OPEN Project

STREP Project FP7-ICT-2007-1 N.216552

Title of Document:	Document about Architecture for migratory user interfaces
Editor(s):	Fabio Paternò, Carmen Santoro, Antonio Scorcia, Giuseppe Ghiani,
Affiliation(s):	CNR-ISTI
Contributor(s):	Armin Jahanpanah, Stefano Marzorati
Affiliation(s):	NEC, Vodafone
Date of Document:	February 20
OPEN Document:	D2.2
Distribution:	EU
Keyword List:	Migration, User Interface, Multi-Device Environments, Adaptation, Continuity
Version:	3.0

OPEN Partners:

CNR-ISTI (Italy)
Aalborg University (Denmark)
Arcadia Design (Italy)
NEC (United Kingdom)
SAP AG (Germany)
Vodafone Omnitel NV (Italy)
Clausthal University (Germany)

"The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. Copyright 2008 by CNR, NEC and Vodafone."

Title: Document about Architecture for migratory user interfaces	**Id Number: D2.2**

Abstract

This deliverable describes the software architecture of a solution for supporting user interface migration. It is a middleware aiming to support automatically the main functionalities of migration (adaptation and state persistence) across multiple devices with various interaction resources.

Table of Contents

1 INTRODUCTION 3

2 INDUSTRIAL PRACTISE IN CONTENT ADAPTATION 4

2.1 CONTENT ADAPTATION REFERENCE ARCHITECTURE 5
2.1.1 Content adaptation engine 5
2.1.2 Device manager 6
2.1.3 Adaptation rule manager 6
2.1.4 Customization manager 7
2.1.5 Personalization manager 7

3 OVERALL ARCHITECTURE 8

4 REVERSE ENGINEERING FOR OBTAINING A LOGICAL UI DESCRIPTION 10

5 MIGRATION TRIGGER AND EXTRACTING THE STATE OF THE UI 12

6 SEMANTIC REDESIGN FROM SOURCE TO TARGET LOGICAL DESCRIPTION 13

7 STATE MAPPING TO THE TARGET CONCRETE DESCRIPTION 17

8 FINAL UI GENERATION FROM TARGET CONCRETE DESCRIPTION TO TARGET IMPLEMENTATION LANGUAGE 19

9 MULTI-CORE PLATFORM: THE NAMUCO UI TOOLKIT 21

9.1 OVERVIEW 21
9.2 ARCHITECTURE 21
9.3 MULTICORE 23
9.4 INTEGRATION IN THE OPEN PLATFORM 24

CONCLUSIONS 26

10 REFERENCES 27

1 Introduction

This deliverable describes the software architecture supporting the migration of the user interface of an interactive application in order to allow users to continue their activities across various devices with different interaction resources. A corresponding prototype is under development.

The first version will address migration of Web applications among desktop and mobile systems. In the following we plan to address other platforms: IPhone (which differs because of the accelerometer and multi- touch screen), multi-media, and digital TV. The reasons for this choice is that the Web is the most common user interface. There are currently hundreds of millions of Web sites and it is increasingly rare to find someone who has never used a Web application. In the meantime, Web technologies have evolved in many directions: the Web 2.0, Rich Interactive Applications, Multimodal Interfaces, … Another important technological trend is the increasing availability in the mass market of many types of interactive devices, in particular mobile devices, which has enabled the possibility of ubiquitous applications. In such environments migratory interfaces are particularly interesting. They allow users to move about freely, change device and still continue the interaction from the point where they left off. Thus, in order to obtain usable migration two aspects are important: preserving the user interface state across multiple devices and adaptation to the changing interaction resources. In the following, we will describe the various software modules of our architecture supporting such two aspects.

We start with a section describing an existing solution (at industrial level) regarding the problem of content adaptation. Then, we move on to present the approach developed in the project: Section 3 provides an overview of the proposed architecture, Sections 4-8 detail its modules, while Section 9 focuses on a multi-core support included in the architecture in order to better address the specific issues raised by devices with multicore capabilities. Lastly, we draw some conclusions in the last Section.

2 Industrial Practise in Content Adaptation

One important aspect of migration is adaptation. For this aspect, there are already some existing solutions at industrial level. In this section, we briefly describe a reference architecture for such solutions, which do not address other aspects related to user interface migration, such as state persistence across various types of devices.

Advances in the capabilities of small, mobile devices, such as mobile phone and PDA (Personal Digital Assistant) have led to an explosion in the number of types of device that can now access the web. Here we refer to the Web that can be accessed from mobile devices as the mobile web.

The sheer number and variety of Web-enabled devices poses significant challenges for authors of Web sites who want to support access from mobile devices. Huge literature is available on this topic, the W3C Device Independent Working Group described many of the issues in a dedicated report.

One approach is to perform a series of optimization aimed to accelerate the browsing experience and to reduce the payload on the network. In this case no change is performed on the look and feel and layout of the content, no change in the user experience can be perceived except browsing speed.

Some protocol optimizations are:

- TCP/IP Optimization
- HTTP Optimization

Some content optimizations are:

- Document Compression/GZIP
- Image Processing/Image Quality
- Intelligent Caching
- Multiparting

Another approach to solving the problem is based around the concept of **Content Adaptation**. Rather than requiring authors to create pages explicitly for each type of device, content adaptation transforms an author's materials automatically. For example, content might be converted from a device-independent markup language, into a form suitable for the device, such as XHTML (**eXtensible HyperText Markup Language)** Basic or WML (**Wireless Markup Language)**. Similarly a suitable device-specific CSS (Cascading Style Sheet) style sheet or a set of in-line styles might be generated from abstract style definitions. Once created, the device-specific materials form the response returned to the device from which the request was made. Other examples of content adaptations are:

- Re-format and re-render any webpage to achieve optimized content, best usability and ease of navigation

- Process Complex HTML Content, Markup Transcoding (html->xhtml)
- Support Frames (IFRAMES, Nested Frames)
- Support Javascript and CSS
- Image conversion, reformatting and resizing of the image according to the screen size)
- Content Segmentation and Prioritization for a better navigation and to solve handset memory limitation

Hereafter we focus on content adaptation approach providing a reference architecture of a content adaptation solution.

