Pervasive applications through scripted assemblies of services

David Svensson, Görel Hedin, Boris Magnusson
Department of Computer Science, Lund University
Ole Römers v 3, 223 63 Lund, Sweden
Email: {david,gorel,boris}@cs.lth.se

Abstract—**This paper proposes a technique for letting end users build pervasive applications by combining services on networked devices. The approach avoids relying on standardized service interfaces which are deemed too limiting, and instead makes use of migratable user interfaces and scripted combinations of services.**

I. Introduction

In a world of pervasive computing, people will encounter a wealth of devices that offer software services in (typically wireless) networks. These services will often be tied to the particular devices, enabling control of and interaction with the devices in powerful ways. We argue that in this setting, interoperability is bound to become a major challenge.

A typical need that can be foreseen is the possibility to combine services, utilizing the combined functionality of several devices. Support for this can facilitate repeated use of a set of connected devices, and also provide inherently new functionality, not given by the individual devices themselves.

However, the device vendors cannot be expected to foresee all possible combinations of services that can be demanded by future users; combinations possibly including future services and devices. This makes the usual approach, where one service interacts directly with another service through a standardized service-specific interface, too limiting. Instead, we propose that the combination of services should be separated from the services themselves, and that this combination is scripted, rather than programmed, to make it easy to adjust by end users. This will allow individual services to be developed independently of other services, but still be integrated into combined services.

In this paper, we describe a mechanism called *scripted assemblies*, that supports such combination of independent services. We have built an experimental system based on this approach, and tried it out on example scenarios. The ideas build on the MUI system [1], that supports remote control through migratable user interfaces. The work has been carried out within the EC-funded integrated project PalCom [2].

The rest of this paper is organized as follows. In Section II, we present our basic approach. Sections III to V go into more detail about non-scripted and scripted assemblies, dealing with issues in the assembly description language. Section VI relates to other work in this area, and Section VII lists some things that remain to be investigated and developed. Finally, Section VIII concludes the paper.

II. Basic approach

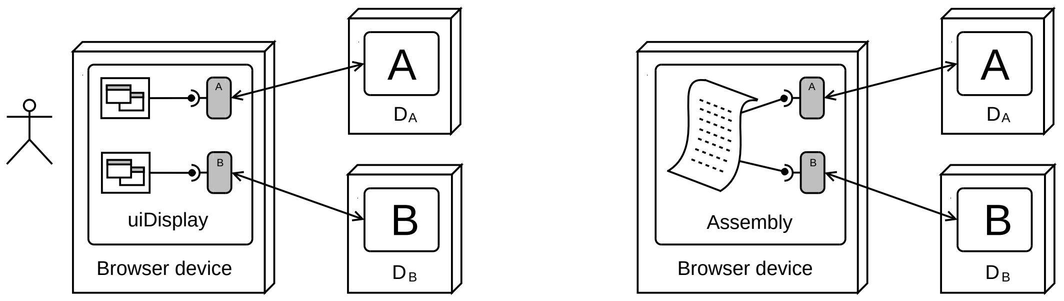
Figure 1 illustrates our approach to dealing with interoperability. There are two services, A and B, located on two different devices, D_A and D_B. The user wants to use and combine these two services. To accomplish this, the user has a third *browser* device that supports device and service discovery. Typically, the browser device is a handheld like a PDA or a mobile phone, but it could also be a general-purpose computer, e.g., a laptop. Each of the services A and B has a service description that can be migrated to the browser device, and rendered as a user interface there, in order to remotely control the service.

In Figure 1(a), the user interacts with A and B through these migrated user interfaces that are rendered on the browser device. This remote control mechanism is provided by a service *uiDisplay* that runs on the browser device. The uiDisplay service receives service descriptions from remote services, creates corresponding user interfaces that are shown on the screen of the browser device, and connects to the services over the network.[1] In the figure, the service descriptions are the gray boxes with a small "hook". After this setup process, there is a two-way, peer-to-peer communication between uiDisplay and each of the remote services A and B. When the user performs an action in one user interface, a command is sent over the connection to the remote service, which can react appropriately. When something happens at the remote service, typically as a result of physical interaction with the device on which the remote service lives, a command is sent in the other direction, typically updating status information in the user interface on the browser device.

