Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

The Journal of Systems and Software xxx (2003) xxx–xxx

The Journal of Systems and Software

www.elsevier.com/locate/jss

Roam, a seamless application framework

Hao-hua Chu *, Henry Song, Candy Wong, Shoji Kurakake, Masaji Katagiri

DoCoMo Communications Laboratories USA, Inc., 181 Metro Drive, Suite 300, San Jose, CA 95110 USA

Received 23 December 2002; accepted 28 December 2002

Abstract

One of the biggest challenges in future application development is device heterogeneity. In the future, we expect to see a rich variety of computing devices that can run applications. These devices have different capabilities in processors, memory, networking, screen sizes, input methods, and software libraries. We also expect that future users are likely to own many types of devices. Depending on users' changing situations and environments, they may choose to switch from one type of device to another that brings the best combination of application functionality and device mobility (size, weight, etc.). Based on this scenario, we have designed and implemented a seamless application framework called the *Roam system* that can both assist developers to build multi-platform applications that can run on heterogeneous devices and allow a user to move/migrate a running application among heterogeneous devices in an effortless manner. The Roam system is based on partitioning of an application into components and it automatically selects the most appropriate adaptation strategy at the component level for a target platform. To evaluate our system, we have created several multi-platform Roam applications including a Chess game, a Connect4 game, and a shopping aid application. We also provide measurements on application performance and describe our experience with application development in the Roam system. Our experience shows that it is relatively easy to port existing applications to the Roam system and runtime application migration latency is within a few seconds and acceptable to most non-real-time applications.

1. Introduction

The era of PC-dominated applications is about to end. Now-a-days, we see widespread use of mobile devices that have sufficient computing and networking capabilities to run a variety of feature-rich applications. They come in a variety of form factors, including smart pagers, cell phones, PDAs (Pocket PCs), handheld PCs, car navigation systems, and notebook PCs. An average person may already own multiple such devices. It is expected that in the near future, the number of such devices will far exceed the number of desktop PCs. As a consequence we expect that the predominant software platform will shift from PCs to mobile devices. This creates a challenge for developers to build applications that can run on different mobile device platforms. To address this challenge, we believe that there is a need to provide an application framework for building multi-platform mobile applications.

We also expect a mobile user may choose to switch from one type of device to another type of device, in the middle of using an application, in order to access necessary application functionality or to become more mobile, based on changing situations, environments, or needs. For example, a user starts planning a vacation online using a desktop computer in his/her office. In the middle of planning, he/she receives an urgent call and must leave the office for a meeting at a remote site. The user would like to continue planning the trip on the bus or during break time between meetings. Given the need for mobility, he/she switches to a lightweight, mobile device (PDA or cell phone) away from the office. Based on this scenario, we believe that when mobile users switch devices, there is a need to allow them to move any running application effortlessly between devices.

Based on these two needs, developers will be required to write *seamless applications*. We define a seamless application to be an application that can run on heterogeneous devices and migrate at runtime among heter-

* Corresponding author.
E-mail addresses: haochu@docomolabs-usa.com (H.-h. Chu), csyus@docomolabs-usa.com (H. Song), wong@docomolabs-usa.com (C. Wong), kurakake@docomolabs-usa.com (S. Kurakake), katagiri@docomolabs-usa.com (M. Katagiri).

0164-1212/03/$ - see front matter © 2003 Elsevier Science Inc. All rights reserved.
doi:10.1016/S0164-1212(03)00052-9

ogeneous devices. There has been an abundance of research in the area of mobile agents that allow applications to move from one host to another at runtime with little or no loss of execution states (Aramira, 1999; Acharya et al., 1997; Banavar et al., 2000; Fünfrocken, 1998; Gray et al., 1997; Lange and Oshima, 1998; Milojicicc et al., 1998; Mitsubishi Electric ITA Horizon Systems Laboratory, 1998; ObjectSpace, xxxx; Peine and Stolpmann, 1997; Strasser et al., 1996; White et al., 1997). These systems are mostly built on top of the Java virtual machine (JVM), which provides the advantage of a common runtime environment. However, these systems make an important assumption of *device homogeneity*. For example, the underlying hosts/devices must be PC or PC-like devices with enough hardware/software (HW/SW) capabilities to run the standard JVM. Given this device homogeneity assumption and Java's "Write Once, Run Anywhere" programming model, it is relatively straightforward to realize runtime application migration among similar devices. However, device homogeneity is not a realistic assumption in today's mobile computing environment where there exists a wide range of mobile devices and information appliances with different HW/SW capabilities. For an example, a cell phone or a PDA in comparison to a desktop computer has a slower processing speed, less memory, little or no permanent storage, slower and unreliable network connectivity, smaller screen size, and limited input capabilities. To address device heterogeneity, Java introduces mutually incompatible Java 2 Micro Edition (J2ME) profiles and configurations for different classes of mobile devices. As a result, Java's "Run Anywhere" ideal does not apply to the device heterogeneous environment.

1.1. Challenges

The focus of the Roam system is to address the *device heterogeneity* problem in runtime application migration. Fig. 1 shows different aspects of this problem that a Java application developer may face when developing a seamless application.

- *Incompatible Java virtual machine configurations and profiles:* Java provides incompatible virtual machine (VM) configurations and profiles for different classes of mobile device. These VM configurations and profiles support different subsets of APIs that application developers can use to build applications. A device with more capable HW (e.g., more memory and faster processors) can typically support a VM with a more extensive Java class library. If an application uses APIs provided by a Java profile or configuration of a more capable platform, this application may not be able to run on less capable platforms that do not support these APIs.

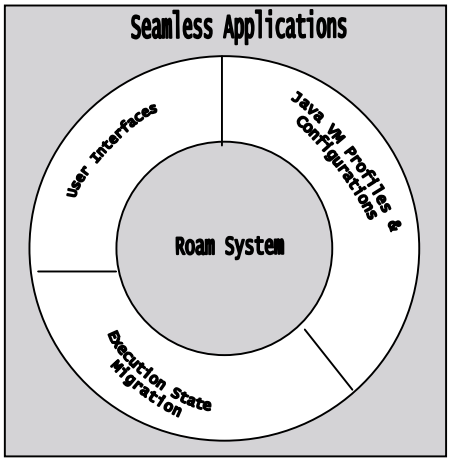

Fig. 1. Challenges for application migration in heterogeneous device environment.

- *Execution state migration:* In a runtime application migration from a source device to a target device, the execution state of an application needs to be captured on the source device, transferred to a target device and restored. The application execution state includes heap, stack, network sockets, file I/O state, and other state information. Device heterogeneity brings an additional challenge in execution state migration. Given the incompatible configurations and profiles on different device platforms, application developers may choose to create multiple device-dependent implementations (M-DD) for an application component (e.g., one implementation for each specific device platform), or they can use an automated tool that can transform an application component into code that can be run on each platform. This means that, if an application has two different implementations (either hand-coded or transformed) on two different platforms, the captured runtime execution state on the source device may not correspond to the different implementation on the target device.
- *User interface:* Different platforms have different *display sizes* and *input methods*. A DoCoMo 503i cell phone has an approximately 120×130 pixel display and a small numeric keypad; a Compaq iPaq PocketPC has a 320×240 pixel display and a stylus; and an average notebook PC has a 1024×768 pixel display, a keyboard and a pointing device. Display size can change the presentation and layout of the GUI components. A large display (e.g., desktop PC) can accommodate big GUI components, and many GUI components can be presented at once in the same window or page. On the other hand, a small display can only accommodate highly compact GUI components, and only a few GUI components can be displayed at once. In addition, the constraints of the devices' display sizes and input methods can affect

the range of tasks that are appropriate on each platform. For example, a notebook PC or a Blackberry pager with a keyboard is suitable for performing tasks that involve text entry. On the other hand, a cell phone with only a numeric keypad is prohibitively difficult to use for tasks involving text entry.

1.2. General approach

To address the challenges described in the previous section, the Roam system provides a seamless application framework for developers to build *resource-aware* seamless applications. By resource-aware, we mean that applications are aware of the underlying device capabilities (Java VM configuration and profile, display size, input method, memory, network, etc.), which may change when they migrate between devices. In order to migrate applications between devices with different capabilities, *adaptation* is needed. The Roam system utilizes the following three adaptation strategies:

- *Dynamic instantiation:* An application is divided into M-DD components that perform the same function, but each component is implemented/re-implemented for different platforms. At migration time or runtime, the set of components that best fit the target device capabilities is selected for instantiation. For example, the Java VM configuration and profile supported on the target device is considered as a part of the target device capabilities in the Roam system. A developer can provide two device-dependent implementations of the same component, one for cell phones that support the J2ME DoCoMo Java (DoJa) profile, and another for notebook PCs that support the Java 2 Standard Edition (J2SE) VM. When an application is started on or migrated to a cell phone, the Roam system *dynamically instantiates* the cell phone implementation.
- *Offloading computation:* The Roam system allows applications to make use of distributed computing resources to run software components that are beyond the target device capabilities. These components may require a more capable Java VM, or have certain CPU, memory, or network requirements, such that it is necessary to dispatch them to remote servers that can better support these requirements. Consider an application that has a component implemented to run only on the J2SE VM. When this application is started on or migrated to a cell phone, the Roam system finds a remote server capable of running the J2SE VM and offloads that software component to that remote server.
- *Transformation:* The Roam system allows device-independent components to be transformed at runtime to fit the target device capabilities. It provides a device-independent GUI toolkit that a developer can use to build user interface (UI) components. At migration time or runtime, these device-independent UI components are transformed to run on the target device.

