IST-002057 **PalCom**

Palpable Computing:

A new perspective on Ambient Computing

Deliverable 43 (2.6.2)

End-User Composition: Software support for assemblies

Due date of deliverable: m 37

Actual submission date: m 37

Start date of project: 01.01.04

Duration: 4 years

Lund University (LU)

Revision: 1.1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	PU
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Integrated Project

Information Society Technologies

1 Table of Contents

2 Executive Summary

WP6, on End User Composition, covers the aspects of the PalCom architecture and infrastructure where end users can put together services and devices in new, ad hoc, ways using *assemblies*. The focus is on end user interaction with the mechanisms for discovery and composition/decomposition of services and assemblies.

This deliverable focuses on the software developed to support end-user composition, and which is part of the PalCom Toolbox infrastructure:

- *Developer's Browser*. We have developed an interactive tool, the Developer's Browser, that supports end-user assembly development. It allows the user to browse discovered devices and services, to interact with services remotely, and to construct and to run assemblies. This browser is a high-end browser, aimed at advanced end-users as well as PalCom developers. It is implemented in Java as an Eclipse plugin, and is designed to run on a general-purpose computer with screen, mouse and keyboard.
- *Supporting Middleware*. To support the browser and other application-level software, middleware software has been developed in Pal-J (the restricted version of Java that runs on both JVM and PalVM). This middleware is useful for any PalCom device, not just browser devices. In particular, we have developed a *Service Framework*, an initial *Resource Manager*, an *Assembly Manager*, a *Discoverable Component Manager*, and support for *Simulated Devices*.

In addition, we discuss interaction with other workpackages, experiments we have made with these software tools, and how the work relates to the PalCom challenges.

2.1 Contributing authors

The following people have contributed to this deliverable:

- Görel Hedin, Lund University
- Boris Magnusson, Lund University
- David Svensson, Lund University
- Sven Robertz, Lund University

We have also benefited from constructive comments from the following people:

- Michael Christensen, Aarhus University
- Simon B. Larsen, Aarhus University
- Reiner Schmid, Siemens

3 Introduction

WP6, on End User Composition, covers the aspects of the PalCom architecture and infrastructure where end users can put together services and devices in new, ad hoc, ways. The focus is on end user interaction with the mechanisms for discovery and composition/decomposition of services and assemblies.

WP6 provides a link between programming level mechanisms, such as components and communication, and application prototypes and scenarios. The concepts developed in the project for services, components, and assemblies are made accessible to the end user in order to discover, compose/decompose and adjust palpable systems for specific end-user needs and deployment.

The previous deliverable from WP6, [Del25], characterizes the main concepts in PalCom, like service, connection, and assembly, from an end-user perspective. It outlines how end-users can compose services using assemblies, and characterizes typical categories of end users and typical categories of devices. For further treatment of the underlying ideas of assemblies and service browsers, see [SvMaHe05, SvMaHe06, Sv06].

The present deliverable focuses on the software developed within WP6: an end-user browser and supporting middleware, belonging to the application layer and the middleware layer of the PalCom ToolBox infrastructure. See [Del39] from WP2 for an overview of the infrastructure.

The rest of this deliverable is structured as follows: Section 4 provides a background discussion of the PalCom concepts that are of particular importance in this deliverable. Section 5 describes the Developer's Browser that has been implemented. Section 6 describes the middleware software that has been implemented to support the software at the application level, in particular the browser. Section 7 discusses the relation to other work packages. Section 8 concludes the deliverable, and gives a brief discussion of how the PalCom challenges are addressed by this work.

4 Background

To give a brief background for the continued discussion, we review a number of the key concepts of relevance to end users, all more thoroughly discussed in [Del25, Del39, Del41]. The concepts are described here according to their current implementation. For this reason, there may be some minor differences when comparing with other deliverables: sometimes the open architecture description is ahead of the implementation, and sometimes it is the other way around, reflecting the iterative development of the concepts.