The interaction through user interfaces solves parts of the problems with standardization of service interfaces. In this case, the human is in the loop, and can make intelligent interpretations of changes in the service descriptions (which show up in the user interfaces). When a new feature is added to a device, perhaps through an update of its firmware, a change in the service can be directly spotted and utilized. There is nothing in uiDisplay itself that is tied to the specific service.

The other part of the figure, 1(b), shows how interaction with the services can be automated, realized through an *assembly*. The assembly is a service which performs much the

[1]The migration of user interfaces, and adaptation to different client platforms, is a research area in itself. See, e.g., [3].



(a) The user interacts with two services, A and B, through user interfaces that are rendered on his browser device. The user interfaces are generated according to service descriptions (shown as small boxes with "hooks").

(b) An assembly on the browser device coordinates A and B, automating the desired combined behavior.

Fig. 1. Our approach to interoperability between services.

same function as uiDisplay, but instead of rendering service descriptions as user interfaces, it coordinates the services according to an *assembly descriptor* (shown as a scroll). The assembly descriptor contains a specification of a combination of services, residing on different devices. Simple assembly descriptors just specify a number of connections between the services, while more advanced assembly descriptors also contain a script, which coordinates the interaction in a more fine-grained way. The assembly shown in Figure 1(b) is of the latter type. It specifies how different events received from A lead to one or more commands sent to B.

The key point in our approach is that *interoperation is separated from services*. This is what makes it possible to combine groups of services that were not created together, without restraining all of them to use standard service interfaces that were already established when the oldest service was created. In the case of user interface rendering, the user controls the interoperation directly, something which is good for trying things out in order to see how they work. This can be impractical, though, for more complex or long-lasting tasks. For these, the assembly works better. The goal is that assemblies should be possible to create and modify by end users. For this reason, we propose that assemblies should be created using a scripting language, rather than using a general-purpose programming language that would require programming skills. In some cases, the interoperation of two services might, however, require programming. For example, in order to convert between different kinds of real-time data streaming formats. To handle this, we propose that such problems are delegated to separate *software services*. Such software services need to be programmed, but can be used and combined in an assembly by an end user.

III. Assembly representations

Our model of an assembly consists of the following parts:

1) A set of devices
2) A set of services (on those devices)
3) A set of connections between those services
4) A set of offered synthesized services, generated by the assembly
5) Logic and scripts defining and constraining how the assembly should be deployed and executed.

Furthermore, an assembly can be fully bound, forming a composition of particular identified services on particular identified devices, or it can be in various ways partially unbound, e.g., to act as a template. In this paper, we will focus on fully bound assemblies.

It is useful to discuss the assembly from several different perspectives: the end user, the expert user, the tools manipulating the assembly, etc. In our system, we use the following important representations of the assembly:

1) An XML representation that is used for storing the assembly in a file system, and for moving or copying it between different devices.
2) A concrete syntax that is used in documents like this to show the same details as in the XML representation, but in a syntax more readable by humans. In principle, this concrete syntax could also be used by editing tools on laptops for creating or editing assemblies by expert users.
3) A representation as an attributed abstract syntax tree (AST) that is used internally by tools accessing and manipulating the assembly. We use the JastAdd compiler construction system for supporting AST programming [4], allowing the internal tools to add computations on the AST as modular aspects. The XML and concrete syntax can trivially be unparsed from the AST representation, programmed as simple JastAdd aspects.
4) Tool-specific editing representations for displaying parts of the assembly information to end users, often in a visual way. E.g., a PalCom browser device can display the connections between services as lines between boxes, and provide graphical commands for composing or changing an assembly.