The Roam system is a framework to assist developers in building applications that can adapt to heterogeneous devices. At design time, a developer must make the decision on how to partition an application into separate components, and for each component, whether to provide *multiple device-dependent* implementations (M-DD), a *single device-dependent* implementation (S-DD), or a *single device-independent* presentation (S-DI). At runtime or migration time, the Roam system applies the most appropriate adaptation strategy, shown in Table 1, based on the developer's design decision and the target device capabilities.

We believe that some adaptation strategies are better suited for certain types of application components. For some UI components, we believe that transformation adaptation may be the most appropriate, given that most UI components cannot be offloaded to a remote server for execution. However, device-independent UI authoring sometimes does not result in high-quality interfaces. In these cases, M-DD UI components are developed, and dynamic instantiation adaptation is preferred. This decision is up to the UI developer. For application logic components, we believe that S-DD implementation, using offloading computation adaptation is often most suitable, because most application logic components can be offloaded to remote servers for execution.

In order to suspend and resume the execution of mobile applications across heterogeneous devices, the Roam system must be able to capture execution state on the source device, serialize and transfer it to the target device, and then restore it. Because M-DD components may be dynamically instantiated, and the implementation of each is different, execution state must be captured using a *device-independent state representation*,

Table 1
Summary of adaptation strategies

Adaptation strategies	Application components	Intended for
Dynamic instantiation	Multiple device-dependent implementations (M-DD)	Customized, high-quality UI components
Offloading computation	Single device-dependent implementation (S-DD)	Application logic components
Transformation	Single device-independent presentation (S-DI)	UI components

and converted into representations specific to each component implementation using *execution state transformation*. The Roam system requires developers to provide execution state transformation logic for M-DD components. There is no need for a developer to provide execution state transformations for S-DI or S-DD components that have a single implementation. When Roam system applies transformation adaptation to an S-DI component, it automatically generates the corresponding execution state transformation.

Another important requirement is *Security*. Proper security policy must be enforced to ensure that only authorized applications are allowed to migrate from one device to another.

1.3. Organization

We organize the remainder of the paper as follows: Section 2 presents the design of the Roam system, Section 3 describes its implementation, APIs and a sample application, Section 4 presents related work, and Section 5 draws conclusions.

2. Design

The Roam system architecture is shown in Fig. 2. A *Roamlet* is an application that runs in the Roam system. A Roamlet can migrate between any two connected devices that have the Roam system running. The architecture contains three main components: *Roam Agent*, *Roamlet*, and *HTTP Server*. Roam agents must be installed and executed on both the source and target devices before a Roamlet can run, and before any Roamlet migration can occur. The flow for a Roamlet migration from a PC device to a PDA device is described as follows:

1. The Roam agent on the source device first negotiates with the Roam agent on the target device. The negotiation involves exchanges of the target device capabilities, the device capabilities needed by each application component, and the code base URL where the Roamlet component byte code can be downloaded from. Based on the exchanged information, the Roam agent decides the appropriate adaptation strategy for each component.
2. The Roam agent on the target device downloads the necessary Roamlet class byte code from the HTTP server for all application components that will be instantiated on the target device.
3. The Roamlet on the source device serializes its execution state and sends it to the Roam agent on the target device. The Roam agent may perform execution state transformation if an application component is transformed or dynamically instantiated.
4. The Roam agent instantiates the Roamlet on the target device.

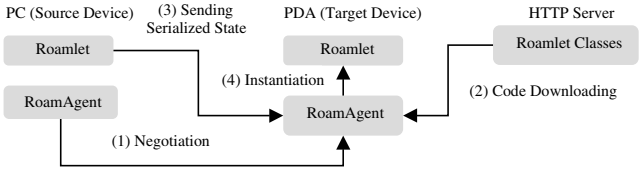

Fig. 2. Roam system architecture.

Roamlets are based on a *component-based programming model:* they are built as components that can be distributed across multiple devices, and can be composed together into a working application through well-defined interfaces. Fig. 3 shows an example of a Roamlet partitioned into three components: a GUI component with device-independent representation (S-DI), a second GUI component with two device-dependent implementations on PC and PDA (M-DD), and an application logic component with a PC implementation (S-DD). The component-based programming model allows the Roam system to apply a different adaptation strategy on different application component, such as offloading an S-DD component to a remote device, dynamically instantiating an M-DD component of a different implementation, and transforming an S-DI component without affecting other components in a Roamlet. This requires each component to invoke procedure calls of any other components regardless of whether they are running on remote or local hosts. In our current Roam system, the component-based programming model is built on top of Java Remote Method Invocation (RMI).

Roamlets and Roam agents use *resource specifications* to describe the device requirements for each component implementation, and the capabilities of a host device, respectively. At migration or load time, the Roam agent can compare the device requirements from the application components with the target device capabilities and decide the best adaptation strategy for each application component. Note that components in a Roamlet may have different device requirements, so a Roamlet is required to provide a separate set of device requirements for each component implementation. The current Roam system supports specification of a limited set of device capabilities, including Java VM configuration or profile, display size and input method, as shown in Table 2. For example, device capabilities for a PC are {J2SE VM, 1024×780 pixels, keyboard and mouse}. The device requirements for the PDA implementation of the GUI component shown in Fig. 3 can be {Personal Java VM, 320×240 pixels, stylus}, and the PC implementation of the GUI component can be {J2SE VM, 1024×780 pixels, keyboard and mouse}.

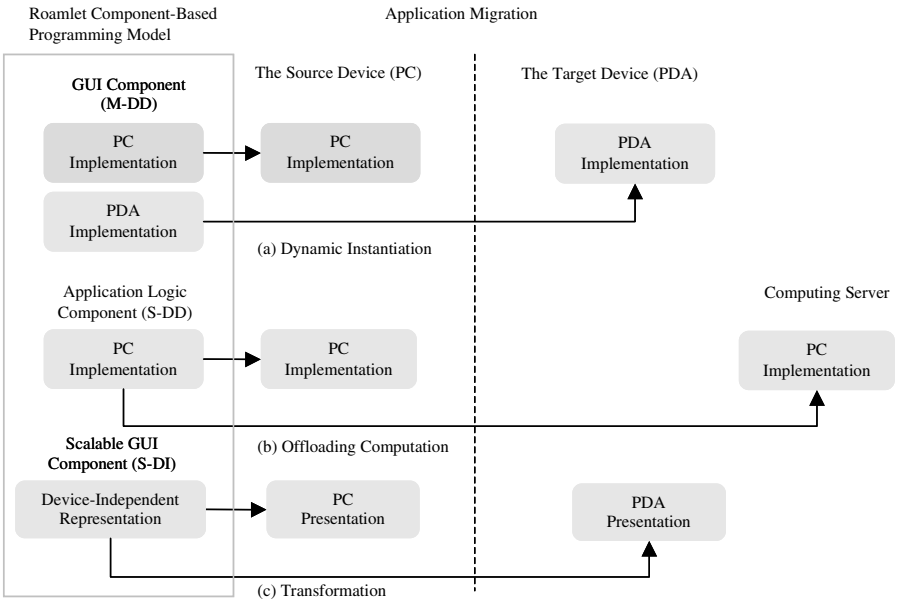

Fig. 3. Roamlet component-based programming model and different adaptation strategies for Roamlet components.

Table 2
Specification for device capabilities

	Types of Java VM	Display sizes	Input methods
PC device capability	J2SE	1024×780	Mouse and keyboard
Pocket PC device capability	PersonalJava	320×240	Stylus and virtual keyboard
Cell phone device capability	J2ME/DoJa profile	100×100	Keypad

Since the scalable GUI (SGUI) component has a device-independent representation, its device requirements can contain a range of capabilities.

2.1. Dynamic instantiation

In dynamic instantiation, a developer provides M-DD implementations for a Roamlet component. The rule for selecting the most suitable device-dependent implementation is that the component implementation that best matches (but not exceeds) the target device capability is selected. An example of a Roamlet migrating from a PC to a PDA is shown in Fig. 3(a). The Roam agent finds that the PDA implementation of the second GUI component exactly matches the target device (PDA) capability and instantiates it.

Using dynamic instantiation, Roamlet developers can program different Roamlet behaviors in separate code segments. At migration time the Roam system loads only the code segment that exhibits a behavior suited for the target device capability. Note that dynamic instantiation is different from programming different behaviors in the same code segment, which is not always desirable in Java. For this approach to work, the Roamlet has to be programmed to run on the least capable device (the *lowest-common denominator* approach) because the code segment that runs on different VM configurations and devices can only assume the most primitive API libraries provided by the least capable VM and devices. This is obviously undesirable for application programmers who would like to take advantage of rich class libraries provided by more capable devices.