4.1 Devices, services, and discovery

A *device* is a computerized hardware node such as a laptop, digital camera, gps, biosensor, etc.. A device typically has a number of *services*, often related to the hardware on the device. For example, a digital camera can have a service for remote control of the camera, with messages for taking photo and sending picture. Through these services it is possible to interact with the camera over a network. There is a PalCom *discovery protocol* that allows devices and their services to be discovered from other devices. There is a basic PalCom service protocol that allows connections to be set up between services and the services to communicate by message sending.

4.2 Service descriptions

Each service has a *service description* that describes its interface using an XML format. The description contains information about a *connection type*, that identifies the structure of its interface. In all our examples in this deliverable, we use the type *control*, which is a general connection type allowing a set of different messages to be sent and received. We have also experimented with other connection types, e.g., for *streamed data*.

A control service can either be a *control provider*, in which case the service description contains a description of which messages it can send and receive, or a *control customer*, in which case it is constructed to be connected to a specific control provider. Assembly Scripts and Remote Views, discussed below, are both control customers, whereas normal services on devices, like cameras, are control providers.

To combine services, an *assembly* is created that defines which services are combined and how. The assembly can contain a *script* with some logic of its own, and which serves as a control customer that can be connected to other control providers. The assembly can also contain descriptions of one or more *synthesized services* that serve as control providers, giving the assembly one or more external interfaces, allowing it to be used in other assemblies.

4.3 Service categories

There are different categories of services, and of particular interest in this deliverable are the following

- *Indigenous* service, i.e., services which are bound to the underlying hardware platform in such a way that it can only run on this particular device.[1] For example, the services of a camera will typically be indigenous. Indigenous services typically run automatically when the device is turned on.

[1] In [Del25] we used the term *native* service. The term *indigenous* is now part of the open architecture.

- *Discoverable component*, i.e., a general-purpose component that offers a service and therefore is discoverable (in contrast to other components that may be part of a device implementation, but which do not offer services). A discoverable component is general-purpose and is not tied to any specific device (it is not indigenous). It can therefore be copied and in some cases migrated from one device to another. Discoverable components can be *instantiated* to running services, at which point connections to them can be set up. The execution of a discoverable component can be started and stopped. If platform requirements are met, discoverable components can be copied to other devices, and they might be *migrateable*, in which case their running instances can be migrated to other devices.[2]
- An *Assembly* is an entity that uses other services and has a *script* for coordinating the interaction with these services. The assembly script serves as a control customer service. The assembly may also offer zero or more *synthesized services* which are control providers. They allow the assembly itself to be assembled by other assemblies. An assembly also plays the role of a discoverable component: it can be copied to another device, instantiated, started, and stopped. An assembly is also migrateable: a running instance can be migrated to another device.

4.4 Simple and advanced devices

Depending on need and hardware platform capabilities, devices may differ in what parts of the PalCom infrastructure they support, and thereby in what kind of services they can host. See Figure 3 in [Del39] for an overview of the PalCom infrastructure. Very simple devices host only indigenous services. More advanced devices can host discoverable components, assemblies, and/or interactive *browsers*.

- To host indigenous services, the device needs only to support the basic PalCom discovery and communication protocol.

- To host discoverable components, the device needs to provide a JVM or PalVM execution platform for running the component, and a Discoverable Component Manager (see later in this deliverable) which is a Service Manager [Del39] that can load, start and stop Discoverable Components.

- To host assemblies, the device needs an Assembly Manager and a Resource Manager (described later in this deliverable) that can store, load, and interpret assembly descriptions.

- To host a browser, the device needs interactive capabilities, and will typically be a laptop or a handheld computer. Simple browsers allow the user to browse devices and services on the network, and to interact with them in a simple way. More advanced browsers allow the user to interact remotely with services and to construct and run assemblies. The Developer's Browser, described in this deliverable, is an example of such an advanced browser.

---

[2] The term *discoverable component* was introduced in [Del25] but is not (yet) part of the PalCom Open Architecture [Del39]. The implementation currently uses the term *software component*. We are not entirely satisfied with either of these terms, but to avoid introducing an additional term before this concept has converged at the architectural level, we keep with the one introduced in [Del25].