2.1 Content adaptation reference architecture

The reference architecture for a content adaptation platform in an industrial setting can be composed by the following functional components:

- Content adaptation engine
- Device manager
- Adaptation rule manager
- Customization manager
- Personalization manager
- Report system
- Call Center and provisioning interface
- Billing interface

Some of these components are not core of the final purpose of the adaptation but are however necessary so that the final solution can be deployed in a real and commercial environment.

In the following we will describe the most relevant ones.

2.1.1 Content adaptation engine

This component performs the adaptation and format transformation of the web page. Its main purposes are:

- to reduce the downloading and rendering time of a web page on the device
- to adapt the presentation of the page to the actual device display dimensions and characteristics.

During the adaptation the engine performs some modifications of the page presentation, according to the rendering capabilities of the target device.

The rearrangement of the page can also imply the segmentation of the original page in several subpages so the that sizes of the adapted subpages do not exceed the phone memory in order to have a more responsive rendering because the browser needs to handle only a portion of the original page. In case of segmentation it is essential that the elements of the page are correctly determined

and managed so that for example the main section of a page is presented in the first subpage rendered.

This engine is hence responsible to recognize particular elements of the page, such as the logo, the main section of the web page, the navigation bar and the advertisements items.

The content adaptation engine perform also some adaptation of page elements that are not supported natively by the browser, some examples are: flash support, HTTPS handling, session management, Java script and ASP support.

2.1.2 Device manager

This component manages the database of the handsets supported in terms of user agent, display size, memory, browser type, character set supported, multimedia capabilities etc.

The component is also responsible to manage the general testing process of a new handset that is configured in the solution.

The need for having such dedicated handsets database is the limited reliability of WURFL (Wireless Universal Resource File) and UAProf (User Agent Profile) resources.

The WURFL is an XML configuration file which contains information about device capabilities and features for a variety of mobile devices. Device information is contributed by developers around the world and the WURFL is updated frequently but not always and without quality verification reflecting new wireless devices coming on the market. **WURFL** is part of a FOSS (Free and Open Source Software) community effort focused on the problem of presenting content on the wide variety of wireless devices.

Drawbacks to relying solely on UAProf are:

1. Not all devices have UAProfs
2. Not all advertised UAProfs are available (about 20% of links supplied by handsets are unavailable, according to figures from UAProfile.com)
3. UAProf can contain schema or data errors which can cause parsing to fail
4. There is no industry-wide data quality standard for the data within each field in an UAProf.
5. The UAProf document itself does not contain the user agents of the devices it might apply to in the schema.
6. UAProf headers can often be plain wrong. (i.e. for a completely different device)

2.1.3 Adaptation rule manager

This is a tool used to organize rules to apply to a whole web site that might be requested to be adapted in an optimized and predefined way.

2.1.4 Customization manager

This is a component responsible to perform some customization to the original web page, it may be used for example to add particular header or footer to the adapted page containing navigation or help links.

2.1.5 Personalization manager

It performs end user personalization, such as storing on the content adaptation platform bookmarks or the user history.

Call Center Interface, provisioning interface, billing interface and reporting system are additional components whose purpose does not need to be clarified.

3 Overall Architecture

Our architecture for migratory interfaces is based on a migration/proxy server. The advantage of this choice with respect to installing the necessary functionalities on the application servers is that we can concentrate them in a single server without the need for replication in the servers supporting the various possible applications. Indeed, we want to apply the migration support to a wide set of applications, and we do not want to force the application developers to use any specific authoring environment or to apply specific annotations to ease the migration process. In general, we consider that a wide set of Web applications for desktop systems already exist and they can be the target for a migration infrastructure.

Our migration infrastructure exploits logical descriptions of user interfaces, specified using XML-based languages. In such descriptions there is an *abstract* level, which is platform-independent and a *concrete* level, which refines the previous one by adding platform-dependent elements and attributes. The environment has a service-oriented architecture based on four main functionalities:

- *Reverse Engineering,* takes the existing Web pages for desktop systems and builds the corresponding logical descriptions;
- *Semantic Redesign*, this module is in charge to perform the adaptation to the target device. For this purpose it takes the abstract elements identified by the reverse engineering module and maps them into concrete elements more suitable for the target device. It also splits the source presentations into multiple presentations if they are too expensive for the interaction resources of such target device.
- *State Mapper,* once a concrete description for the target device has been obtained then the state resulting from the user interactions in the source use interface is associated with it. The abstract elements are used to identify which concrete elements in the source interface correspond to the concrete elements in the target interface.
- *User Interface Generator,* this module generates the user interface in some implementation language. One concrete description for a given platform, for example a graphical form-based interface, can be associated with various implementation languages (such as Java, XHTML, C#). The generated user interface is then uploaded on the target device.

In addition, when the host acting as a migration/proxy server passes the Web pages to the client, it adds to such pages Ajax scripts, which are used to communicate to the server the interface state accessed through DOM (Document Object Model) when the migration is triggered. The server also modifies the links and the “action” attribute of the form elements so that any reference contained in the page, when selected, is forced to pass through it.