The Connect4 Roamlet is a multi-platform game that we implemented to illustrate dynamic instantiation adaptation. Its GUI component provides M-DD presentations, one matching the PDA device capability and one matching the PC device capability. When Connect4 is instantiated on the PC, its PC presentation is selected for instantiation (Fig. 4(a)). When the user later migrates the Connect4 game to a PDA, its PDA presentation is selected for instantiation (Fig. 4(b)). The PDA presentation is a scaled down version of the PC presentation with some features and buttons removed to fit the smaller PDA display size (such as "undo" and "new game").

Dynamic instantiation creates a challenging situation, where a Roamlet is instantiated with a different implementation on the source device from that on the target device, as shown in Fig. 3(a). The execution state of the PC GUI component on the source device does not correspond to the PDA GUI component instantiated on

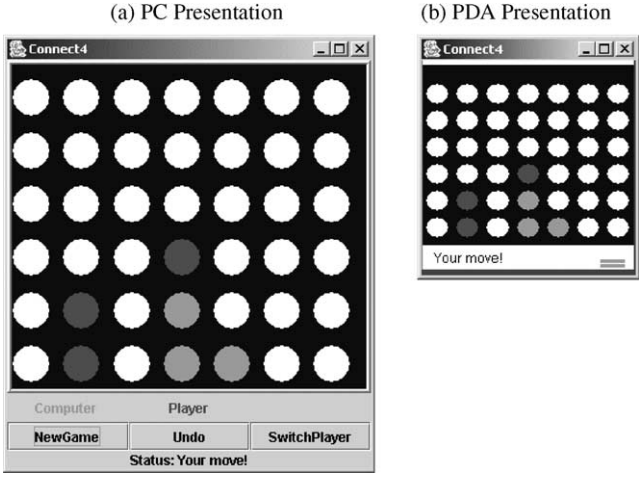

Fig. 4. Screenshots of M-DD presentations of Connect4 Roamlet: one presentation for PC and one presentation for PDA.

the target device. For example, the PC GUI component might use radio buttons to present a choice, while the PDA GUI component might use a drop-down list instead. In order to migrate the execution state of the PC GUI component to the PDA GUI component, execution state must be transformed before it can be restored on the target device (e.g., mapping the value of the radio button to the drop-down list). This state transformation logic is provided by developers.

2.2. Offloading computation

Offloading computation allows a Roamlet to delegate the execution of computation- and memory-intensive components to any remote server that has the device capabilities to run them. Offloading computation is intended for an application component that has a S-DD implementation. Fig. 3(b) shows offloading computation for an S-DD application logic component. After the Roam agent finds that the application logic component requires a PC-equivalent device, which exceeds the PDA host's capabilities, it attempts to find a computing server (PC) where it can offload this application logic component. Both the Roamlet users and the Roam agent can specify a list of servers that can accept offloadable components. When a server is found, the Roam agent directs the application logic component to be migrated to that server. A Roam agent can apply offloading computation to an M-DD component if none of its device-dependent implementations can run on the target device. For example, if an M-DD component has two implementations on PC and PDA and the target device is a cell phone, dynamic instantiation will fail. In this case, the Roam agent will find the most capable server on the list, and offload an appropriate M-DD component to it. The precedence rule for applying different adaptation strategies is that a Roam agent first applies dynamic instantiation adaptation to an application component. If the dynamic instantiation adaptation fails, it will apply offloading computation adaptation.

The Connect4 game in Fig. 4 contains a S-DD application logic component that is offloadable. The application logic component keeps track of the state of the game and implements an artificial intelligence (AI) engine that computes the computer's next move. Since the AI could be computationally- and memory-intensive, it requires a device with a fast processor and sufficient memory. When the Connect4 game is running on the PC, the application logic component is instantiated locally. When the Connect4 game is migrated to the PDA, the application logic component is offloaded to a computing server.

Not all components in a Roamlet are offloadable. The Roam system allows a Roamlet to specify its components as either non-offloadable or offloadable components. Non-offloadable components are required to run on the target device. For example, GUI components and security-sensitive components are rarely offloaded. If the target device does not have the capability to run the non-offloadable components, the application migration simply fails and an error is presented to the user. A Roamlet can also specify components as *reverse-offloadable* or *non-reverse-offloadable*. Consider the same example in Fig. 3(b). At a later time, the user wants to migrate the Roamlet back to the PC from the PDA. If the application logic component is reverse-offloadable, it will be pulled back from the computing server to the target PC device; otherwise, the application logic component will stay on the computing server. It is a trade-off between paying a one-time transfer cost of migrating the application logic component (in the case of reverse-offloadable) vs. the continuous cost of communicating with that remote component in a Roamlet (in the case of non-reverse-offloadable).

2.3. Transformation

Transformation is a runtime UI generation process from a device-independent representation constructed by developers at design time to device-dependent presentations. Fig. 3(c) shows transformation of a device-independent UI component into a PDA presentation when the Roamlet migrates to a PDA. To build a device-independent representation and to generate device-dependent presentations, we provide a device-independent GUI library and a runtime transformation tool, which we called the SGUI toolkit.

The design of the SGUI toolkit is shown in Fig. 5. It contains three modules: *SGUI library*, *transformation manager*, and *render manager*. The process flow goes as follows. At design time, UI designers use widgets provided by the SGUI library to construct device independent UI presentations. At runtime, the

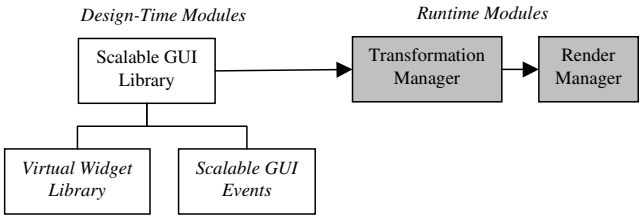

Fig. 5. SGUI toolkit.

transformation manager transforms a device-independent representation into a device-dependent presentation that satisfies the target device constraints (display size and the available input methods). The render manager displays the device- presentation on the target device.

2.3.1. Shop hunter application

"Shop hunter" is a hypothetical multi-platform application that helps a user to buy items anytime, anywhere. We will use the shop hunter application to illustrate transformation adaptation supported by the SGUI toolkit. The shop hunter application allows a user to input new shopping items, to view a list of shopping items, and to search for directions to nearby stores with the cheapest prices. We have prototyped a device-independent representation of the shop hunter UI using our SGUI toolkit, and applied transformation to generate device-specific presentations for notebook PCs, PDAs, and cell phones which are shown in Fig. 6.

The first noticeable difference between different device-specific presentations is how the device's displayable screen size changes the *layout* and *pagination*. The notebook PC presentation fits on one page, the PDA presentation is laid out on three different pages, and the cell phone presentation is broken up into six pages. The second noticeable difference between the screenshots is that the "sort" UI components are different between the notebook/Pocket PC presentation and the cell phone presentation. The notebook and Pocket PC presentations use a radio button group and the cell phone presentation uses a more compact drop-down list. The third noticeable difference is that the UI components corresponding to "add shopping item" is missing on the cell phone presentation, because this task involves text entries and is therefore inconvenient on a cell phone.

2.3.2. SGUI library

The SGUI library contains two modules: the *virtual widget library* and the *SGUI events*. The virtual widget

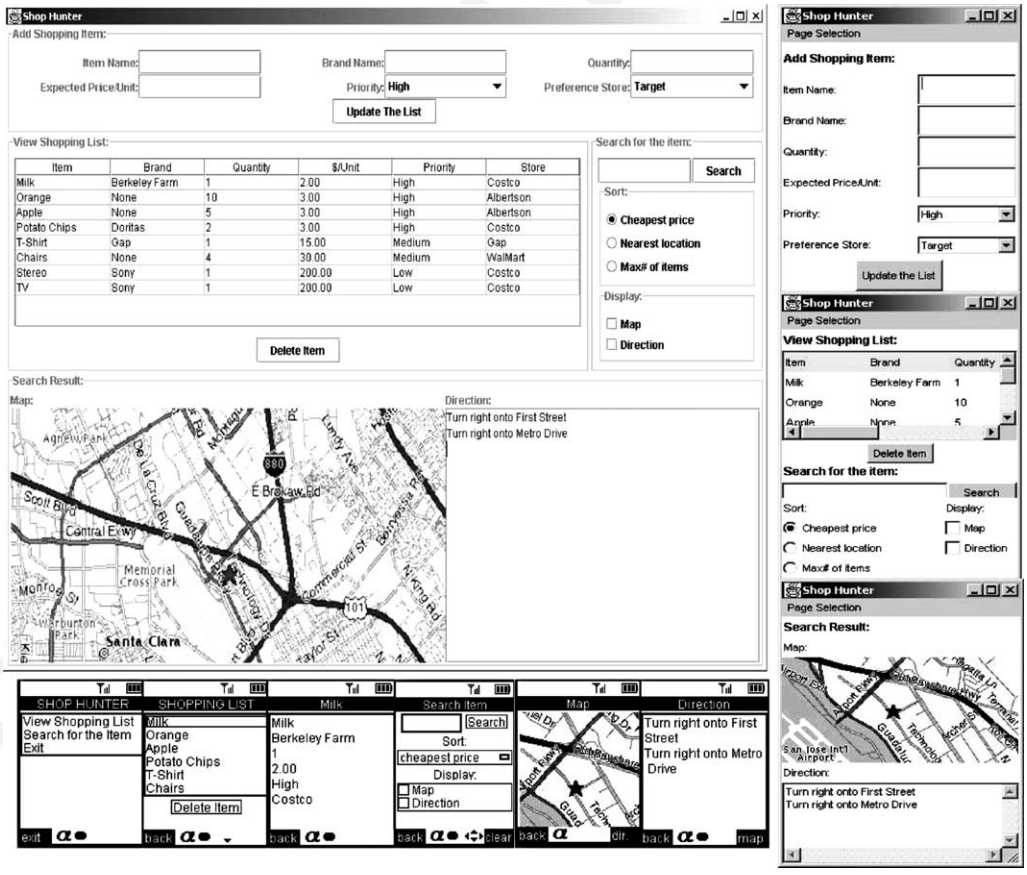

Fig. 6. Shop hunter application.

library contains a comprehensive set of UI widgets similar to Java's Swing library. It includes widgets such as labels, buttons, menus, text fields, scroll bars, tables, etc. The virtual widget library is implemented for each Roam-supported device platform. Since the virtual widget library is based on Swing, its APIs look very similar to Swing APIs.