In this paper, we will use the concrete syntax when providing examples. In Section IV we will discuss simple assemblies with devices, services, and connections, but without scripts. In

Section V we will discuss how scripts can be added to capture the execution logic of an assembly.

IV. Simple assemblies

A. A remote slide show assembly

Some simple assemblies consist only of a set of connections between services on particular devices, and have no logic of their own. As an example, consider an assembly RemoteSlideShow which composes a video projector, a laptop, and a PDA. Slides are sent from the laptop to the video projector, and the user controls the actions, next slide, previous slide, etc., from the PDA. The assembly itself resides on the PDA.

To illustrate the use of this assembly, consider the following scenario:

> The user is a university professor who has weekly lectures in room E:1406 at the university. On the first lecture, she creates the assembly RemoteSlideShow by connecting her laptop, her PDA, and the video projector in room E:1406. This is done by a few visual commands on the PDA. On the PDA, she then uses commands to select the desired presentation, and to step through the slides. At the next lecture, she simply activates the existing assembly, which will then discover and connect the devices according to the assembly description. She can then immediately select the appropriate presentation and step through the slides.

B. Local device and service names

The RemoteSlideShow assembly is shown in Figure 2 in concrete syntax. The assembly introduces a number of local device names: projector, laptop, pda; and a number of local service names: control, images, uiDisplay, and imageViewer. These local names are used within the assembly, e.g., to define the connections, and inside assembly scripts (discussed later).

Typically, the local names are taken from the logical names used in the device and service descriptions of actual devices. But refactoring to other names inside the assembly (for greater readability), would not affect the behavior of the assembly. These names are not used for binding the assembly to real devices and services. For such binding, the global names are used (see below).

In the proposed language, the service names are simple rather than structured. However, it could easily be generalized to support structured names. This would be useful since the services on a device are typically structured in a hierarchy, and it would be useful to keep that hierarchy in the local names of the assembly.

C. Global names of devices and services

Devices and services are identified by globally unique names. The globally unique names have an internal structure including a globally unique identifier, versioning information, and a logical name (which does not need to be unique). Typically, these names are quite long, and not intended to be very readable to a human. In the example in Figure 2, we simply display them as "global-device-name" and "global-service-name". These unique names are used for making it possible to reconnect an assembly to the same devices and services as used when the assembly was constructed. The assembly also has such a unique name itself (the value of "this"), with the same structure as a service name. The versioning information in the globally unique names is used by tools to make safe upgrades of an assembly when a service or a device has been upgraded. Note, however, that when an assembly is upgraded (rebound device and/or service), this has to be somehow visible to the user, and testing might be needed (unless it can be deduced that testing has already been carried out).

Two different devices, e.g., two projectors, can have (different instances of) the same service on them. To uniquely identify a service instance, i.e., a service on a particular device, the global names of the hosting device and the service are combined.

D. Connections

The connections part in the assembly specifies how the services in the assembly are connected to each other, using clauses on the following form:

```
providing-service on device-1 ->
customer-service on device-2
```

Connections can be either data connections (uni-directional), sending messages from provider to customer, or control connections (bi-directional), where messages can go in both directions. For example, in Figure 2, the connection `images on laptop -> imageViewer on projector` is a data connection where JPEG images are sent from the laptop to the projector. The connection `control on laptop -> uiDisplay on pda` is a control connection. As an example of messages over this connection, the PDA can send a message "next" to the laptop, to go to the next slide. The laptop will then send a status message back to the PDA, showing the name of the currently shown slide.

MIME types are used for specifying the types of connections. For example, the type of the `image-imageViewer` connection is `image/jpeg`, whereas the type of the `control-uiDisplay` connection is `application/x-palcom-control+xml`. These types are not explicitly visible in the assembly, but belong to service descriptions that are available for each device through the discovery protocol. The service description also classifies a service as being either provider or customer.

E. Static-semantic constraints on the assembly

There are certain semantic constraints on how assemblies may be constructed. The local names should be declared and used correctly. E.g., two devices named the same way is forbidden, using an undeclared name is forbidden, using a device name where a service name is expected is forbidden, etc. This boils down to normal name and typechecking rules similar to those in simple programming languages.