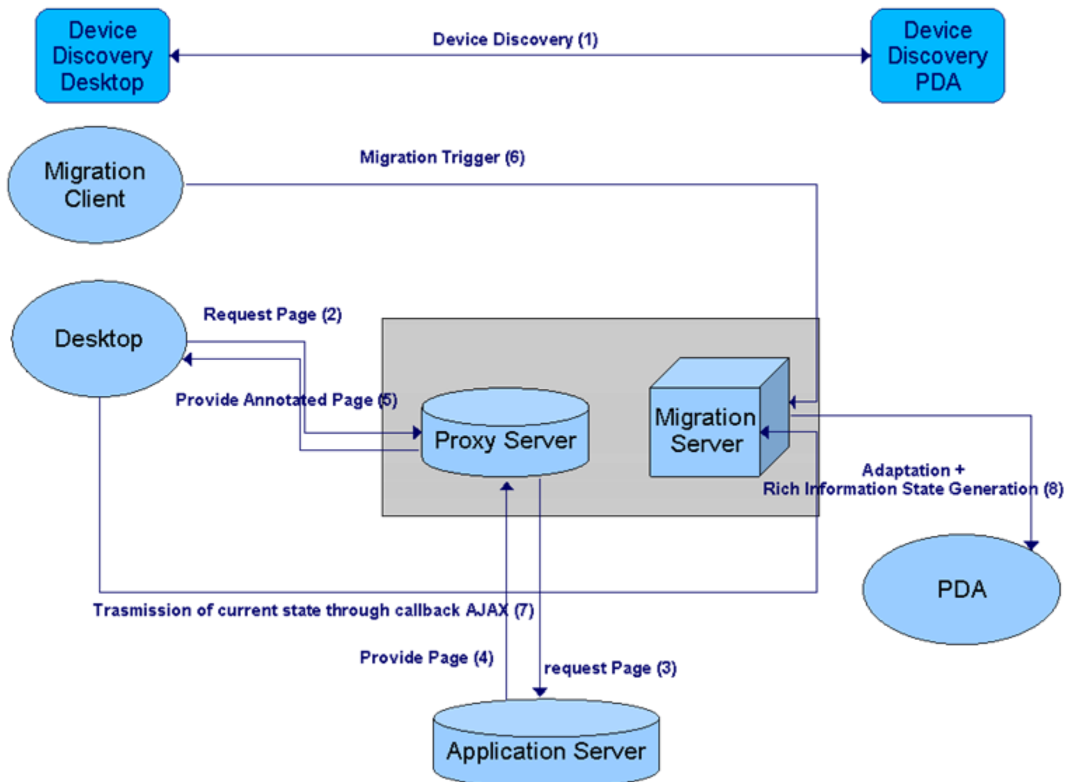

Figure 1: Architecture of the Support for Migratory User Interfaces.

All the devices that are involved in the migration should run a tiny application, called Migration Client, which is used for two purposes:

- in the device discovery (phase (1), in Figure 1), when the devices interested in migration are identified and provide information about themselves,
- to trigger migration.

When the user accesses a page (2) the request goes through the proxy/migration server (3, 4, 5, 6), which also inserts in the page some Java script that will be used to get the state of the user interaction at migration time. Users can trigger migration (6) through an interface separated from the application interface, which shows the list of available devices from which the user can select the target one. Then, the state of the current page is sent to the server, which will perform content adaption, applies the state to the newly generated version and uploads it onto the target device (8).

4 Reverse Engineering for Obtaining a Logical UI Description

The main purpose of the reverse engineering part is to analyse the implementation of the existing Web application desktop version, capture the logical design of the user interface (in terms of basic tasks supported and the ways to appropriately structure the user interface in order to accomplish them), which will then be used as the starting point for the design and generation of the interface for the target device. Some work in this area has been carried out previously. For example, WebRevEnge [1] automatically builds the task model associated with a Web application, whereas Vaquita [2] and its evolutions build the concrete (namely: platform-dependent) description associated with a Web page.

The reverse engineering module can reverse both single XHTML pages and whole Web sites. When a Web page is reversed into a presentation, its elements are reversed into different types of concrete interactors and combination of them by recursively analysing the DOM tree of the X/HTML page. In order to perform this transformation, well formed X/HTML files are needed. However, since many of the pages available on the Web do not satisfy this requirement, before reversing the page, the W3C Tidy (http://tidy.sourceforge.net/) parser is used for correcting features like missing and mismatching tags and returns the DOM tree of the corrected page, which is analysed recursively starting with the body element and going in depth. Depending on the type of node analysed, the algorithm of the reverse engineering follows one of the following branches:

- The X/HTML element is mapped onto a concrete *elementary interactor*. This is a recursion endpoint. The appropriate interactor element is built and inserted into the logical description. For example, DOM nodes corresponding to the X/HTML tags <img> (image), <a> (anchor) and <select> (selection) cause the generation of concrete objects of type respectively *image*, *navigator* and *selection*. The properties of the objects in the Web implementation are also used to fill in the attributes of the corresponding concrete user interface elements, so that this information can be elaborated for producing an appropriate element in the target device, out of the peculiarities used in the source Web page. For instance, the italic attribute of a text concrete element is set to true although in the X/HTML implementation it might appear as either <i> (italic text style) or <em> (emphasysed text), which are two different manners for highlighting an element.
- The X/HTML node corresponds to a concrete *composition operator*. In this case, the proper composition element is built and the function is called recursively on the X/HTML node subtrees. The subtree analysis can return both elementary interactors and composition of them. In both cases the resulting nodes are appended to the composition element from which the analysis started. For example the node corresponding to the tag <form> is reversed into a Relation composition operator and <ul> (unordered list) into a Grouping. Depending on the considered node to be reversed,

appropriate attributes are also stored in the resulting element at the concrete level (e.g. typical X/HTML desktop <ul> lists will be mapped at the concrete level in a grouping expression using bullets listed following a vertical positioning).

- The node does not require the creation of an instance of interaction in the concrete specification (for example, if in the Web page there is the definition of a new font, no new element is added in the concrete description). If the node has no children, no action is taken and we have a recursion endpoint (this can happen for example with line separators such as <br> tags). If the node has children, each child subtree is recursively reversed and the resulting nodes are collected into a grouping composition which is in turn added to the result.

In the reverse process, the environment first builds the concrete description and then the abstract one. In TERESA XML [3] the concrete descriptions are a refinement of the abstract one, which means that they add a number of attributes to the higher level elements defined in the abstract descriptions. Thus, the process for reversing a concrete description into the corresponding abstract one consists in removing the lower level details from the interactor and composition operators specification, while the structure of the presentations and the connections among presentations remain unchanged. In practice, there is a many-to-one relation between the elements of the concrete user interface and the abstract user interface (both for the interaction objects and the composition operators): the concrete user interface indicates several ways to refine and abstract element for the platform under consideration. Therefore, it is easy to derive the abstract logical objects corresponding to the different concrete interaction objects. For instance, considering the desktop platform, we can have at the concrete level a text_link, an image_link and a button, which are all possible refinement options for a navigator interactor. However, since all such elements share the same objective, which is navigating between different parts of the user interface, the result of reversing each of these concrete elements will be an abstract navigator object.