The SGUI events are abstractions of user interaction events such as mouse, keypad, or stylus input. Since different device platforms support different input methods, the SGUI library describes mappings between device-dependent events and abstract device-independent events. This abstraction allows UI developers to handle these events regardless of which platform a UI is running on. For example, an abstract *action* event associated with a button press can be generated from a mouse click on a PC, a tap from stylus on a Pocket PC, or a select-key press on a cell phone. The event is realized as a device-dependent event at runtime, and then delivered to the application.

For some applications, mapping between device-dependent GUI events to device-independent events may not always be possible due to platform constraints. For example, an application may contain an interactive map that can zoom in or out when a user clicks on a specific point in the map image. On a PC, a mouse event can be used to capture the specific location of the mouse click. However, a cell phone lacks a pointing input method; as a result, it is not possible to map an equivalent event on a cell phone. To address the need for device-dependent events, the SGUI library also retains a set of events for device-dependent input methods. This allows a scalable application to enhance its UI on a particular device platform. Since device-dependent events are generated only on some specific device platforms, they should not be used to implement any core features of UI that are expected to be available on any devices.

2.3.3. Device-independent representation

UI designers specify two kinds of information in the device-independent representation—*task model* and layout. Fig. 7 depicts the top portion of the device-independent representation for the shop hunter application. The device-independent representation has a tree-like structure. The child nodes of the root node are task nodes representing different tasks. For the shop hunter application, there are three tasks: add shopping item, view items, and search item. A task node can have subtask nodes, and so forth. The add shopping item task contains a scalable button called update list, and logical panel A. The button is a scalable UI component instantiated from the virtual widget library. A logical panel is a presentation unit that groups related child logical panels and scalable UI widgets. A spatial layout can be specified on the relative positions of widgets and panels contained in logical panels. For the add shopping item task node, UI designers would specify that logical panel A be placed at the top, and update list button at the bottom. Logical panel A contains six smaller logical panels A1 through A6 arranged in a 2×3 grid.

UI designers can specify *task preferences* on task nodes. The task preference tells which platforms with which input methods are suitable to display this task. In the shop hunter application, a UI designer would specify the Add Shopping Item task to be suitable for displaying on notebook PCs or PDAs with virtual keyboards, but not for cell phones. As a result, UI components corresponding to Add Shopping Item task will not be displayed on cell phones as shown in Fig. 6.

In addition to layout, UI designers can specify other layout properties on logical panels, such as *layout pri-*

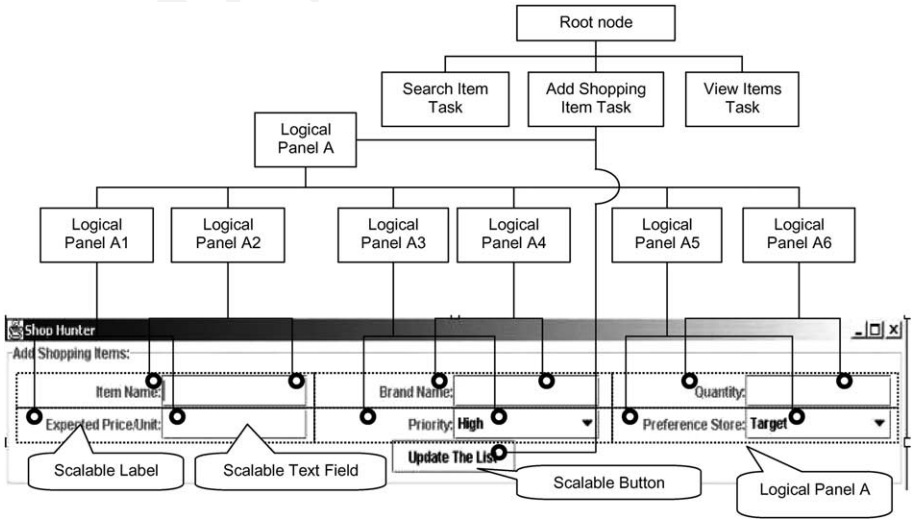

Fig. 7. Device-independent representation for the top portion of shop hunter application.

ority, *split-ability* and *title*. Layout priority decides the order in which logical panels will be placed on pages. Split-ability decides if descendant nodes of a logical panel can be placed on separate pages. By default, all logical panels are not splittable. Titles of logical panels are used to create a navigation menu bar for moving between pages. For example, in the Pocket PC presentation of the shop hunter application of Fig. 6, the navigation menu bar Page Selection contains the titles of logical panels as menus.

Integration of the layout structure and the task model in the device-independent representation tree poses a limitation that the task model hierarchy must not conflict with the layout structure hierarchy. It is not possible to specify a task that places its widgets in the same logical panel as widgets from another task. For example, the view items task cannot place a widget under Logical Panel A1 because it is also under Add Shopping Item task. To address this limitation, we are looking into separating the task model from the layout structure into two presentation trees similar to Eisenstein et al. (2001).

Note that UI designers must construct device-independent representations for both static and dynamic content. In the following sections, we explain how the transformation manager transforms a device-independent representation into a device-specific presentation.

2.3.4. Transformation manager

The design of the transformation manager is shown in Fig. 8. It contains three sub-modules: *task manager*, *layout algorithm* and *widget transformation*. The first step in transformation manager involves the task manager. Its job is to choose the appropriate tasks for the target device platform, and to remove widgets belonging to tasks not appropriate for the target device platform. For example, if UI designers specify that the Add Shopping Item task in the shop hunter application is not suitable for cell phones; widgets corresponding to the Add Shopping Item task are removed by the task manager and not displayed on the cell phone presentation.

The second step in the transformation manager involves the layout algorithm working with widget transformation. Its job is to paginate the presentation into separate pages according to the layout specified by UI designers and also to meet the constraint that a page cannot exceed the target device's displayable screen size.

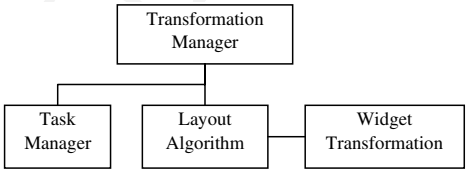

Fig. 8. Transformation manager design.

2.3.5. Layout algorithm

We define two requirements for our layout algorithm. First, the generated presentation should have reasonably high quality without requiring significant device-specific customization from UI developers. One implication of high quality and the display size constraint is that the generated presentation must fit the target display without scrolling, because scrolling generally degrades usability. Second, the algorithm must be fast, so that it does not cause significant presentation generation delay. There have been many methods on how to layout widgets for different display sizes. Some of these methods, such as the one proposed by Masui (1994), are highly computationally complex. Some methods require too much information from developers: for example, Humanoid (Szekely et al., 1992) asks developers many layout-related questions before it can generate a final presentation. Among all proposed methods, TEX (Linton, 1989) is the most promising method in formatting two dimensional box-like GUI widgets (Masui, 1994; Olsen et al., 2000). TEX allows each widget to report its desired size for positioning.

Fortunately, Java has default layout managers that are similar to TEX. Java also allows UI developers to specify the desired location of each widget through a set of predefined layout constraints. In order to meet our first requirement, we propose to have only one set of layout specification from UI developer, and this set of layout specification is used to generate layouts for other platforms. The specification is the same as the Grid Bag Layout Constraint in Java. We recommend UI developers to specify the layout according to the presentation on the largest device display, such as a presentation on a PC, for reasons detailed in Section 3.6. The layout algorithm will try to follow that specification as closely as possible when it is creating layouts for other platforms with smaller display sizes.