F. Dynamic constraints on the assembly

There are additional semantic constraints that can be checked only dynamically, i.e., when trying to activate the assembly:

1) Device bound: When activating an assembly, the device declarations will be bound to descriptions of actually discovered devices. Naturally, it may be the case that it is not possible to discover a given device. It might be broken, turned off, not within range, etc. The operation of the assembly may then be limited for the moment.

2) Service bound: Even if a device is bound, it is not guaranteed that all its services are available. Some services may be down, depending on the state of the device. It might also be the case that when an assembly is changed so that a device declaration is rebound to another device, the new device does not have all the declared services, and this is then flagged as errors or warnings. The tools can then guide the user in trying to rebind to another service on the same device, or possibly to a service on another device.

3) Connection well formed: If the services of a connection are bound, it is checked that the connection is well formed. I.e., the service declared as the providing service should indeed be specified as a providing service in its service description, and similarly for the customer service. Furthermore, the MIME types of the provider and the customer should match.

V. Scripted assemblies

In the RemoteSlideShow example, the assembly simply connects existing services directly to each other. A more advanced assembly can itself receive and send messages and perform actions internally. These actions are written in a simple script language that can be used by an end-user. In the present experimental system, the script is edited as text, but in future versions, we plan to provide visual tools for editing the scripts. If the internal logic is more complex than the script language can handle, parts of the logic can be delegated to new software services, programmed in a general-purpose language.

Below, we extend the assembly representation to include a scripting possibility. The basic idea is that the assembly can be connected to other services to receive and send messages. The body of the script is an event handler that receives messages from other services and acts upon them. The possible actions (supported so far) are to send messages to other services and to store values in variables local to the script.

A. GeoTagger as a scripted assembly

GeoTagger is one of the end-user scenarios studied in PalCom, see Figure 3. It is an application intended for use by landscape architects. The idea is that photos taken with a camera should be automatically tagged with the current GPS coordinates and stored in a backend database on a laptop. This application is realized as a scripted assembly running on a handheld PDA. The assembly combines and coordinates services running on the camera, the GPS device, and the laptop.

Figure 4 shows the scripted assembly. In the event handler, clauses are written as

```
when message from service on device {
   actions
}
```

where the actions can access data in the message, send new messages to other services, and perform simple computations (assignments of local variables).

The service coordStuffer on the PDA device is a software service that can receive an image in JPEG format, and a GPS coordinate, and which sends out an image tagged with the GPS coordinate. This is a typical example of a computation that is too complex to express directly in the scripting language, and that is instead implemented as a software service.

As shown in the example, the assembly interacts with other services by receiving and sending messages. Thus, the assembly implicitly plays the role of a service that connects to the other services. The “this” expression used in the assembly script refers to the assembly itself viewed as a service. Received messages that are not listed in the event handler are simply ignored.

B. Additional constraints on the assembly

The introduction of the script in the assembly makes it possible and necessary to check additional constraints, statically

```
assembly RemoteSlideShow {
  this = global-service-name;
  devices {
    projector = global-device-name;
    laptop = global-device-name;
    pda = global-device-name;
  }
  services {
    control on laptop = global-service-name;
    images on laptop = global-service-name;
    uiDisplay on pda = global-service-name;
    imageViewer on projector = global-service-name;
  }
  connections {
    control on laptop -> uiDisplay on pda;
    images on laptop -> imageViewer on projector;
  }
}
```

Fig. 2. A simple assembly.