5 Migration Trigger and Extracting the State of the UI

Migration can be triggered by the user, alternatively it can be automatically triggered by the smart environment when some events (such as very low battery/connectivity) are detected, or even a mixed solution can be envisaged, in which the environment suggests possible migrations based on the devices available and then the user decides which migration actually trigger.

In case of *user* migration trigger, the user selects the device on which the UI should migrate by interacting with a Migration Client, a thin application on the client device which provides information regarding the device characteristics and therefore allows the user to select the target device.

In case of *automatic* migration trigger, it is supposed that the Trigger Manager can identify situations where it would be better for the user to change device. This can be decided according to a number of rules that could consider the device descriptions (which include for example an indication whether the device is personal or it can be used by some/all members of a group), the state of the device (for example, if it is a single user device and it has been already taken by another user then it cannot be considered available for migration), information regarding where the device is located (in the case of stationary devices it is the corresponding room).

In both cases (user or automatic trigger) when a request for migration to another device is triggered, the environment has to extract and collect information regarding the current *state* of the user interface: this is a precondition to ensure state persistence during migration, necessary to support task continuity across multiple devices. The process of state extraction includes the identification of the last element accessed by the user in the source device version of the application, and it basically depends on the user’s inputs performed till the time when the migration is triggered. More in detail, the state refers to the information entered by the user (e.g. fields which have been already filled by the user), but also other pieces of information can be important for maintaining the information associated to the user session (e.g. the history of the pages the user has already visited, cookies, ..). It is worth pointing out that, in order to be able to collect such data deriving from e.g. user’s interactions, the pages have to be slightly modified before being actually used by the user. Indeed, when the clients access the Web pages, their requests are in reality captured by a proxy server, which downloads the pages from the application servers, and it annotates them with scripts that support capturing the UI state. After having performed this step the page is able to capture (and continuously update) the current state of the UI resulting after the various user interactions and, when the migration is activated, send such collected data to the Migration Server.

6 Semantic redesign from source to target logical description

The semantic redesign transformation changes the logical description of a user interface for a given platform into a logical description for a different platform. The aim is to support a similar set of tasks and communication goals but provide input for obtaining an implementation that adapts to the interaction resources available.

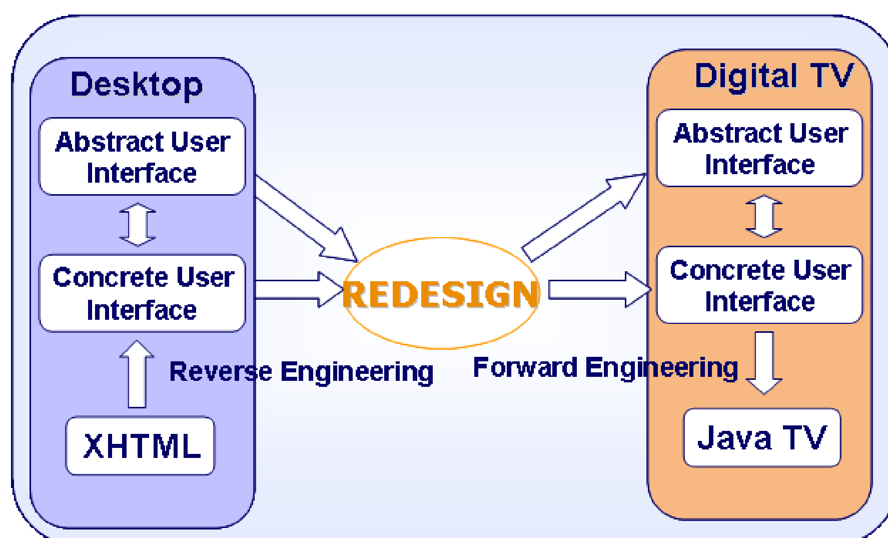

Figure 2: Architecture of the Adaptation Part for the Digital TV.

In particular, the redesign module analyses the input from the desktop logical descriptions and generates an abstract and concrete description for the target platform, from which it is possible to automatically generate the corresponding implementation. Figure 2 shows the process in the case of adaptation from desktop to digital TV.

Figure 3 shows the various phases of semantic redesign in the case of desktop-to-mobile transformations. After having parsed the CUI (Concrete User Interface) for desktop platform (see "Parsing CUI" rectangle in Figure 3) there are three main steps:

- transforming the desktop logical interface into a mobile logical interface while preserving the semantics of the various activities,
- calculating the resulting cost in terms of resources,
- Possible splitting of the logical interface into presentations that fit the cost sustainable by the target device. This phase can occur or not, depending on the characteristics of the device at hand.

The "Generator CUI" rectangle in Figure 3 refers to the phase in which the obtained CUI is transformed into a Final User Interface, by using the constructs of

the particular implementation language considered (this phase will be better detailed in Section 8).

In the first transformation the concrete elements of the desktop description are substituted by concrete elements supported by the mobile platform (for example, a radio-button with several elements can be replaced with a pull-down menu, which occupies less screen space). In this transformation, further rules are applied to adapt the elements of the user interface to the characteristics of the new platform even when the transformation from the source platform to the target platform does not change the type of interactor. For instance, images originally displayed in the source (desktop) platform are resized according to the screen size of the target (mobile) device, while keeping the same aspect ratio. In some cases they may not be rendered at all because the resulting resized image would be too small or the mobile device does not support them. Text and labels can be transformed as well, since they may be too long for mobile devices. In converting labels, we use conversion tables to identify shorter synonyms or abbreviations.