The general strategy for the layout algorithm is to start from the lowest nodes in the device-independent representation tree, and then work its way up to the root node. When it encounters a non-splittable logical panel that cannot fit the display size using the specified grid bag layout, the layout algorithm applies *flow layout*. We choose flow layout because it is simple, and it involves minimum computation while maintaining a reasonable presentation (Vanderdonckt, 1995). Since using flow layout may not generate a high quality layout, it is *only* applied when the specified grid bag layout fails. If flow layout also fails on a non-splittable logical panel, the layout algorithm may apply widget transformation to find smaller alternative presentations. The detailed steps of the layout algorithm are described as follows:

1. The algorithm starts by finding the *lowest-level unsplittable node* with the highest layout priority in the presentation tree as shown in Fig. 9. A lowest-level

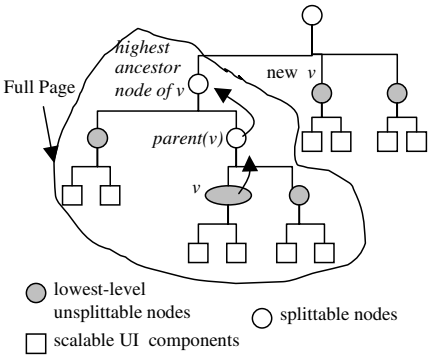

Fig. 9. Layout algorithm.

unsplittable node is defined as a node whose child nodes are all widgets (or leaf nodes), and the widgets have to be placed on the same page. The goal is to try to place widgets that are assigned higher layout priorities to the first few pages, so that they are easier for users to locate. Denote v as the chosen node.

2. A default *style* corresponding to the target device platform is applied to the set of widgets under node v. The style guide sets common properties of widgets such as font size, spacing between components, etc. The purpose of the style is to have a consistent look across different widgets. The set of widgets under node v are laid out on a page according to the grid bag layout constraints specified by the UI developer. The precise size of the page is then calculated.
3. If the size of page is bigger than the device display size, the page is *over-filled*. We apply flow layout on child nodes under v, and re-compute the page size. If the page is still over-filled, we check if the node is splittable or not. If the node is splittable, we allocate a new page to place extra widgets that cannot fit on the previous page. If the node is non-splittable, widget transformation is applied to as many child nodes under v as needed such that the flow layout can fit them on one page. If widget transformation also fails, the layout algorithm will leave some widgets out of the page and notify its users. Details on widget transformation are discussed in Section 2.3.7.
4. If the size of the page is smaller than the size of the device screen size, the page is *under-filled*. This means that the algorithm can accommodate additional widgets from sibling nodes of node v. The algorithm proceeds to find the *highest ancestor node* of node v in the presentation tree as shown in Fig. 9, such that using that ancestor node can pack the maximum number of widgets in a page without violating the display size. Denote the parent node of node v as $PARENT(v)$. The algorithm tries to place the set of widgets under PARENT(v) on a page. This is done by repeating step 2, i.e., applying the current style and the grid bag layout specification to the widgets under PARENT(v). If the resulting page is still under-filled, it tries $PARENT(PARENT(v))$, and so forth until the resulting page becomes full.
5. If the page is *full*, a new page is allocated for placing remaining widgets. The algorithm first finds another lowest-level unsplittable node with the next highest task priority from the presentation tree that hasn't been placed on any pages. This node becomes the *new node* v, and the algorithm goes to step 2.
6. When all nodes are processed—that is, placed on a page—the algorithm terminates.

For the shop hunter application in Fig. 6, the table showing the shopping list has different device-specific presentations on Pocket PCs and cell phones. For the cell phone presentation, the table is transformed into a selection list using item names as indices. The map image is scaled down on the Pocket PC presentation and on the cell phone presentation. Techniques involving scaling images and graphic objects (W3C, 2000) are outside the scope of this paper.

2.3.6. Page navigation

While the layout algorithm is automatically generating a new layout, it is also constructing a simple navigation menu bar for moving between pages. If the application already has a menu bar, the algorithm will simply add a customizable menu named "Page Selection" by default as the first menu from the left on the menu bar. If the application does not have any menu bar, a new one is created. On the Pocket PC, the menu bar always appears at the top of each page under the Page Selection menu bar as shown in Fig. 6. On the cell phone, the first page is always the top-level task navigation menu. For each generated page, a menu item is created and it will be automatically added as the item of the "Page Selection Menu." The name for each menu item is from the title property of the occupying logical panel.

2.3.7. Widget transformation

Widget transformation is the transformation from one widget, either primitive or composite, to another that consumes less space in the graphical layout. Transformation is guided by *transformation rules*, similar to rules in a rule-based system. Widget transformation rules are selectively applied to yield a presentation which is optimally usable given display size constraints. For example, Fig. 10 shows the effect of a transformation rule mapping a large, composite widget containing pairs of property labels and value textfields, into a smaller composite widget containing a property list box and a single value textfield. The transformed composite widget is smaller and remains easy to use.

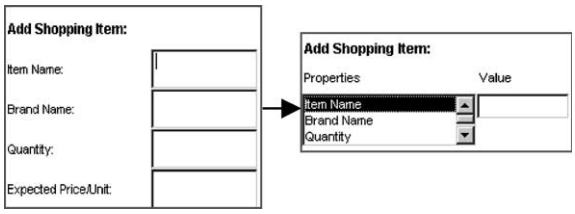

Fig. 10. A multiple-to-multiple transformation.

At the core of widget transformation is an algorithm to determine which widgets to transform, and which transformation rules to apply to them.

To determine the widgets that are more acceptable to transform, developers specify which widgets are *core*—representing important application functionality, and which are *optional*—less important for use of the application. These hints are provided for primitive widgets, as well as those that are combined into composite widgets. We prefer to transform optional widgets rather than core widgets whenever possible, so that we degrade usability as little as possible (Vanderdonckt, 1995); only when there are no optional widgets, or when the layout size reduction is insufficient to meet display size constraints, are core widgets transformed.

Determining the transformation rules to apply to the widgets is more complex. There are four types of transformation rule:

- *One-to-one:* transforms from a single primitive widget to another single widget, e.g., a rule transforming a list into a drop-down box.
- *One-to-multiple:* transforms from a single widget into multiple widgets, e.g., a rule transforming a single table into multiple lists or drop-down boxes.
- *Multiple-to-one:* transforms from multiple widgets of the same type into a single widget, e.g., a rule transforming a group of radio buttons or checkboxes into a single drop-down list box, and a rule transforming a group of one-line textfields into a single multi-line textarea.
- *Multiple-to-multiple:* transforms from multiple widgets of different types into other multiple widgets, e.g., a rule transforming a group of label and textfield pairs into a drop-down list box and a single textfield, as shown in Fig. 10. Note that a multiple-to-multiple rule is composed of a set of multiple-to-one rules, and a set of classic relationships described in Vanderdonckt (1995).

Each of these classes of transformation rule has a priority assigned to it, corresponding to the demonstrated usability of widgets transformed by rules in the class. That is, we prefer transformation rules that tend to result in easier-to-use widgets over transformation rules that tend to result in harder-to-use widgets. One-to-one rules are preferred first, then one-to-multiple, multiple-to-one, and finally multiple-to-multiple. Note that multiple-to-multiple rules can change the UI drastically, and degrade usability, compared to one-to-one rules. Conversely, note that one-to-one rules tend to offer less space savings than multiple-to-multiple rules. Our prioritization balances this tradeoff: we want as large—and consequently as usable—a UI as possible given display size constraints, but we accept smaller and less usable UIs as necessary.

We limit the set of transformation rules that may be applied by eliminating rules whose original widgets require input methods that are unsupported by the transformed widgets on the target device. For example, if the original UI contains a J2ME DoJa button that has an action triggered on a mouse-in event, any rules transforming that button to a softkey are not applicable, because softkeys do not support any equivalent to mouse-in events. The current SGUI toolkit provides a set of commonly used transformation rules. We also allow developers to add their own transformation rules.

2.3.8. SGUI state and event transformation

When an application is migrated from a device of one platform to another device of a different platform, the presentation on the target device may use a different set of widgets from the source device, so there is a need to map running state between the source presentation and the target presentation. This is done by state synchronization and restoration between the running state of a device-specific widget and its corresponding device-independent widget. These two mechanisms are supported in the SGUI library to ensure that the device-independent representation always has the most recent widget state right before and right after any migration. Prior to any application migration, the running state of a device-specific widget is synchronized with its corresponding device-independent widget. After an application migration, the running state of the device-independent widget is restored to the device-specific widget.

Event transformation is also needed to support the same UI interaction across different device-dependent presentations. This is done by a bi-directional mapping mechanism between device-dependent events generated from device-dependent widgets and their corresponding device-independent events generated by a device-independent widget. This mapping mechanism is supported in our SGUI library.

2.4. Security

We assume that Roam agents have been installed securely and they are trusted entities. We also assume that a user would only migrate applications among his/her personal devices (not public and not shared) such as cell phones, PDAs, and home PCs. This means that a

```
public abstract class Roamlet implements Serializable {
        public Roamlet(Object[] args);
        public synchronized final void migrate(String hostname);
        public synchronized boolean onInitialization();
        public synchronized boolean onRemoval();
        public synchronized boolean onArrival();
        public synchronized boolean exit();
        static final void instantiate(String className, Object[] initArgs, String codebase);
};
```

Fig. 11. Roamlet runtime migration API.

user should know when to expect an application migration on which device and from which device.