4.5 End-user categories and assembly editing

An end-user is a person that makes use of PalCom devices and services. As discussed in more detail in [Del25] we envision different categories of end users, ranging from naïve end users that only operate turn-key PalCom systems, over intermediate users that can adjust existing PalCom assemblies, to advanced end users that construct new PalCom applications by assembling existing devices and services, and by scripting their interaction. None of these end user categories need programming skills.

Browsers are used for constructing or adjusting assemblies. Depending on the skills and needs of the end users, different kinds of browsers are relevant:

- Handheld browsers. These browsers are suitable for doing simple assembly construction and adjustments, but the limited screen size may make more advanced development of assemblies difficult. We have developed the *MUI browser* [Del25, SvMaHe05], as an example of such a browser.
- Laptop browsers. With the larger screen and better input capabilities (keyboard and mouse), laptops can host browsers with more advanced assembly editing functionality. The *Developer's Browser* and the *Visual Browser* are examples of laptop browsers.

The Developer's Browser is described in this deliverable. It is implemented as an Eclipse plugin [Eclipse], and is focused on supporting assembly developers rather than ordinary end users that simply need to adjust or inspect an assembly. Although the Developer's Browser can be used by very skilled end users, it is primarily intended for developers.

There is also a need for more intuitive browsers that are easy to use for a larger category of end users who mainly adjust or inspect assemblies, rather than construct new ones. For this purpose, a *Visual Browser* is under development as well, and a prototype of this browser has been used in some of the scenario prototypes [Del44], see also the screenshot in Fig 9 in [Del39].

Both the Developer's Browser and the Visual Browser focus on *manual* assembly construction, i.e., the user explicitly identifies particular services and devices that should take part in the assembly. There is also research in PalCom on adding *task-driven* assembly construction which make use of resource and contingency management to automatically adjust and reconfigure assemblies, see [Del42].

4.6 Scenarios

The different PalCom scenarios have allowed us to extract interesting requirements on the assembly mechanism, as was detailed in [Del25]. We have also used parts of the scenarios to illustrate different features of assemblies. In this deliverable, we illustrate the Developer's Browser by making use of the OnSite scenario, and, in particular, the GeoTagger: an assembly of a camera, a gps, and a storage server, that together accomplishes that all taken photos are tagged with gps coordinates and stored on the storage server. The scenarios are discussed in more detail in [Del44].