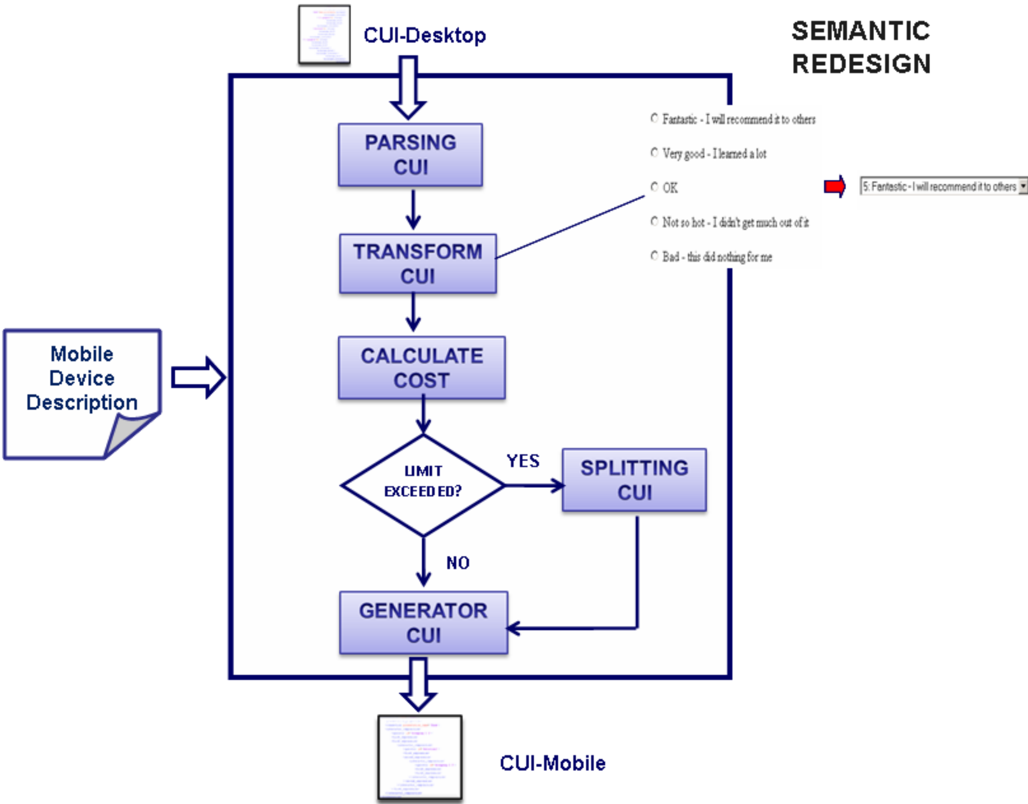

Figure 3: Desktop-to-Mobile Semantic Redesign.

In order to automatically redesign a desktop presentation for a mobile device, we need to consider semantic information and the available resource limitations. If we only consider the physical limitations, we may end up dividing large pages into smaller ones that are not meaningful, since they result from considering only

some aspects, e.g. the available space in the screen of the target device. To overcome this problem, we also consider the composition operators indicated in the logical descriptions. Indeed, our algorithm tries to maintain in the same presentation interactors that are collected together through some composition operators. For instance, suppose that a user has to specify her personal information for receiving updates/news from a low cost airline newsletter: in this case all the user interface objects supporting the editing/showing of the user's personal information are logically grouped together. Such a logical grouping should be reflected into an adequate presentation in the final user interface in such a way to convey the semantic relationship that exists among the various objects also through opportune visualisations so that the user can easily understand that they altogether contribute to achieve the same specific (sub-)goal and they are all grouped together (e.g.: on GUI a typical technique is using a graphical fieldset for grouping together all the grouped fields). Thus, the environment aims to preserve the communication goals of the designers and obtain interfaces that are easy to use because each presentation is composed of objects that are semantically related to each other in that they all contribute to achieve a specific goal (or subgoal). In addition, the division of pages according to the logical completion of a task (or a subtask) also allows for maintaining the consistency of user interfaces through different devices, which is especially convenient for users who interact with the same application through different devices (as happens with migratory user interfaces). Page splitting requires a change in the navigation structure with the need for additional navigator interactors for accessing the newly created pages. More specifically, the algorithm for calculating the costs and splitting the presentations accordingly is based on the number and cost of interactors and their compositions. The cost is related to some device resources that are needed in order to support a specific interactor within a presentation. For instance, in Graphical User Interfaces a meaningful dimension to be taken into account for any user interface object is the number of pixels that are needed for displaying the object itself, for a textual label a relevant dimension can be the font size used, etc.. After the initial transformation, which replaces the desktop concrete elements with mobile concrete elements (for example, a text area for the desktop platform could be transformed into a simpler text edit on the mobile platform), the cost of each presentation is calculated. If such a cost fits the cost sustainable by the target device, then no further processing is required. This means evaluating whether in a graphical presentation the screen area occupied by a set of user interface elements can still be appropriate (in terms of usability) when displayed on the new device. The optimal case is when the user interface displayed in the new target device does not require any scrolling (neither horizontal nor vertical in order to be shown). Otherwise (namely: if the set of UI objects in the new target device forces the user to use scrolling movements beyond a certain tolerance threshold), the presentation is split into two or more pages following this approach: the cost of each composition of elements is calculated. The one with the highest cost is associated to a newly generated presentation and is replaced in the original presentation with a link to the new presentation. Thus, if the cost of the original presentation after this modification is under the maximum allowed cost, then the

process terminates, otherwise it is recursively applied to the remaining compositions of elements. In the case of a complex composition of interface elements, which might not be entirely included in a single presentation because of its high cost for the target device, the algorithm aims to distribute the interactors equally amongst presentations of the mobile device, by creating multiple presentations in which UI objects are distributed by following the sequence in which they appear in the original presentation (from left to right and from top to bottom).

If a certain element is not supported by the new platform, the algorithm tries to substitute the interactor with a simpler one which is supported by the platform (for instance if a device does not support videos, the engine could try to provide e.g. first frame of the video) otherwise the element is removed.

The cost that can be supported by the target mobile device is calculated by identifying the characteristics of the device through the *user agent* information in the HTTP protocol, which can be used to access more detailed information in a local XML repository with device descriptions obtained through WURFL (wurfl.sourceforge.net/), a device description repository containing a catalogue of mobile device information. Initially, we considered UAProfiles but sometimes such descriptions are not available or are wrong and they require an additional access to another server where they are stored. As already mentioned, examples of elements that determine the cost of interactors are the font size (in pixels) and number of characters in a text, and image size (in pixels), if present. One example of the costs associated with composition operators is the minimum additional space (in pixels) needed to contain all its interactors in a readable layout. This additional value depends on the way the composition operator is implemented (for example, if a grouping is implemented with a fieldset or with bullets the costs are associated with the space taken by the surrounding rectangle or the bullets). Another example is the minimum and maximum interspace (in pixels) between the composed interactors. However, it is worth pointing out that, in order to manage some possible splitting issues, we decided to have some tolerance threshold within the algorithm. This has to be done in order to avoid some awkward situations such as having a presentation with only one element which occupies a small portion of the presentation itself: in this case this element will be included into another presentation, even if this means going beyond the supposed cost of such a presentation.