Based on these assumptions, we have applied an authenticated encryption scheme (Rogaway et al., 2001) that can provide both *privacy* and *authentication*. Privacy means that application execution state is encrypted so that only the intended recipient with the correct password can decrypt it. Authentication means that the intended recipient can distinguish a legitimate application migration sent by its user from a malicious application migration sent by other users. Roam's security works as follows. At the start of the application migration, the Roam agent running on the source device prompts the user for a password. This password is a one-time password that is valid only for one application migration. At the same time, it will also generate a one-time number called a *nonce*. The only requirement for the nonce is that it is a new number for each application migration. The nonce can be generated from a random number generator or a simple counter that increments each time it is used. Based on the one-time password and nonce, the application execution state is encrypted as ciphertext. The ciphertext is sent together with the cleartext nonce to the target device. On the target device, the Roam agent asks the user for the same password that was used to encrypt the ciphertext on the source device. Based on the password and the nonce, the Roam agent can decrypt the ciphertext to get the application execution state. If the password used in decryption does not match the password used in encryption, the authenticated encryption algorithm returns "INVALID" to the Roam agent and the migration request is denied.

3. Implementation

We have implemented the Roam system using the Java language and the Java language features: RMI, Serialization, and Reflection. The Roam system currently runs on the PCs with JVM, and on Pocket PC (PDA) devices with Personal Java VM. In the future, we are planning to port the Roam system to other JVMs with RMI and Serialization. The total code size of the Roam system is approximately 15,000 lines, with 8500 lines in the SGUI toolkit.

The Roam system provides a set of Roamlet APIs. The core class in the Roamlet APIs is called Roamlet, which all Roamlets must extend. The Roamlet APIs allow a Roamlet to migrate at runtime, to be instantiated dynamically, to offload computation, or to be transformed. We describe the essential Roam APIs here.

3.1. Roamlet runtime migration API

The runtime migration API is encapsulated in a class called Roamlet shown in Fig. 11. All Roamlets must extend the class Roamlet. A Roamlet calls the migrate( ) method to be dispatched to a target device specified by the hostname parameter. The Roam agent calls the Roamlet's onInitialization( ) method when it is first instantiated on a device. The Roam agent calls the Roamlet's onRemoval( ) method before it is removed from the source device, and after it migrates successfully to a target device. The Roam agent calls the Roamlet's onArrival( ) method when it arrives at the target device. A Roamlet calls the exit( ) method to remove itself from the Roam system. The instantiate( ) method is called to instantiate a Roamlet on a local host. After a Roamlet is instantiated on a device, the Roam agent creates a separate thread to run it.

3.2. Roamlet component description API

A Roamlet can describe its components using the API shown in Fig. 12. A Roamlet calls addComponentDesc( ) to add a component descriptor, RoamletComponentDesc, for each of its components. RoamletComponentDesc describes the type of component (type, which can be M-DD, S-DD, or S-DI), the class name of this component, the class name of the component to be instantiated (classname), and a list of possible implementations for this component. Each implementation of a component is described in a separate ImplDesc, which contains the class name (classname) of the implementation and the device requirement for this implementation (req).

```
public abstract class Roamlet implements Serializable {
        ... // from Figure 11
        public synchronized boolean addComponentDesc(RoamletComponentDesc desc);
};

public class RoamletComponentDesc {
        public RoamletComponentDesc(String classname, ComponentType type, ImplDesc implDesc[]);
};

public class ImplDesc {
        public ImplDesc(String classname, DeviceRequirement req);
};
```

Fig. 12. Roamlet component description API.

3.3. SGUI API

The SGUI library provides four packages (sgui, sgui.event, sgui.dd.event, and sgui.ir) for UI developers. The sgui package provides virtual widgets similar to widgets available in the Swing library with prefix S, such as SButton, SCheckbox, STextfield, SLabel, etc. The sgui.event package provides device-independent events with S prefix, such as SActionEvent. The sgui.dd.event package provides device-specific events, such as SMouseEvent. UI developers follow the Java 2 event model to add SGUI event listeners and specify actions upon the reception of SGUI events.

The sgui.ir package provides APIs for constructing a device-independent representation. One of the core classes is the LogicalPanelNode shown in Fig. 13. It has an add( ) method to add virtual widget and logical panels as its child nodes. A LogicalPanelNode can specify a list of properties, such as layout constraint, task preference, layout priority, whether it can be paginated across multiple pages, and whether it represents core or optional features.

3.4. Chess Roamlet

We have taken a Java-based Chess game that is freely available on the Internet and rewritten it as a Roamlet. The Chess game has two modes: multiplayer mode where two users play against each other, and computer mode where a user plays against AI. The mobile environment is setup as in Fig. 14: a Pocket PC device running Personal Java VM, a notebook PC running a standard J2SE VM, and they are both connected by an 802.11 wireless LAN. The user starts the Chess game in computer mode on the notebook PC and then migrates it to the Pocket PC. There is a third device, a desktop PC, acting as the remote server where the AI component can be offloaded to.

The Chess Roamlet contains the following components:

- An application logic component is implemented as an offloadable device-dependent component (S-DD). It runs an AI engine that calculates the computer's next move by building a search tree, which is both computationally and memory-intensive. It has a device requirement of a PC.
- A GUI component is implemented as a non-offloadable device-independent component (S-DI) using the SGUI library. It draws a player board that holds both a chessboard and chat/message boxes where two players can send messages to each other.
- A second GUI component is implemented as a non-offloadable device-dependent component with two implementations (M-DD). These two implementations have two different sets of images showing chess pieces, the set for the PC presentation has larger images than that of the Pocket PC presentation.

The Chess game in computer mode with PC presentation is shown on the left side of Fig. 15. When the user migrates the Chess game to the Pocket PC, the first (S-

```
- sgui.SComponent
        |
        + - (device-independent widgets: SButton, SCheckbox, STextfield, SLabel, SPanel, etc)

public class LogicalPanelNode extends Node {
        public LogicalPanelNode(NodeGridBagLayoutConstraint layoutConstraint,
                                boolean[] taskPreference,
                                int layoutPriority,
                                boolean splittability,
                                boolean coreFeature);
        public add(Node node);
}
```

Fig. 13. Sample APIs for constructing device-independent representation.

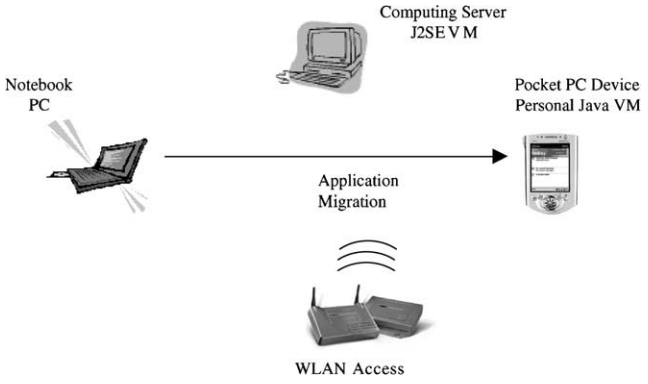

Fig. 14. Experimental setup of Chess Roamlet.

DI) GUI component is transformed into the Pocket PC presentation shown on the right side of Fig. 15. Since the player board is too large for the Pocket PC's display, SGUI layout algorithm paginates it into two pages. The first page shows the chessboard, and the second page shows chat and message boxes. A page navigation menu is added to the right of the menu bar for users to switch between two pages. The M-DD GUI component is instantiated with the Pocket PC presentation and it loads the smaller set of images for chess pieces. The application logic component is offloaded to the computing server.

3.5. Experience on development effort and performance

The Chess game was originally implemented to run on the PC using Java AWT GUI library. We have asked third party developers, who are new to Roamlet programming, to rewrite the Chess game as a Roamlet using SGUI library. The experience from the third party developers have told us that most of the efforts in rewriting the Chess application fall into the following three categories: (1) modularizing an application into application logic components and GUI components, and converting them into Roamlet components, (2) porting GUI components to use the SGUI library, and (3) adding multiple device-specific GUI components for platforms that are not supported by the original Chess application. Their experience also told us that it is relatively straightforward to separate application logic components from UI components in a well-written Java application. The difficult part is to decide the type (S-DI, S-DD, or M-DD) for each component. Finally, their experience told us that, since our SGUI library follows Java Swing library, porting AWT components to use SGUI components is relatively straightforward.

The original Chess game has approximately 7900 lines of code. After the converting it into a Roamlet, it has approximately 9000 lines of code, which is about 14% increase in code size. Most of the additional code comes from code that converts Java objects into Roamlet components, and the second (M-DD) GUI component that supports the Pocket PC presentation. This shows the tradeoff between development cost and UI quality. That is device-dependent UIs in general have higher quality, but developers need to create M-DD presentations for different platforms, resulting in increased application code size. This tradeoff can also be found in the Connect4 game. When its UI component has two device-dependent implementations for PC and PDA, the code size is approximately 1200 lines. The code size is reduced to 1100 lines when its UI component is re-implemented as a device-independent component using SGUI library.