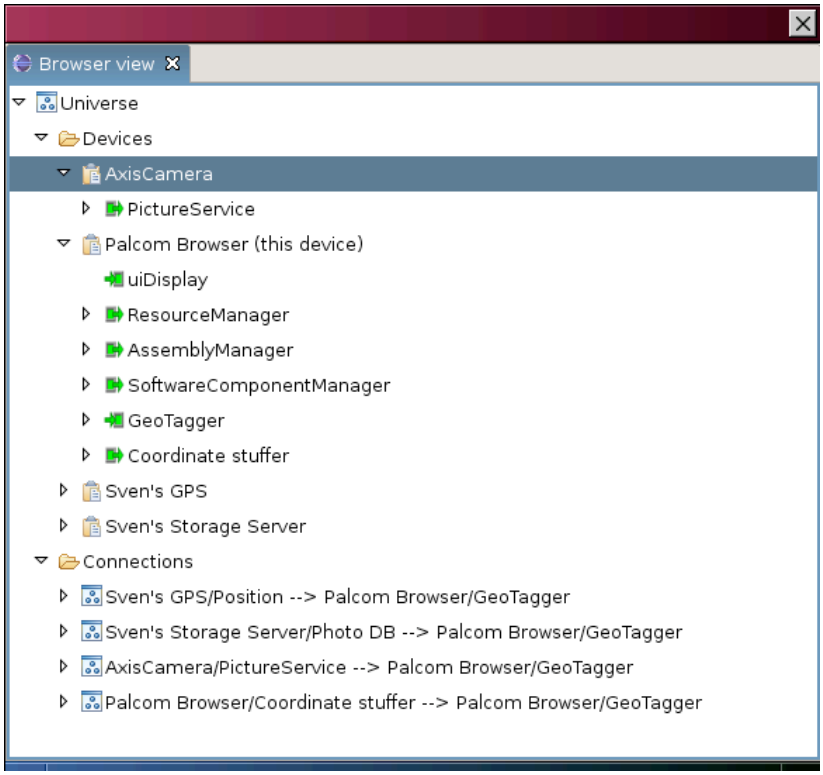
5 The Developer's Browser

The Developer's browser is implemented as a plugin to the Eclipse environment [Eclipse], and is thus implemented in Java, and designed to run on a computer with screen, keyboard, and mouse. The software comprising the Developer's Browser is available in the PalCom cvs repository, and will become open-source software later in the project. An installation and user's manual is available [WN113]. The Developer's Browser is built on the components available in the PalCom Toolbox and is thus compatible with existing PalCom software sharing the same codebase. This includes the Mui browser for handheld devices, which supports browsing and simple assemblies without scripts [Del25].

The Developer's browser provides the following fundamental features: Browsing of Devices and Services, Remote interaction with service, and Assembly editing. Each of these features appear as a "View" or "Editor" in an Eclipse window.

5.1 Device and Service Browsing

The figure below shows a screenshot of the *Browser view*. It shows the discovered devices and their services as a hierarchical clickable list.



- *AxisCamera* is a real PalCom device, a network camera that has been reprogrammed to talk the PalCom discovery and service protocol [MaJa07]. The AxisCamera provides a service *PictureService* for taking pictures, also shown in the hierarchical list.
- The browser presents itself as a device "`PalCom Browser (this device)`". It has a number of services including a *Coordinate stuffer* (a discoverable component) and *GeoTagger* (an assembly). Other services on the browser are various managers for middleware software that have been made into services, and thereby made accessible over the network:

ResourceManager, AssemblyManager, etc.. These managers will be discussed later in the deliverable.

- *Sven's GPS and Sven's Storage Server* are simulated devices that can be shown on the screen as separate windows (not shown here). See section 6.5 for a discussion on simulated devices.

The Browser view also shows the connections between services of the discovered devices. For example, the PictureService of the AxisCamera is connected to the GeoTagger on the browser device:

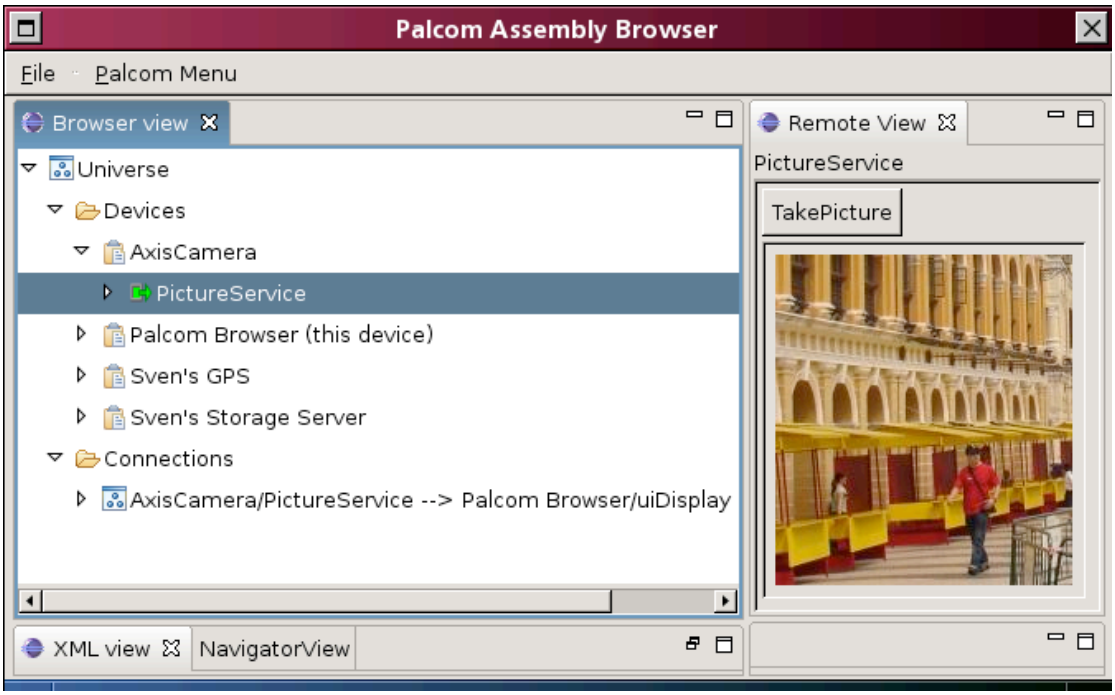
```
AxisCamera/PictureService --> Palcom Browser/GeoTagger
```

