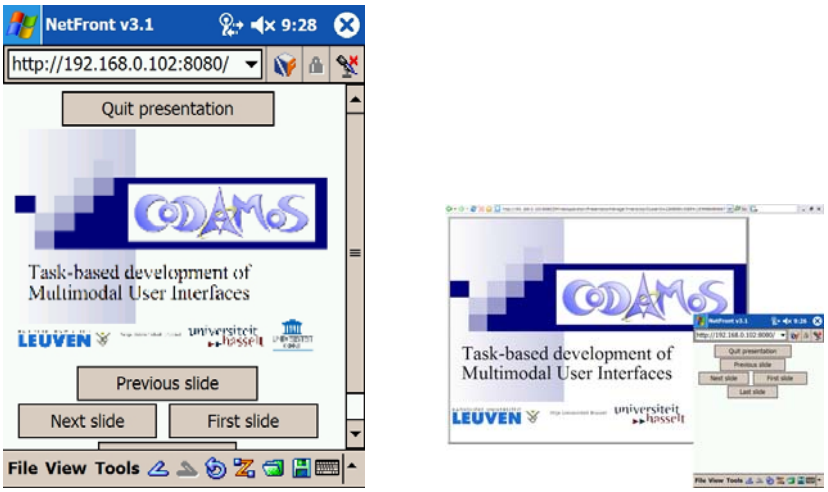

Fig. 7. Example of Fig.2 rendered on a single PDA (a) and in an environment with a PDA and a desktop computer (b)

3.3 Constructing a distribution plan

In the two previous sections we talked about the structure of the runtime architecture and the rendering engine. However the question how to divide an enabled task set into a usable federation of partial enabled task sets has not yet been discussed. In this section we discuss the first approach we have implemented and some observed problems with this approach. Afterwards we propose a solution asking some extra modelling from the designer.

Task-Device mappings using Query Transformations

In our first approach, we use a query language, SparQL[5], to query the information in the environment model which is a runtime instantiation of the Interaction Environment Ontology (section 2.3). SparQL is a query language for RDF[6] and can be used to pose queries at ontologies modelled using the OWL[7] language.

[5] http://www.w3.org/TR/rdf-sparql-query/
[6] http://www.w3.org/RDF/
[7] http://www.w3.org/TR/owl-features/

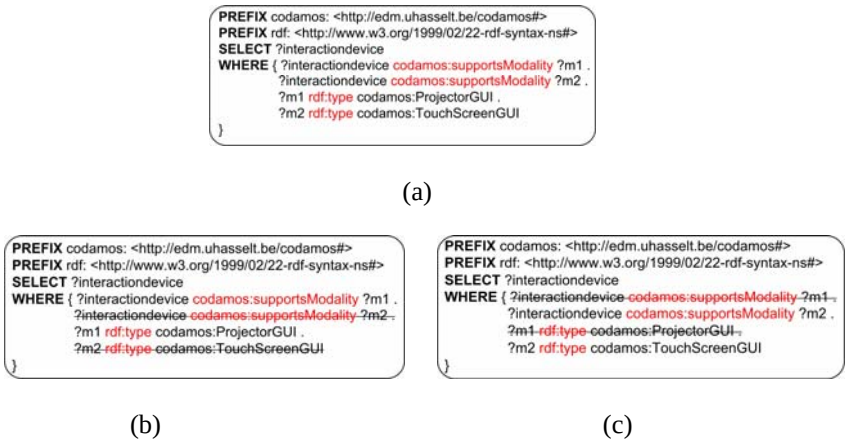

Fig. 8. Queries deducted from the Modality Interaction Constraint related to the *Show Slide* task of the example in Fig. 2. Query (a) searches for a device supporting all the modalities in the equivalence relation. Queries (b) and (c) are reduced queries that are constructed if query (a) did not return a result.

To map each task of the enabled task set to the appropriate Interaction Device, the Modality Interaction Constraint related to task task will be transformed to a SparQL query. Fig. 8 shows an example of the mapping of the Modality Interaction Constraints attached to the *Show Slide* task of our previous example. This constraint says that modality m_4 (ProjectorGUI) and modality m_1 (ScreenGUI) are equivalent for this task. The more modalities in the equivalence relation are supported by the interaction device, the better suited it will be for executing the task. This is what the query in Fig. 8(a) tries to achieve. In this query, an interaction device which supports both modalities is m_4 and m_1 searched for and when it is found, the task is deployed on the device. Now suppose we have a Desktop device in the environment attached to a projector but not to a screen. This means the Desktop supports the ProjectorGUI modality only. The query in Fig. 8(a) will return no interaction device. As a result the system will reduce the query to find a device that supports only one specified modality. In this case this is feasible because the constraint defines an equivalence relation so the devices supporting only one (or more) of the required modalities will also be able to handle the task. The first query that will be tried is the query in Fig. 8(b) because the ProjectorGUI modality is defined first in the modality constraint. Because we have a Desktop individual in the environment which supports this modality, it will be returned and the task is deployed on the device. If such a device is still not found, the system will try the query in Fig.8 (c) after which the task is deployed on a device with a Screen attached.