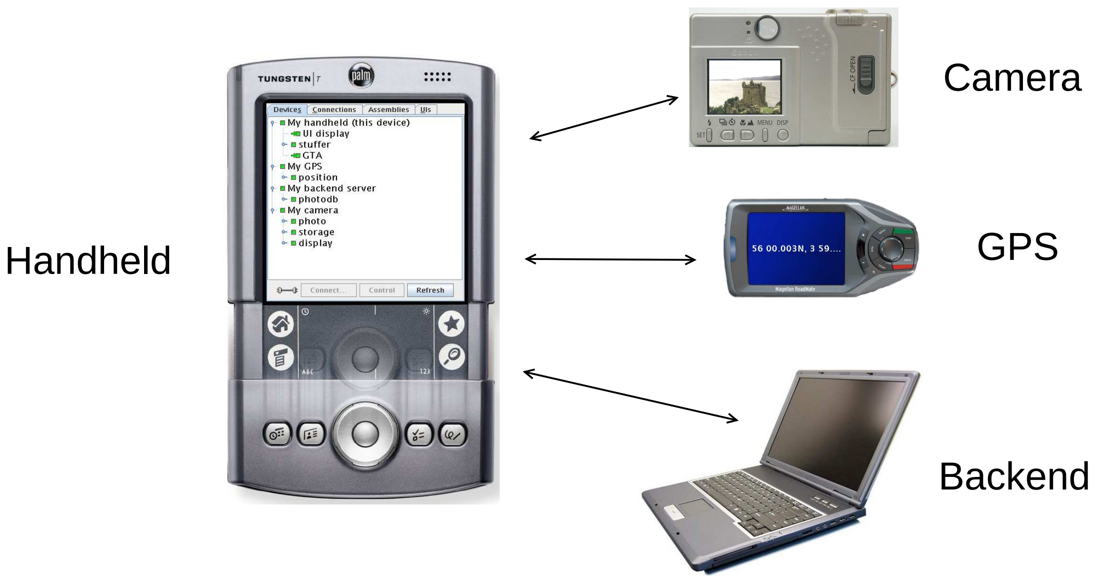


Fig. 3. The GeoTagger scenario

and dynamically. In the static part, the name and type analysis is extended to the local variables. In the dynamic part, it is checked that the bound services actually have the incoming and outgoing messages used in the script, with the appropriate message structure.

C. *Loopback mechanism*

It might be the case that the assembly is located on the same device as some of the other services. A loopback mechanism is used which allows the assembly to communicate in the same way with these services as with services on other devices, without causing any messages to go out unnecessarily on the network.

The loopback mechanism is used also if the assembly connects two services on the same device: the network is transparent, and messages between services will only go out on the network if the services are on different devices.

Note that it is often the case that services on the same device are tightly bound and communicate with each other directly (not via an assembly). For example, when taking photos with a digital camera, the photos will be stored locally on the camera. This process is a bottleneck and needs to be carried out as efficiently as possible, to allow pictures to be taken at high speed.

Assemblies for connecting services on the same device are useful when the services are more unrelated, i.e., when they could in principle be located on different devices, but just happen to be located on the same device.

D. *Moving the assembly?*

For a scripted assembly, its location can dramatically affect the efficiency. For the GeoTagger, there will be large messages sent that include JPEG images. Suppose the assembly is located on the PDA (a natural choice since an assembly interpreter will need some kind of general-purpose platform to run on). In this case, JPEG images will be sent from the camera to the PDA, coordinates added to the image on the PDA, and the “stuffed” image is sent to both the camera and the laptop backend.

Clearly, if the coordinate stuffer and the assembly were moved to the camera or to the laptop, network traffic would be substantially reduced. It might be possible to move them to the camera if it is sufficiently advanced to serve as a general-purpose software service platform. And moving them to the laptop should be possible, but then the assembly would rely on the laptop which might be heavy for the user to always carry with him.

Note that if the assembly is moved to another device, its script does not need to change. There is nothing in the assembly script that makes it depend on its own platform.

If the assembly and coordinate stuffer are moved to the camera or laptop, it might still be the case that the user would like to control the assembly from the PDA, e.g., to activate it. Future versions of our system will support this by using an uiDisplay service for remote control of assemblies.