Similarly, the Coordinate stuffer, the storage server and the gps are also connected to the GeoTagger (the assembly).

The user can also interactively create new connections directly in the Browser view, thereby causing services to interact.

5.2 Remote interaction with a service

The figure below shows a screenshot of the *Remote view* that can be used for interacting with a service on a remote device. The figure shows a remote view of the PictureService on the AxisCamera. We can compare the remote view to an ordinary remote controller for a TV. Similar to a remote controller, the remote view has buttons for controlling the device, e.g., to make the camera take a picture: "`TakePicture`". In contrast to a remote controller, the remote view can also receive messages from the device and present information in the view, in this case, an image of the picture taken.

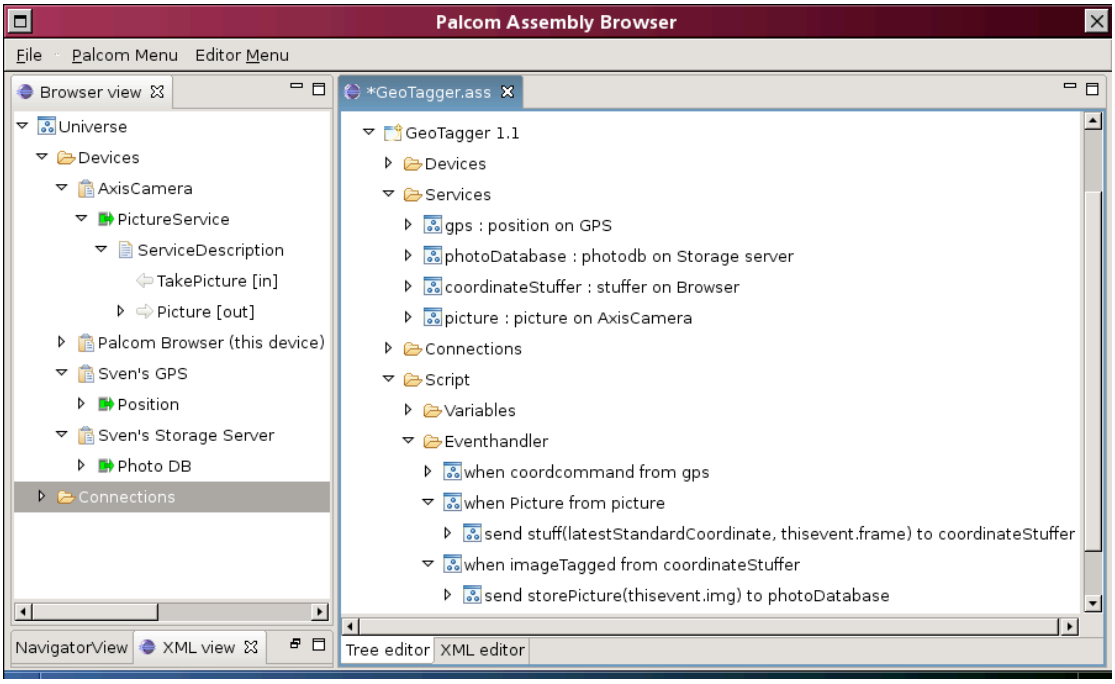


It is services of type *control provider* that can be displayed in a Remote view. The user can bring up a new Remote View by selecting a control service in the Browser View, and issuing a menu command *open Remote view*. The user interface of a remote view, with buttons, etc., is automatically rendered from the service description. This

camera has only one *control* service, and we can therefore think of this view as a remote view of the camera device itself. But it is also possible for a device to have several control services that provide different interaction interfaces. For example, we could imagine a camera with one control service for normal operation (taking pictures, etc.), another control service for changing the settings of the camera (e.g., in what precision to store pictures), and a third control service for upgrading the software of the camera.