We have measured *migration latency* of application migration. Migration latency is defined as the elapsed time from the moment the user accepts the migration request on the target device, to the moment when the Roamlet is completely restored on the target device. We have measured latency for both Connect4 and Chess games migrating from a notebook PC (Intel Pentium III-800 CPU running Windows XP) to a Compaq iPAQ H3870 Pocket PC (StrongARM 206 MHZ CPU running

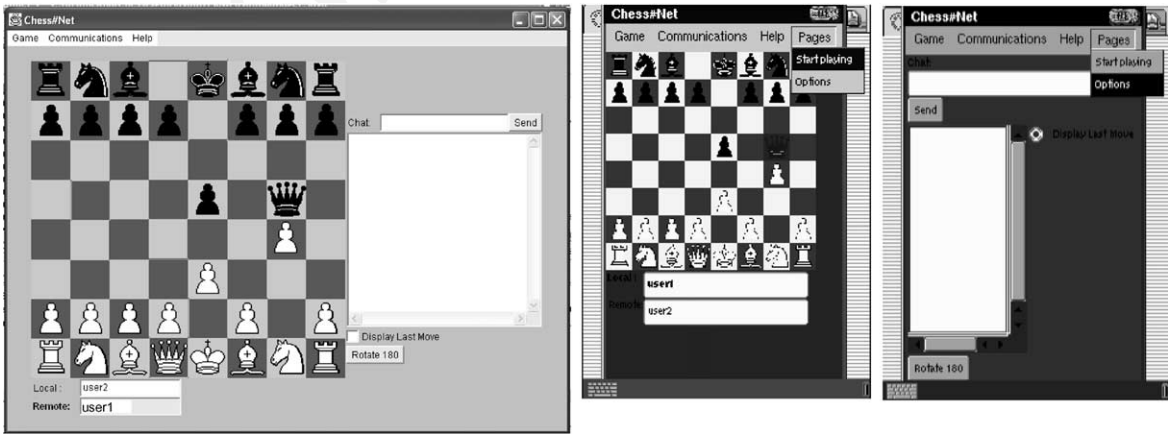

Fig. 15. Screenshots of Chess game on notebook PC and Pocket PC after migration.

Table 3
Migration latency measurements

Applications	Migration latency			
	Notebook PC to Pocket PC		Pocket PC to notebook PC	
	Context transfer time (s)	Transformation time (s)	Context transfer time (s)	Transformation time (s)
Chess	4.9	10.2	4.2	1.1
Connect4	2.8	6.4	1.2	1.4
Shop hunter	N/A	3.7	N/A	0.9

Linux) and vice versa. We divide the migration latency into two parts: (1) *context transfer time* is the time for saving the execution state on the source device, sending and restoring it on the target device, and (2) *GUI transformation time* is the time for transforming a device-independent GUI component into a target device-specific presentation. The measurements are shown in Table 3.

Measurements show that transformation time dominates the migration latency when migration is from a notebook PC to a Pocket PC, because the layout algorithm is computationally intensive and it runs the target device which is a slower Pocket PC. However, when migration is from a Pocket PC to a notebook PC, the context transfer time dominates the migration latency. This is due to serialization of the Roamlet execution time on a slow Pocket PC. For the shop hunter application, transformation is offloaded to a PC. Measurements show that the transformation manager takes less time to generate a presentation for a bigger display (0.9 s) than for a smaller display (3.7 s). The reason is that generating a presentation for a smaller display is likely to involve testing different paginations and transformation rules.

3.6. End-user usability

Our limited experience on the available Roamlet applications shows that SGUI toolkit can generate consistent presentations across different platforms. There are three aspects of consistencies: task consistency, layout consistency, and transformation consistency. Task consistency means that, unless the developers specify that some tasks are not appropriate on certain platforms, all tasks will be presented across all platforms. Layout consistency means that, if UI components are laid out next to each other on one platform, they are likely to be found adjacent to each other or on adjacent pages on other platforms. Transformation consistency means that users can expect similar transformations to be applied to same type of UI components across applications. We believe that these consistencies contribute to better *learnability* of applications.

Our simple page navigation menu works only for simple UIs but not complex UIs. We believe that we can improve the usability and efficiency of generated page navigation by incorporating device-specific navigation models provided by SGUI toolkit or explicitly specified by UI developers. To generate better page navigation, the navigation model should consider interactions and relation between tasks and device's input methods.

4. Related work

We divide the related work into two parts: mobile agent systems and multi-platform UIs.

4.1. Mobile agent systems

There has been an abundance of research work in the area of migrateable mobile agent systems. Aglet (Lange and Oshima, 1998) is one of the most well-known Java-based agent systems. It provides a Java development toolkit and libraries for building mobile agents that can move from one computer to another. Like Roamlet, Aglets do not require any modifications to the Java VM. An Aglet is shielded through an Aglet Proxy, which protects the Aglet from unauthorized access. Aglets run in an execution environment called AgletContext, which is like the Roam agent. However, Roamlets differ from Aglets in that Roamlets assume a heterogeneous device environment, whereas the Aglet assumes a homogeneous PC or PC-equivalent environment. Furthermore, the Roam system implements dynamic instantiation and computation apportioning to address the device heterogeneity problem. We have found similar Java-based agent systems in Jumping Beans (Aramira, 1999), MOA (Milojicicc et al., 1998), Concordia (Mitsubishi Electric ITA Horizon Systems Laboratory, 1998), Voyager (ObjectSpace, xxxx), Mole (Strasser et al., 1996) and Telescript (White et al., 1997). Voyager is a commercial product by ObjectSpace, and it incorporates features of mobile agents and mobile objects into the ORB and COBRA. MOA performs resource management on migrateable applications as to what system resource they can use on the target device, and it also supports migrateable communication channels (the external running state) so that running communication channels can migrate with the rest of the applications. There are also similar but non-Java-based agent systems, e.g., Agent-Tcl (Gray et al., 1997).

Fünfrocken (Fünfrocken, 1998) and Ara (Peine and Stolpmann, 1997) describe ways to achieve *strong migration* for Java applications. Strong migration means that the call stack, which is a part of the thread state that is not serializable, can also be migrated. Fünfrocken uses a preprocessor that adds code to capture and to restore the state of the program stack at various execution points. Ara proposes modifications to the JVM interpreters. Since thread state migration is not a specific challenge for heterogeneous device, the Roam system supports only weak mobility.

Sumatra (Acharya et al., 1997) is an extension of Java that supports resource-aware migrateable mobile applications. Sumatra's resource-awareness is based on a monitor-feedback-adaptation loop. A resource monitor watches the level and quality of the system resources, and provides updates to the applications either continuously or on-demand. Applications can then adapt based on the resource updates, e.g., migrating some threads to remote devices to perform the computation. Sumatra is focused on building adaptive applications in the PC or PC-equivalent device environment, and adaptation is based on runtime resource level and quality. This is different from the Roam system, which is focused on adaptation in a heterogeneous device environment. Roamlets adapt according to target device capability at migration time and load time only.

We have found parallel and ongoing research efforts at IBM (Banavar et al., 2000). The IBM researchers have proposed a new application model for pervasive computing. They have described several research challenges, e.g., device-specific rendering and application apportioning, which are similar to the problems that the Roam system is addressing.

4.2. Multi-platform user interfaces

The model-based approach offers an attractive alternative to build a high-level tool for multi-platform UIs (Szekely, 1996). In model-base systems such as Humanoid (Szekely et al., 1993), ITS (Wiecha et al., 1990), UIDE (Sukaviriya et al., 1993), and Mickey (Olsen, 1989), UI designers would use a declarative language to specify abstract and high-level *models* that describe what UI should be. The model-base system would automatically generate low-level UI executable code. The model-based technique can be exploited for multi-platform UI generation. For example, UI designers can specify the high-level models using AIOs (abstract interactive objects), which are device-independent objects. The AIOs are mapped to device-dependent CIOs (concrete interactive objects) supported by the target platform UI library. This AIO-CIO selection technique has been illustrated in model-based systems for purposes other than multi-platform UI generation. For example, Humanoid uses a "replacement hierarchy" for selecting the most appropriate presentation template for displaying a semantic object. UIDE contains constraints on how to select the most appropriate interface actions to represent application actions specified by the UI designers in application models.

In recent work, RedWhale software (Eisenstein et al., 2000, 2001) employs the model-based approach by introducing abstract models that specifically target multi-platform UIs. It is comprised of platform model, presentation model, and task model. The platform model allows the UI designers to specify platform constraints that the UI generation must follow, such as device screen sizes, input methods, etc. The presentation model specifies the visual appearance of the generated UI, which may include the layout of the UI components and the mapping between AIOs and CIOs. The task model specifies the decomposition of big tasks into smaller subtasks. The device-dependent UI generation is based on all three models.

Another recent model-based work is Paternò et al's ConcurTaskTree (Paternò, 2000). Its task model contains user tasks, abstract tasks, interaction tasks, and application tasks. Each task is related to each other via a set of temporal relationships such as enabling, deactivation, and iteration. The ConcurTaskTree is generally accepted to be a powerful task model to construct UIs. However, it is mainly targeted to single platform UIs. If we try to apply ConcurTaskTree to generate multi-platform UIs, all UIs will have the same interaction pattern as the interaction tasks are tightly coupled in the model. However, devices such as cell phones and PDAs have distinctive interaction patterns. To solve this problem, our device-independent representation only has user tasks. The interaction pattern information is expressed as properties of the task, e.g., task preference, or as platform-specific interaction models that would interact with the task model. The latter expression will be our future work. With our task model, we can easily customize the interaction pattern for each platform.