The semantic redesign module can take into account the different features of the modalities that can be supported. For example, in vocal interfaces, it is important that the system always provides feedback when it correctly interprets a vocal input and it is also useful to provide meaningful error messages in the event of poor recognition of the user's vocal input. At any time, users should be able to interrupt the system with vocal keywords (for example "menu") to access other vocal sections/presentations or to activate particular features (such as the system reading a long text).

7 State mapping to the target concrete description

The state refers to the information entered by the user, but also other pieces of information that can be important for the user session (such as the cookies). State persistence is necessary to support task continuity across multiple devices. In particular, when a request for migration to another device is triggered, the environment detects the state of the user interface as modified by the user input (elements selected, data entered, …), and identifies the last element accessed in the source device. For this purpose, when clients access the Web pages, their requests pass through a proxy server, which downloads the pages from the application servers, and also adds to them the scripts able to capture the UI state and communicate it to the server. Such scripts through a polling-based monitoring mechanism, implemented through an Ajax script, determine whether or not a migration was triggered by the migration client. When the user sends a migration request an AJAX callback function is automatically activated, which sends the DOM (containing the state of the current page) collected through a specific script. The information is collected in a string formatted following an XML-based syntax and sent to the server. This mechanism was chosen because only an application running on the browser in the client device can access the application DOM, and the AJAX Script can transmit the data without requiring any explicit action from the user.

As already mentioned, the Migration Client should be running in the source device in order to let the user select the migration target and trigger the migration. When the migration is triggered, the migration client sends the IP of the source and target devices to the migration server.

Then, the migration platform will first associate the content state of the page on the source device to the concrete description of the version for the target device. This is obtained through the State Mapper module, whose purpose is to update the CUI for the target device (which has been produced by the Semantic Redesign module) with latest information regarding the state of the user interface contained in the DOM of the source page just before migration. The corresponding elements in the two logical descriptions are easy to identify because each object of the CUI has a unique identification label (ID), which is the same of the corresponding XHTML/DOM element from which that CUI element was generated by the reverse engineering process. One possible complicating factor is when the semantic redesign has transformed a specific concrete object C1 (for a specific platform) into a different concrete object for the target platform, C2. In this case, since the same ID is maintained among the two concrete objects C1 and C2, the association between the concrete object and the corresponding DOM element is still straightforward (the same ID is maintained). Nevertheless the State Mapper may require a further step, that is, adapting the value of the DOM element to specify the new concrete object. For instance, it might happen that, as a result of the semantic redesign process, a radiobutton element was translated into a pull-down menu element. Therefore, the values included in the specification of the radiobutton element (e.g.: the different items of the radiobutton) have to be

appropriately adapted and used to fill in the specification of the pull-down menu element.

8 Final UI Generation from Target Concrete Description to Target Implementation Language

This phase is in charge of building the Final User Interface (FUI) in an implementation language suitable for the target device, starting with a concrete description of the user interface for the platform considered (the so-called CUI or Concrete User Interface): depending on the considered interaction platform and the specific implementation language, a particular transformation is selected by the UI Generator.

More in detail, the algorithm underneath the UI (User Interface) Generator starts with parsing the target Concrete User Interface description which has a tree-like form, therefore the starting point is represented by the *root* node. Such a root node is generally a CUI "presentation" element which roughly will correspond in the final implementation language to the "container" of the various final UI objects. Therefore, the first step is to create the transformation between the root CUI node onto a construct of the final UI of the target device, and using the primitives of the final implementation language considered. For instance, if XHTML language is considered as target language, the CUI presentation node will be translated onto a <html> container node. After creating this, the next step will be to progressively populate such container by appending UI elements to it. In order to do this, UI Generator has to recursively call a procedure that analyses the type of CUI element that is encountered as a child (*elementary* CUI object or composed expression of CUI objects), transforms it onto a construct of the Final implementation language used, and adds such resulting construct to the final UI that will be incrementally created in this way.

As we said, two types of elements can be analysed by the UI Generator during the visit of the tree-shape CUI:

- In case of *elementary* CUI object, the UI generator will carry out a transformation of such a CUI object in the corresponding UI element of the implementation language considered (e.g. Java, XHTML, C#, ..). For instance if we consider as target implementation language XHTML, a CUI listbox element can be mapped onto a <select> element, which is composed of a number of <option> elements defining the various items within the listbox. However, if we consider another implementation language (e.g. Java, or C), for the same CUI listbox element we will produce a different implementation.
- In case of *composition* node (e.g. multiple elements combined by some concrete techniques for grouping them like fieldsets, lists, ..), the associated compositional techniques will be implemented in the final implementation language used. For instance, if at the implementation level we consider XHTML language, in this case when passing from the concrete level to the implementation one, the concrete level primitives are mapped into XHTML constructs, for instance concrete grouping techniques can be mapped onto tag <fieldset> (which identifies a group of

form elements as being logically related) and <div> (with various attributes, which defines a division or a section in a XHTML document), which can be used as composition techniques at the XHTML implementation level.

9 Multi-Core Platform: The NaMuCo UI Toolkit

9.1 Overview

Namuco (Native Multicore UI Toolkit) is a library of Java and C++ classes that enable application programmers to implement graphical user interfaces while sensibly exploiting multicore capabilities of the different devices that participate in the migration environment.

The goal is to provide simple, but good-looking UI elements which allow developers to build UIs similar in look and feel to existing interfaces on mobile devices (e.g. Apple iPhone), game consoles (e.g. Xbox™ Live! Arcade UI, Playstation™ Store) or set-top boxes (e.g., Channel lists, program guides) and to distribute the workload of computationally intensive UI functions evenly on to the respective number of CPU cores on the different devices.

The framework is well-suited for multimedia applications and will provide particularly a compositing and effect layer allowing for features such as rotated and scaled windows, transparency and animated widgets.