5.3 Assembly editing

The figure below shows a screenshot of the *Assembly Editor*, opened on the GeoTagger assembly description. The description contains a list of *devices*, a list of *services* on those devices (*gps*, *photoDatabase*, *coordinateStuffer*, and *picture*), a list of *connections* between the services, and a *script*.



The script can have local *Variables*, and also an *EventHandler* that defines actions to be taken for each message received from the connected services. Typical actions are storing values in local variables, and sending messages to connected services.

In the example there are three when-clauses, taking care of three different messages (all details not shown):

- When a gps coordinate arrives, it is stored in a local variable.
- When a picture arrives, it is sent on to the coordinator stuffer, along with the current gps coordinate.
- When the stuffed image is received from the coordinate stuffer, it is sent on to the photo database.

Internally, the assembly is represented as an abstract syntax tree (AST). The AST provides a high-level object-oriented representation of the assembly that is suitable for the assembly manager and other managers to operate on. This allows separation between the internal abstract representation (the AST) and different ways of rendering the assembly, e.g., as an XML text, or as visual or semi-visual renderings. In the figure, the AST is rendered as a hierarchical list (called *Tree editor*). Here, the user

can edit the list by menu commands in order to add/delete/change descriptions of devices, services, and connections, and to edit the assembly eventhandler, etc.

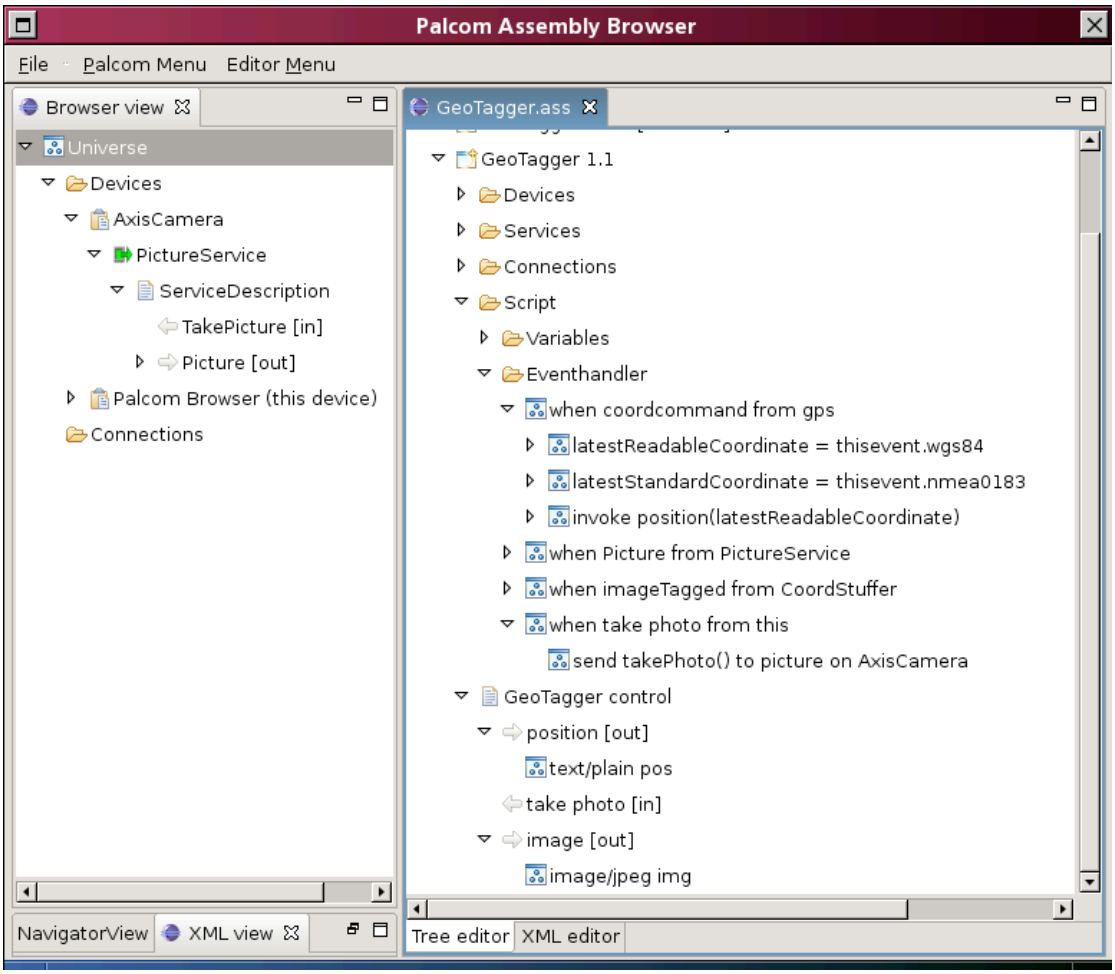
When creating an assembly, it is common to first experiment with the devices and services by setting up remote views, and by adding explicit connections between services in the Browser view. To create the assembly in an easy way, it is possible to simply drag the relevant devices, services, and connections from the Browser view to the Assembly Editor. By dragging over a connection, the relevant devices and services are automatically added to the assembly.