Although the model-based approach is an attractive technique for creating SGUI tools, it has usability and authoring issues (Myers, 1995) such as loss of fine-grained control of UI details, the quality of generated UI, and the time needed for UI designers to learn and construct models. Our SGUI toolkit combines the model-based approach with the benefits of traditional widget-based UI programming, as a solution to some of these issues. Programming with the SGUI library is very similar to programming with a traditional widget library such as Swing, because the SGUI library is consisted of low-level UI widgets rather than abstract models. The transformation in SGUI is at the level of widgets and events, rather than from abstract interaction models to widgets and events. This has additional benefits that no code generation is necessary, and that complex models

of *UI Logic* (data flow constraints, sequencing, side effects, etc.) (Szekely et al., 1992) are also not needed.

Microsoft has a commercial product available called the. NET Mobile Internet Toolkit (Microsoft Corporation, xxxx), which targets different mobile devices ranging from the more capable PocketPC platform to relatively simple pagers. The Mobile Toolkit does not use a model-based approach. Rather, it assumes developers have already implemented desktop PC versions of the user interface components, and that they want to port from existing code. The Mobile Toolkit allows developers to specify the corresponding UI presentation on mobile platforms, but the layout specification is very limited. It only offers flow-based layout (rather than the more sophisticated Grid-Bag layout). In addition, developers must group widgets together into a *form*, the function of which is very similar to our LogicalPanelNode. However, .NET mobile forms cannot be placed on the same page even if a page has enough space to accommodate them.

5. Conclusion

In this paper, we present challenges, design, and implementation of the Roam system. The Roam system is a seamless application framework for building seamless applications that can migrate at runtime across heterogeneous devices. The Roam system provides adaptation strategies at the component level, including dynamic instantiation, offloading computation, and transformation.

There are many future directions to improve the Roam system. One problem with the existing SGUI toolkit is that it is difficult to customize a device-independent representation for a particular device. The reason is that the device-specific customization requires developers to add transformation rules that are both device-specific and application-specific. When the developers change the device-independent model at a later time, they may also have to update these transformation rules that are affected by the change.

We also like to support seamless application migration for real-time applications such as video conferencing. For example, a use may want to migrate a video conferencing from a mobile phone to a car navigation system when he/she is entering a car. This places a real-time constraint on migration latency. We want to make sure that the interruption time is minimized during application migration. One possible approach is to hide the migration latency, by delaying the application termination on the source device, until adaptation has been applied to the application components on the target device.

Finally, we would like to support a Roam execution state repository service where a user can save a Roam application on one device, and restore it at a later time on any device. This overcomes the limitation in the current Roam system where application migration cannot be suspended. We would like to extend this to collaborative applications where each user can save and restore application execute state individually without affecting other users.

6. Uncited references

Chu et al. (2001), NTT DoCoMo (2001), Sukaviriya et al. (1994), Sun Microsystems (1998, 2000a,b) and Wiebus (xxxx).

Acknowledgements

We would like to thank Dan Rosen for his help in proof-reading this document. We would also like to thank reviewers of JSS for their comments.

References

Acharya, A., Ranganathan, M, Saltz, J., 1997. Sumatra: a language for resource-aware mobile programs. Mobile Object Systems, Lecture Notes in Computer Science April, 1997.

Aramira, Inc., 1999. Jumping Beans™ White Paper, October 1999. Available from <http://www.jumpingbeans.com/index.html>.

Banavar, G., Beck, J., Gluzberg, E., Munson, J., Susman, J., Zukowski, D., 2000. Challenges: an application model for pervasive computing. In: Proceedings of the 6th Annual International Conference on Mobile Computing and Networking, Boston, MA, August 2000, pp. 266–274.

Chu, H., Song, H., Wong, C., Kurakake, S., 2001. Seamless Applications over Roam System, UbiTools'01 (Part of ACM UbiComp'01), September 2001. Available from <http://choices.cs.uiuc.edu/UbiTools01/>.

Eisenstein, J., Vanderdonckt, J., Purerta, A., 2000. Adapting to mobile contexts with user-interface modeling. In: Proceedings WMCSA'00, December 2000.

Eisentein, J., Vanderdonckt, J., Puerta, A., 2001. Applying model-based techniques to the development of uis for mobile computers. In: Proceedings IUI' 01, January 2001.

Fünfrocken, S., 1998. Transparent migration of Java-based mobile agents (capturing and reestablishing the state of Java programs). In: Proceedings of the 2nd International Workshop on Mobile Agents, Stuttgart, Germany, September, 1998.

Gray, R.S., Kotz, D., Nog, S., Rus, D., Cybenko, G., 1997. Mobile agents for mobile computing. In: Proceedings of the Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis, Fukushima, Japan, March, 1997.

Lange, D.B., Oshima, M., 1998. Mobile agents with Java: the Aglet API. World Wide Web Journal.

Linton, A., Vlissides, M., Calder, R., 1989. Composing user interfaces with interviews. IEEE Computer 22 (2).

Masui, T., 1994. Evolutionary learning of graph layout constraints from examples. In: Proc. of ACM UIST'94, November 1994, pp.103–108.

Microsoft Corporation, Best Practices for the Microsoft Mobile Internet Toolkit Image Control. Available from <http://msdn.microsoft.com/theshow/Episode023/default.asp>.

Milojicicc, D.S., LaForge, W., Chauhan, D., 1998. Mobile objects and agents (MOA). In: Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS). Santa Fe, New Mexico, April, 1998.

Mitsubishi Electric ITA Horizon Systems Laboratory, 1998. Mobile Agent Computing A White Paper, January 1998.

Myers, B., 1995. User interface software tools. ACM Transactions on Computer Human Interaction 2 (1), 64–103.

NTT DoCoMo, Inc., 2001. i-mode Java Content Developer's Guide, May 2001.

ObjectSpace, Inc., Voyager. Available from <http://www.objectspace.com/products/voyager>.

Olsen, D., 1989. A programming language basis for user interface management. In: Proceedings of CHI'89, Austin, Texas, May 1989, pp. 171–176.

Olsen, D., Jefferies, S., Nielsen, T., Moyes, W., Fredrickson, P., 2000. Cross-modal Interaction using XWeb. In: Proc. of ACM UIST'00, November 2000.

Paternò, F., 2000. Model-Based Design and Evaluation of Interactive Applications. Springer-Verlag, London.

Peine, H., Stolpmann, T., 1997. The architecture of the ara platform for mobile agents. In: Proceedings of the 1st International Workshop on Mobile Agents, Berlin, Germany, April 1997.

Rogaway, P., Bellare, M., Black, J., Krovetz, T., 2001. In: Eighth ACM Conference on Computer and Communications Security (CCS-8). ACM Press, pp. 196–205.

Strasser, M., Baumann, J., Hohl, F., 1996. Mole—a Java based mobile agent system. In: 2nd ECOOP Workshop on Mobile Object Systems, Linz, Austria, July 1996, pp. 28–35.

Sukaviriya, P., Foley, J.D., Griffith, T., 1993. A second generation user interface design environment: the model and the runtime architecture. In: Proceedings InterCHI'93, April 1993, pp. 375–382.

Sukaviriya, P., Kovacevic, S., Foley, J.D., Myers, B., Olsen, D., Schneider-Hufschmidt, M., 1994. Model-based user interfaces, what are they and why should we care? In: Proceedings UIST'94, November 1994, pp. 133–135.

Sun Microsystems, 1998. PersonalJava™ Technology—White Paper, August.

Sun Microsystems, 2000. Java™ 2 Platform, Standard Edition White Paper, June 2000.

Sun Microsystems, 2000. Java™ 2 Platform Micro Edition(J2ME™) Technology for Creating Mobile Devices, White Paper, May 2000.

Szekely, P., 1996. Retrospective and challenges for model-based interface development. In: Proceedings of 3rd Int. Workshop on Computer-Aided Design of User Interfaces CADUI'96, June 1996, pp. xxi–xliv.

Szekely, P., Luo, P., Neches, R., 1992. Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design. In: Proceedings CHI'92, May 1992, pp. 507–514.

Szekely, P., Luo, P., Neches, R., 1993. Beyond interface builders: model-based interface tools. In: Proceedings InterCHI '93 April 1993, pp. 383–390.

Vanderdonckt, J., 1995. Knowledge-based systems for automated user interface generation: the trident experience. Technical Report RP-95-010, Fac. Univ. de N-D de la Paix, Inst. D'Informatique, Namur.

White, J., et al., 1997. System and Method for Distributed Computation Based upon the Movement, Execution, and Interaction of Processes in a Network, US patent no. 5603031, February 1997.

Wiebus, S., Connect4 Applet. Available from <http://www.physics.adelaide.edu.au/~swright/java_apps/Connect4/>.

Weicha, C., Bennett, W., Boies, S., Gould, J., Green, S., 1990. ITS: a tool for rapidly developing interactive applications. ACM Transactions on Information Systems 8 (3), 204–236.

W3C, 2000. Scalable Vector Graphics (SVG) 1.0 Specification, November 2000. Available from <http://www.w3.org/TR/SVG/>.