Notice that the queries in Fig. 8 easily extend to support the other three CARE properties by adding/removing rules such as the one presented in Fig. 9. The ModalitySubClass in the query can be one of the leaf Modality subclasses. In case of the Assignment relation this is easy because we want to select a device supporting only one modality. Complementarity is analogue to the Equivalence relation.

However, here all the modalities in the relation should be supported by the interaction device. In case of the Redundancy relation rules are added for each redundant modality.

```
?interactioncluster codamos:supportsModality ?m .
?m rdf:type ModalitySubclass .
```

Fig. 9. Extension rule for generating SparQL queries from Modality Interaction Constraints

We can summarise our approach as the execution of queries searching for an appropriate device supporting the modalities according to the CARE property relating the modalities. Priority for the execution of the queries is given to the modality specified first in the CARE relation (e.g. *ProjectorGUI* in the example of Fig. 8).

Partial Enabled Task Set refinements

We now have presented a way to use de Modality Interaction Constraints to divide an enable task sets into partial enabled task sets for a feasible distribution. However this distribution is not always the best case scenario.

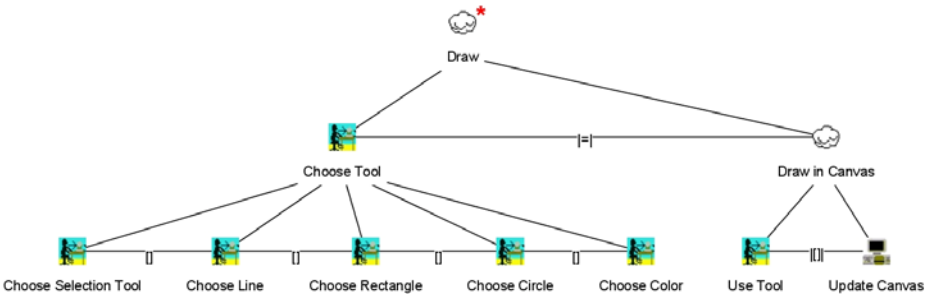

Fig. 10. Draw task of a drawing application

Consider the example in Fig. 10. This example shows the typical task in a drawing application where the user can choose a tool and use this tool to draw on a canvas using the direct manipulation paradigm. Suppose all the tasks are annotated with the same Modality Interaction Constraint: *E(MouseDirectManipulation, TouchScreenDirectManipulation).* This means the use of the *MouseDirectManipulation* modality is equivalent to the *TouchScreenDirectManipulation* modality. When we consider an environment containing a desktop computer supporting the first modality and a PDA supporting the second modality, and we apply the approach described above, all the tasks will be assigned to the device supporting the first modality because neither device supports both. However in some cases a user might prefer to have the user interface distributed where the tasks concerning tool selection are deployed on the PDA and the large canvas is displayed on a desktop computer.

Another possible scenario could be a media player where the operation buttons are displayed on the user's cell phone and the actual media is playing on the user's PDA to maximize screen space for displaying the media. In order to know whether the user would prefer a maximal distribution of the tasks rather than a maximal combination of the tasks on one particular device, the user has to specify this in his/her user profile. In the latter case the approach discussed above where modalities are transformed to queries can be applied. When a maximal distribution is desirable, some more meta-information regarding the task composition should be necessary.

One way to solve this problem is to let the designer define *Task Set Constraints (TSC)* in the task model. These constraints enable the designer to specify which tasks are desirably grouped on the same Interaction Device, and which tasks are desirably not grouped together on the same Interaction Device. Applied to the example in Fig. 10 the designer can specify the subtasks of the *Choose Tool* tasks are desirably grouped together and these same tasks are desirably not grouped with the sub tasks of the *Draw in Canvas task*. Taking into account this information during the runtime, the distribution controller can decide to prioritise the break-up of the enabled task set even if deployment is possible on a single device according to the Modality Interaction Constraint if the property of maximal distribution is chosen.

4 Related work

In this section we will discuss work related to our approach.