The assembly can also be edited as XML text. This is done by selecting the *XML editor* tab (details not shown). This can be useful during development and debugging of the browser itself, but is of course less intuitive than the tree editor. When switching to the XML editor, the AST is unparsed to XML text which the user can edit. After editing, the XML text is parsed and the AST is updated.

5.4 Synthesized services

An assembly can be equipped with a *synthesized service* that allows the whole assembly to be viewed as a single service. If the assembly has a synthesized service, the user can open a remote view of the assembly (on the same or another browser device). And the assembly can be combined into another assembly.

The figure below shows the definition of a synthesized service *GeoTagger control* (the type of the service is *control provider*).



The synthesized service has three commands:

- An *in*-command "`take photo [in]`" for taking a photo. The in-command corresponds to a message that the assembly can receive, and handle in the eventhandler: "`when take photo from this …`"
- Two *out*-commands: "`position [out]`" for displaying the most recent gps coordinate, and "`image [out]`" for displaying the most recently stuffed image. The out-messages are sent using the syntax "`invoke …`" in the eventhandler.

Synthesized services are useful for presenting an interface to an assembly as a whole, as in this case of the GeoTagger. It allows the GeoTagger to be made part of another assembly.

Synthesized services are also useful for *adapting services* to other interfaces. For example, suppose we would like to replace the current gps in the GeoTagger with another gps with another service interface. One possibility is of course to simply change the GeoTagger assembly description and change the script to handle the new interface. Another possibility is to keep the GeoTagger assembly, and rebind it from the old gps to a new adaptor assembly that makes the new gps look like the old one, by using a synthesized interface. An alternative to using an assembly to build the adaptor is of course to implement an adaptor service, using a general-purpose programming language. But building the adaptor using an assembly is much less work, and can be done by an end user. Of course, if the service interface of the new gps is very different from the old one, general-purpose programming might be needed. For example, if one gps provides its coordinates according to one standard, and the other gps according to another standard. The assembly scripting language is not sufficiently powerful to program such a coordinate translation. An appealing solution in this case is to implement a general coordinate translation service as a discoverable component, and to use an assembly to combine the old gps and the translation service into an adapted service.