To achieve these effects and to provide best possible performance, the library will use multiple threads internally to leverage multicore CPUs.The innovative adaptive load-balancing implementation always utilizes the full processing performance and automatically adapts to changing numbers of available cores at runtime. These features are a key advantage with respect to application migration.

The second important aspect is ease of adaptation and migration of GUI layouts to different devices: For example, it is planned that Namuco provides layout manager classes which make it possible to adapt the GUI layout to different screen resolutions in a convenient and flexible way.

Memory footprint and run-time resource requirements are targeted to be very low in order to ensure smooth interaction also on mobile and embedded devices.

9.2 Architecture

To maximize integration and interoperability with various other OPEN components and systems, the toolkit is implemented in the Java programming language. Additional C++ code is also used in order to gain maximum performance for multimedia applications. E.g. the UI library will use NLE-IT's C++ Task Programming Interface (TPI) to distribute workload over all available CPU cores. To integrate the native C++ functions into the Java code, we use the Java Native Interface (JNI).

Namuco will be based on existing Java GUI toolkits: We will build a prototype for the review meeting based on Java's Abstract Window Toolkit (AWT).

Currently, we evaluate if a later switch to the Standard Widget Toolkit (SWT) would be beneficial as SWT already utilizes JNI to call native C++ functions for drawing of widgets. Thus, we expect to be able to hook-in multicore extensions there.

The extension of AWT will be done via wrapper classes and the delegation concept, which also makes it possible to reuse and extend the existing event handling functionality to allow for event handlers in native code etc. There will also be Java classes for handling of animations and transitions of GUI elements.

Namuco consists of several parts (see diagram in Figure 4) :

- The Java interface that is used by application developers to create and manage the UI elements of their application.
- The native C++ side (a dynamic library, "DLL") providing performance-critical functions using multicore implementations. For this, it will utilize our TPI multicore programming library internally. Most likely all higher-end drawing functions (like crossfading/blending, animations, etc.) will be implemented here.
- The JNI "glue" code that is located between the above layers and propagates the Java interface calls down to the native C++ side
- A graphics display subsystem ("Renderer") that draws lines, boxes, bitmaps, text, etc. This is used to draw the UI widgets like Buttons, Checkboxes, etc. This part can either be implemented using native OS functions or by a custom cross-platform graphics library based on memory buffers. Currently we use X11.

Viewed from a higher level, Namuco provides the following functions:

- Widget implementations (for windows, buttons, lists, etc.)
- handling state and updates, messaging logic, and event handling
  - E.g. when the user moves the mouse, clicks a button, scrolls a window, etc., the corresponding event is stored in a message queue
  - On update, widgets get their events from the queue and react to these events, i.e. change their state
- interaction of events, widgets, and animations
- rendering and compositing
- transformation (rotation, scaling)

The framework is built as a class hierarchy. In order to allow application programmers the object-oriented creation of user interface elements, the UI toolkit provides a set of classes like

- Common types and objects (Point, Rectangle, Color, Image, …)
- Windows and Widgets (Button, List, Scrollbar, ...)
- Event Listener interfaces
- Renderer and Compositor

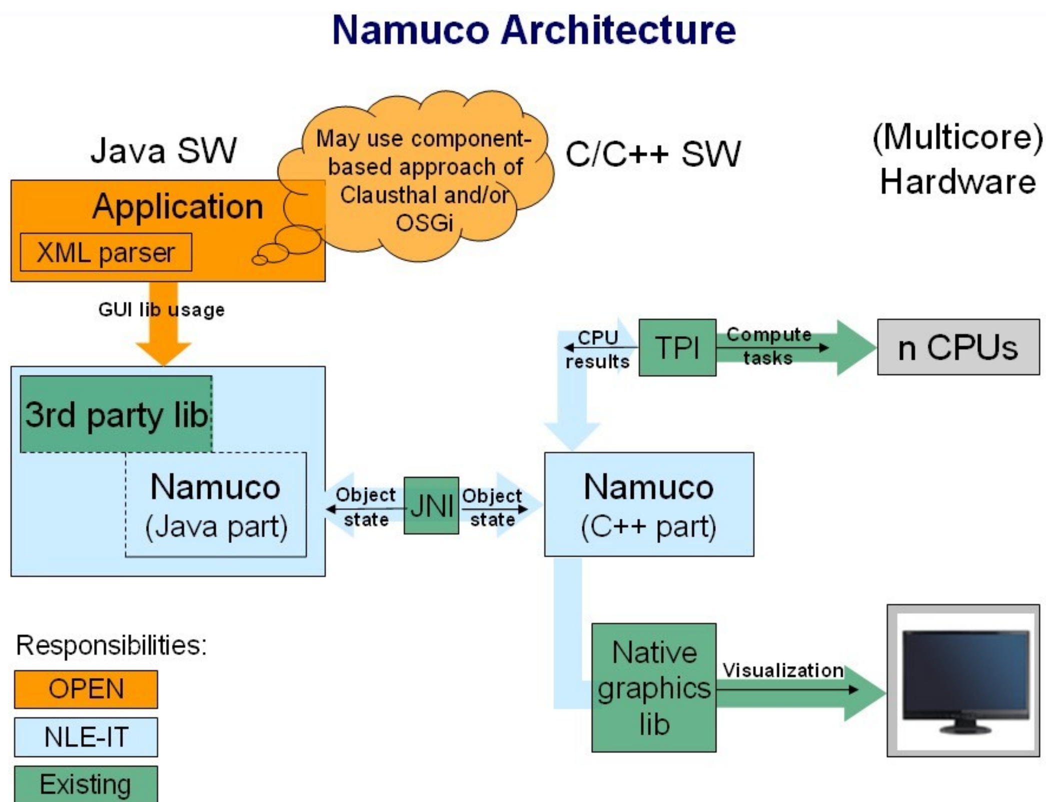

Figure 4: The Namuco Architecture

9.3 MultiCore

In order to provide the best possible performance, Namuco will use multiple threads internally to leverage multicore CPUs. For this, we plan to use NLE-IT's Task Programming Interface (TPI), which provides an innovative adaptive load-balancing implementation that enables applications to always utilize the full processing performance provided by the CPU.

Especially the system automatically adapts to changing numbers of available cores at runtime, which is a key advantage with respect to application migration.