VI. Related work

The scripted assemblies we propose are related to W3C’s Web Services Choreography Description Language [5]. WS-CDL *choreographies* are expressed in an XML language, and govern peer-to-peer interoperation between a number of services. Like our assemblies, the choreographies are external to all the participating services. One thing that differs is the context. WS-CDL is intended for E-business, taking place between Web services on the Internet. There is no notion of physical devices, which are important in our approach, and in pervasive computing in general. The purpose of WS-CDL is also not on keeping interoperability between services when facing service interface changes. Instead, the choreography is more like a contract that is decided on before a business

```
assembly GeoTagger {
  this = global-service-name;
  devices {
    gps = global-device-name;
    camera = global-device-name;
    backend = global-device-name;
    pda = global-device-name;
  }
  services {
    gps on gps = global-service-name;
    photo on camera = global-service-name;
    storage on camera = global-service-name;
    display on camera = global-service-name;
    coordStuffer on pda = global-service-name;
    photo_db on backend = global-service-name;
  }
  connections {
    gps on gps -> this;
    photo on camera -> this;
    storage on camera -> this;
    display on camera -> this;
    coordStuffer on pda -> this;
    photo_db on backend -> this;
  }
  script {
    variables {
      text/plain latestReadableCoordinate;
      text/nmea-0183 latestStandardCoordinate;
    }
    eventhandler {
      when position from gps on gps {
        latestReadableCoordinate = thisevent.WGS84;
        latestStandardCoordinate = thisevent.NMEA-0183;
      }
    when photo_taken from photo on camera {
        send show(latestReadableCoordinate) to display on camera;
        send sendme_photo() to storage on camera;
      }
    when photo from storage on camera {
       send sendme_stuffed_image(
          latestStandardCoordinate, thisevent.Photo)
        to coordStuffer on pda;
      }
    when stuffed_image from coordStuffer on pda {
       send store_photo(thisevent.Image) to photo on backend;
        send store_photo() to storage on camera;
      }
    }
  }
}
```

Fig. 4. A scripted assembly.

relationship is started, making it possible for all parties to keep the internals of their services private.

Another closely related project is Obje at PARC [6]. Obje targets the same basic problem, and seeks to enable interoperability without relying on domain-specific standards. A difference is that Obje builds on mobile code. Using mobile code, in the form of a proxy object that is distributed to clients and executed there, services are able to "teach" clients how to communicate. This way, it is possible to let users combine their clients with new services, some of whose features were unknown at the time the clients were written. There is also a possibility to let the proxy object generate a user interface, giving a situation similar to that of Figure 1(a), where the proxy object corresponds to our service description. Another difference, though, is the way of programmatically interfacing the proxy objects. Obje proxy objects are (Java) code, which requires the capability of running Java on clients, and their interfaces are so called *meta interfaces*, offering only very generic operations, such as reading a chunk of data. In contrast, our service descriptions are distributed as XML, which can be handled on almost any device, and they contain domain-specific operations: the operations are invoked by the user through a user interface, or by the assembly script. There is no concept in Obje corresponding to the assembly. Instead, the user directly connects components written in Java.

Cooltown at HP Labs [7] is an early pervasive computing project, whose target is to bring the Web to things in the physical world. By embedding wirelessly accessible web servers into things, it is possible for a user to interact with them in a Web browser on his handheld device. It is also possible to connect one device to another, by sending a URL to one of the devices, identifying a resource on the other. There is nothing domain-specific in the Web protocols involved, so this can be seen as a way of achieving basic parts of the interoperability we look for. But, apart from the client-server model being inherent in the interaction between Web clients and servers, one big difference is that our assemblies can define other aspects of a service interoperation than just pure connections.