Berti et al. [3] describe a framework supporting migration of user interfaces from one platform to another. Unlike our goals they accentuate migratory interfaces where it is important that a user who is performing a certain task on one device can continue performing the same task on another device. In our approach we aim to distribute the subtasks a user is currently performing among several devices in the user's vicinity to exploit the available interaction resources. In their paper, they discuss three aspects to allow usable interaction of migratory interfaces that are also applicable to our focus:

- adaptability to the device's available Interaction Resources (our approach uses an ontology-based environment model);
- applying specified design criteria (allocation of devices is based on a designed augmented task model);
- and insurance of continuity of the task performance (the environment model can be updated and the allocation can be updated accordingly).

Furthermore they acknowledge the need for multimodal interaction to support smooth task execution.

Bandelloni et al. [2] also use interface migration as a starting point, but they extend their approach to support partial migration where only some parts of the user interfaces are migrated to another device. In this way user interface distribution is accomplished. Migration and partial migration are executed by taking into account the source user interface, performing runtime task analysis, and finally deploying the updated concrete user interface on the target device(s). This is in contrast to our approach where first the environment is examined to determine which interaction resources are currently available, before mapping the abstract user interface

description onto a concrete one. In this way at each state of the user interface an appropriate distribution among the interaction resources is achieved according to the available interaction resources.

Florins et al. [10] describe rules for splitting user interfaces being aimed at graceful degradation of user interfaces. Several algorithms are discussed to divide a complex user interface developed for a platform with few constrains in order to *degrade* the user interface with the purpose of presenting the interface in pieces to the user on a more constrained platform (e.g. with a smaller screen space). Although nothing is said about user interface distribution, these algorithms can be used in our approach complementary to the distribution plan discussed in 3.3.

CAMELEON-RT [1] is a reference model constructed to define the problem space of user interfaces released in ubiquitous computing environments. Their reference model covers user interface distribution, migration and plasticity [16]. This is also the problem domain of our approach. The work presents a conceptual middleware whereupon context-aware interactive systems can be deployed. The architecture is divided in several layers such as the platform layer, representing the hardware, the middleware layer, representing the software deducting the adaptation, and the interaction layer, where the interface is presented to the user in order to enable interaction with the system.

5 Conclusions and future work

In this paper we have described a development process where some decisions regarding user interface distribution and selection of modalities can be postponed to the runtime of the system. In this way the user interface can adapt to volatile environments because selection of devices and modalities accessible to the user's vicinity are taken into account. At the moment we are still performing some tests regarding the refinement of the division into partial enabled task sets. User tests are planned to find out whether the proposed information is enough to obtain a usable interface and whether more information regarding the user's preferences is needed.

In future work we will look at possibilities to extend the layout management. Since we are using XHTML in the rendering engine, Cascading Style Sheets[8] can be used to complement the layout management in obtaining a more visually attractive user interface. However, at the moment we have implemented a basic flow layout algorithm to align the graphical user interface components. We plan to use layout patterns which are commonly used in model-based user interface development, e.g. [15].

Another research direction we plan to follow in the future is the generalisation of the Modality Interaction Constraints to more general Interaction Constraints. The querying mechanism used at runtime, based on SparQL, can also be used at design time where designers can construct a more specific query than the one generated by the runtime architecture. However we have to deliberate about the drawbacks: constructing these queries is not straightforward thus a mediation tool has to be

[8] http://www.w3.org/Style/CSS/

implemented to let a designer postulate the requirements about user interface distribution in a way a more complex query can be generated.

Acknowledgements

Part of the research at EDM is funded by EFRO (European Fund for Regional Development), the Flemish Government and the Flemish Interdisciplinary institute for Broadband Technology (IBBT). The CoDAMoS (Context-Driven Adaptation of Mobile Services) project IWT 030320 is directly funded by the IWT (Flemish subsidy organization).