It also supports disabling of CPU cores at runtime, without interruption of the running application and with no special requirements on the programmer's side. This behaviour is especially useful for mobile and embedded devices as application performance can be adapted with respect to power consumption and battery life.

In the first instance we will use this multicore system to accelerate rendering of GUI transitions and animation of certain elements, for example crossfading of bitmaps (alpha blending) using multiple CPU cores.

For this, bitmaps and screen regions are split into tiles which are processed in parallel on multiple CPU cores using TPI.

9.4 Integration in the OPEN Platform

As CNR-ISTI's adaptation tool provides an XML description of the concrete UI layout for the target device as output after the UI migration process (as it has been described in previous sections), a Java application that uses Namuco can use this XML description to build the GUI layout on the fly (when the application starts up or at runtime).

CNR-ISTI already has a UI Generator that could be extended to support this functionality for Namuco. In this case, the UI Generator (called "XML parser" in Figure 4 and Figure 5) creates the Java Namuco GUI elements and layout based on the XML description for the target application. To allow this, the UI Generator has to be integrated into the Java application by the application programmer, i.e. there needs to be a Java interface for the UI Generator.

After this step, attaching the application logic to the created UI widgets can be realized in several ways:

For example, the UI Generator could provide the application with a DOM[1]-like tree or a hash table that contains references to the created UI widgets. The application can then use a certain widget by looking up the corresponding reference.

[1] Document Object Model: A hierarchical representation of document elements like text fields, buttons, or images

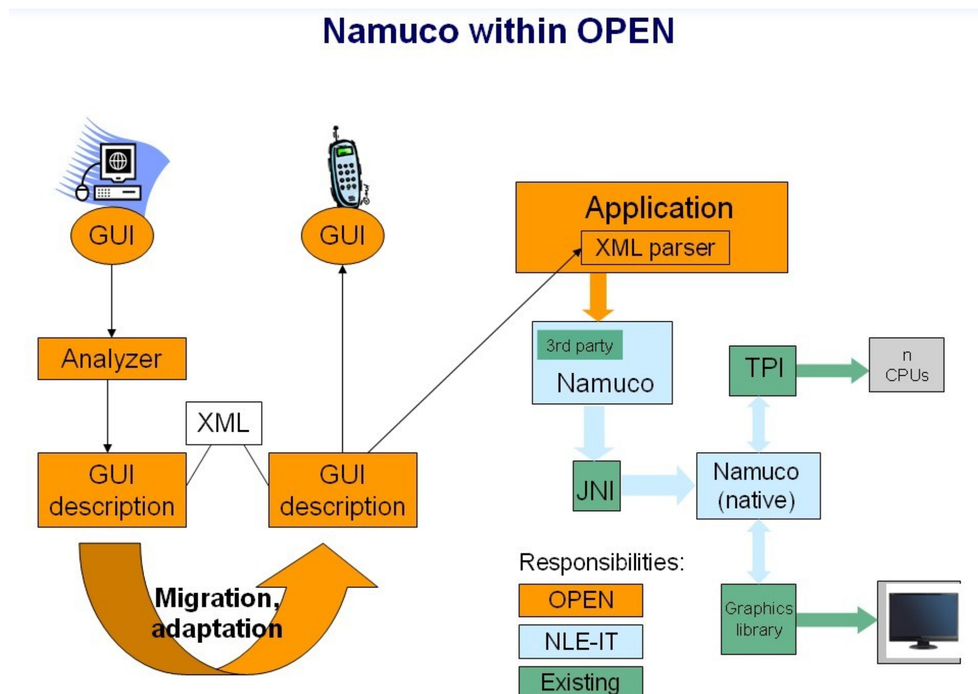

Figure 5: Integration of Namuco within OPEN

We are also analyzing how to integrate the Namuco GUI library with the other contributions in the project following a different approach based on the use of component-based Java applications. In this scenario, applications are composed of modules that can interact using a special middleware layer that also could facilitate dynamic reconfiguration. In general, one should keep in mind, that although a few tasks regarding management of state and logic are supported when using Namuco, the major part of these tasks has to be implemented on the application side.

However, since the GUI library's knowledge about the application state is very limited, it cannot perform migration of data and state information stored inside GUI widgets reliably. Therefore, it is the responsibility of the application to get the state values from the respective widgets and hand them to the OPEN platform.

There are several possibilities to implement transfer of state information in a Java application: for example, one could use a relational database system to store the necessary state information for migration in a central (network) place, which is also accessible by the target application.

Conclusions

This deliverable has described the first version of the software architecture for supporting user interface migration. The solution proposed assumes the existence of an initial Web desktop version of the user interface considered, and it is then able to dynamically create a version for a different platform, with the user interface state updated with the results of the user interactions with source version. We have described how the desktop version can be adapted to a mobile version, obtaining a version which has interaction techniques requiring less space and with original large pages split into more manageable pages. This solution can generate user interface implementations in languages, which are not web languages, such as Java. We plan to address this possibility in the second year, if there is interest in the OPEN consortium. In addition, we plan to consider its extension to support also multi-user Web applications.

10 References

[1] Paganelli, L. and Paternò, F. (2003) A Tool for Creating Design Models from Web Site Code. International Journal of Software Engineering and Knowledge Engineering, World Scientific Publishing 13, 2, 169-189.

[2] Bouillon, L. and Vanderdonckt, J. (2002) Retargeting Web Pages to other Computing Platforms. Proceedings of WCRE'2002, Richmond, Virginia, 29 October-1 November, pp. ~339-348, IEEE Computer Society Press, Los Alamitos.

[3] Mori, G., Paternò, F., Santoro, C., Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions. IEEE Transactions on Software Engineering (August 2004, 30,8, pp.507-520)

[4] Wagner, J., Jahanpanah, A., and Träff, J. L. 2008. User-Land Work Stealing Schedulers: Towards a Standard. In *Proceedings of the 2008 international Conference on Complex, intelligent and Software intensive Systems - Volume 00* (March 04 - 07, 2008). CISIS. IEEE Computer Society, Washington, DC, 811-816

[5] Wagner, J., and Jahanpanah, A. 2007. Implementing a Work-Stealing task scheduler on the ARM11 MPCore. In ARM Developers Conference 2007