Jini and UPnP are important technologies for network services. Jini [8] is tied to the Java programming language, and clients interact with services through proxy objects, distributed to the clients at discovery time. Our objection, also stated by Obje, is that this approach requires the interfaces of the proxy objects to be standardized at the domain level. To partly overcome this, there is a framework for user interface services built on top of Jini [9]. But, still, the tight connection to the Java language makes it inconvenient to build assemblies on top of Jini. UPnP [10] is not tied to Java, or to another programming language: devices and services are described using XML. But the focus in UPnP is on standardization of device types at the domain level. There are standards for devices such as printers, scanners, lighting controls, and digital security cameras, among others [11]. Therefore, UPnP is not directly usable as a platform for assemblies, either.

VII. Future work

In our continued work on scripted assemblies, we will look into a number of issues including synthesized services, binding of services, message types, and service versions. Synthesized services are services that are offered by the assembly itself, allowing control of the combination of services, rather than of each service individually. A simple case of a synthesized service could be to collect the most important parts of the participating services' interfaces into one interface, for convenience. Another case could be to have an interface for changing the activity state of the whole assembly.

So far, we have considered only fully bound assemblies where each service declared in the assembly is bound to a specific service on a specific device. We will look into more elaborate support for service bindings. One example is to investigate cases where an assembly can be functional without all services being present. This may give degraded, but acceptable, functionality of the assembly. In other cases, it may be enough with one out of a set of services for full functionality. There are also possibilities for experimenting with partially bound assemblies, where the identity of a device or service is not filled in, but can be specified later, perhaps when the assembly has been moved into a new context. In relation to this, versioning of assembly descriptors becomes important.

Currently, we demand exact matching of MIME types for connecting services. However, we will investigate the use of subtyping to allow services to be connected where the types match only partially.

VIII. Conclusions

This paper has presented scripted assemblies as a technique for letting end users combine services, and for letting them control the cooperation between services in a script. The assembly concept allows the interoperation between services to be separated from the services themselves. As a consequence, it is possible to adjust aspects of the interoperation at a later time, without re-programming the services, and to incorporate services with different, or changed, interfaces, by manipulating the assembly only. We see this as a way of easing interoperability in pervasive computing systems.

In the paper, the current language of assembly descriptors has been presented, exemplified by scenarios from the PalCom project, and possibilities for future development and experimentation have been discussed.

References

[1] D. Svensson, B. Magnusson, and G. Hedin, "Composing ad-hoc applications on ad-hoc networks using MUI," in *Proceedings of Net.ObjectDays 2005, 6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World*, Erfurt, Germany, September 2005, pp. 153–164.

[2] PalCom. Palpable Computing: A new perspective on Ambient Computing. [Online]. Available: http://www.ist-palcom.org/palcom-info.pdf

[3] P. Rigole *et al.*, "A Component-Based Infrastructure for Pervasive User Interaction," in *International Workshop on Software Techniques for Embedded and Pervasive Systems STEPS'2005*, Munich, Germany, May 2005.

[4] T. Ekman, G. Hedin, and E. Magnusson. JastAdd: an open-source Java-based compiler compiler system. [Online]. Available: http://jastadd.cs.lth.se

[5] N. Kavantzas *et al.* (2005, Nov.) Web Services Choreography Description Language Version 1.0. W3C. [Online]. Available: http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/

[6] "Obje Interoperability Framework," Palo Alto Research Center (PARC), 2003, http://www.parc.com/research/projects/obje/Obje_Whitepaper.pdf.

[7] T. Kindberg *et al.*, "People, Places, Things: Web Presence for the Real World," in *Proc. 3rd IEEE Workshop Mobile Computing Systems and Applications (WMCSA 00)*, 2000, pp. 19–28.

[8] J. Waldo, "The Jini Architecture for Network-Centric Computing," *Communications of the ACM*, pp. 76–82, July 1999.

[9] B. Venners. (2005) The ServiceUI API Specifi cation, Version 1.1a. [Online]. Available: http://www.artima.com/jini/serviceui/Spec.html

[10] UPnP^{TM} Forum, "UPnP Device Architecture 1.0," Tech. Rep., December 2003, version 1.0.1.

[11] ——. UPnP^{TM} Standards. [Online]. Available: http://www.upnp.org/standardizeddcps/