References

1. Lionel Balme, Alexandre Demeure, Nicolas Barralon, Joëlle Coutaz, and Gaelle Calvary. Cameleon-rt: A software architecture reference model for distributed, migratable, and plastic user interfaces. In Markopoulos et al. [11], pages 291–302.
2. Renata Bandelloni and Fabio Paternò. Flexible interface migration. In IUI '04: Proceedings of the 9th international conference on Intelligent user interface, pages 148–155, New York, NY, USA, 2004. ACM Press.
3. Silvia Berti and Fabio Paternò. Migratory multimodal interfaces in multidevice environments. In ICMI '05: Proceedings of the 7th international conference on Multimodal interfaces, pages 92–99, New York, NY, USA, 2005. ACM Press.
4. Tim Clerckx, Jan Van den Bergh, and Karin Coninx. Modeling multi-level context influence on the user interface. In PerCom Workshops, pages 57–61. IEEE Computer Society, 2006.
5. Tim Clerckx, Kris Luyten, and Karin Coninx. Dynamo-aid: A design process and a runtime architecture for dynamic model-based user interface development. In Rémi Bastide, Philippe A. Palanque, and Jörg Roth, editors, EHCI/DSV-IS 2004, volume 3425 of Lecture Notes in Computer Science, pages 77–95. Springer, 2004.
6. Tim Clerckx, Chris Vandervelpen, Kris Luyten, and Karin Coninx. A task-driven user interface architecture for ambient intelligent environments. In IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces, pages 309–311, New York, NY, USA, 2006. ACM Press.
7. Tim Clerckx, Frederik Winters, and Karin Coninx. Tool Support for Designing Context-Sensitive User Interfaces using a Model-Based Approach. In Alan Dix and Anke Dittmar, editors, International Workshop on Task Models and Diagrams for user interface design 2005 (TAMODIA 2005), pages 11–18, Gdansk, Poland, Sep 26–27 2005.
8. Joëlle Coutaz, Laurence Nigay, Daniel Salber, Ann Blandford, Jon May, and Richard M. Young. Four easy pieces for assessing the usability of multimodal interaction: the care properties. In Knut Nordby, Per H. Helmersen, David J. Gilmore, and Svein A. Arnesen, editors, INTERACT, IFIP Conference Proceedings, pages 115–120. Chapman & Hall, 1995.
9. Anind K. Dey. Providing Architectural Support for Building Context-Aware Applications. PhD thesis, College of Computing, Georgia Institute of Technology, December 2000.
10. Murielle Florins, Francisco Montero Simarro, Jean Vanderdonckt, and Benjamin Michotte. Splitting rules for graceful degradation of user interfaces. In AVI '06: Proceedings of the working conference on Advanced visual interfaces, pages 59–66, New York, NY, USA, 2006. ACM Press.
11. Panos Markopoulos, Berry Eggen, Emile H. L. Aarts, and James L. Crowley, editors. Ambient Intelligence: Second European Symposium, EUSAI 2004, Eindhoven, The

Netherlands, November 8-11, 2004. Proceedings, volume 3295 of Lecture Notes in Computer Science. Springer, 2004.
12. Fabio Paternò. Model-Based Design and Evaluation of Interactive Applications. Springer Verlag, ISBN: 1-85233-155-0, 1999.
13. Fabio Paternò and Carmen Santoro. One model, many interfaces. In Christophe Kolski and Jean Vanderdonckt, editors, CADUI, pages 143–154. Kluwer, 2002.
14. Davy Preuveneers, Jan Van den Bergh, Dennis Wagelaar, Andy Georges, Peter Rigole, Tim Clerckx, Yolande Berbers, Karin Coninx, Viviane Jonckers, and Koen De Bosschere. Towards an extensible context ontology for ambient intelligence. In Markopoulos et al. [11], pages 148–159.
15. Daniel Sinnig, Ashraf Gaffar, Daniel Reichart, Ahmed Seffah, and Peter Forbrig. Patterns in model-based engineering. In Robert J. K. Jacob, Quentin Limbourg, and Jean Vanderdonckt, editors, CADUI 2004, pages 195–208. Kluwer, 2004.
16. David Thevenin and Joëlle Coutaz. Plasticity of user interfaces: Framework and research agenda. In Interact'99, vol. 1, Edinburgh: IFIP, IOS Press, pages 110–117, 1999.
17. Jean M. Vanderdonckt and François Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems, pages 424–429, New York, NY, USA, 1993. ACM Press.
18. Chris Vandervelpen and Karin Coninx. Towards model-based design support for distributed user interfaces. In Proceedings of the third Nordic Conference on Human-Computer Interaction, pages 61–70. ACM Press, 2004.
19. Mark Weiser. The Computer for the 21st Century. In Scientific American, 1991.

Questions

Michael Harrison:

Question: You seem to have a static scheme. You do not deal with the possibility that the ambient noise level might change and therefore cause a change in the configuration. Would you not require a more procedural (task level) description to describe what to do in these different situations?

Answer: It is a static technique. Extensions to CTT have been considered that relate to similar features of ubiquitous systems and it would be interesting to see how there could be an extension to deal with dynamic function allocation.

Laurence Nigay:

Question: We developed a tool called ICARE in Grenoble, describing ICARE diagrams for each elementary task of a CTT. We found it difficult to see the link between the task level and the ICARE description, the border is not so clean. Do you have the same problem?

Answer: Depends on the granularity of the task model. When it is a rather abstract task, you have a different situation than when it is concrete. This is a factor that comes into play.