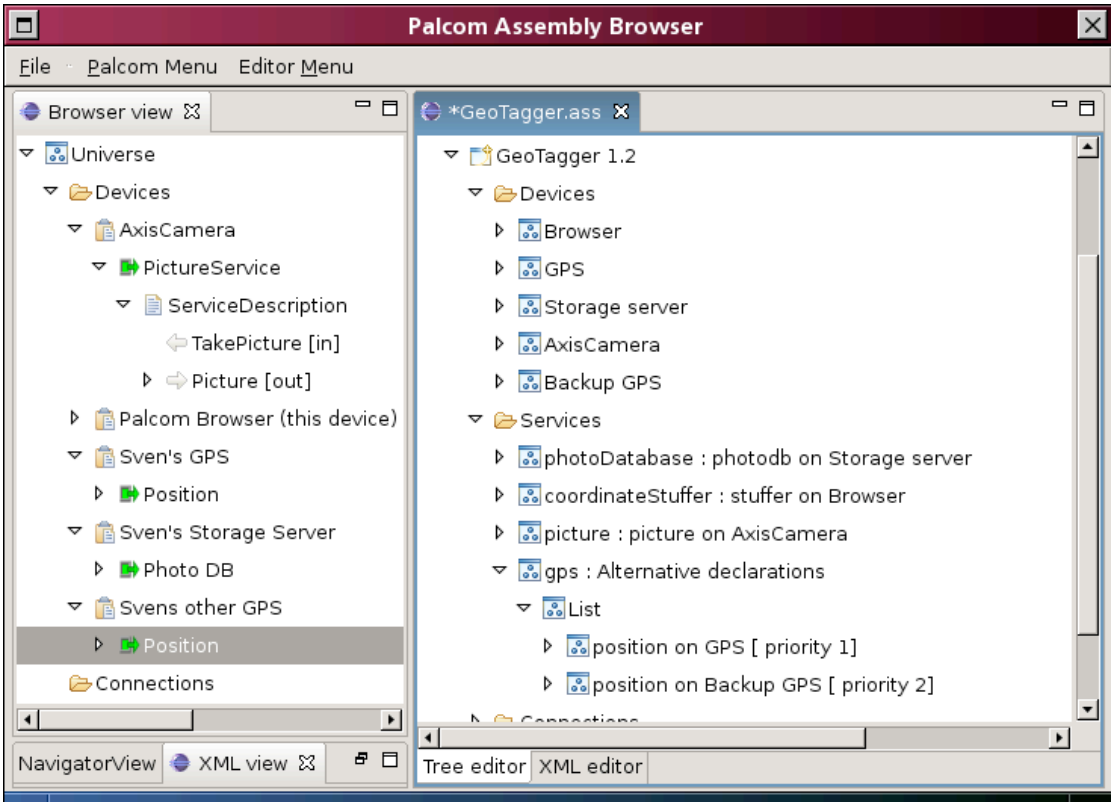
5.5 Initial support for bindings

In [Del25] we identified the need for an assembly to be able to express different kinds of *bindings* to its assembled services, allowing certain variations concerning which actual devices and services to bind to. This area is one where WP6 on End-User Composition interacts with WP5 on Resource and Contingency Management. In order to establish an initial part of the PalCom architecture to support bindings, we have implemented two very simple cases: 1) all assembled services are, by default, optional, and 2) it is possible to specify several alternative services.

Optional services: Each service declared in the assembly is currently, by default, optional. This means that if a particular service listed in the assembly description is somehow not possible to connect to, the rest of the assembly will run according to the script, but there will be no messages sent to/from the missing service.

For example, consider the GeoTagger assembly discussed earlier. If the photo database service is not available, this is made visible in the Assembly Editor by tagging the photo database with the text "`Unknown Device`". But the assembly will still run, and since the other services are available, the images will be tagged with gps information by the coordinate stuffer, they will simply not be sent on to the photo database.

Alternative services: It is possible to declare several alternative actual services, and to specify their priorities. In the figure below, a new version of GeoTagger has been created which has two alternative services for the gps: "`gps: Alternative declarations`". One on the normal device "`GPS [priority 1]`" and one on the backup device "`Backup GPS [priority 2]`". If the normal device is not available, the assembly will automatically connect to the backup device, so called hot-swapping, see [Del44]. See also [Del44] for ongoing work on more advanced ways of selecting services based on resources.



5.6 Experiments with versioning and epidemic updates

In [Del25] the need for *epidemic updates* was identified, in order to support simple updating of assemblies. We have now implemented initial experimental support for such updates. The basic idea is that all services have version numbers and that updated services can be sent automatically over the network whenever devices connect. The version numbers are used both for identifying that two versions are the same or different, and for representing a derivation graph of the versions, allowing merging support between versions when needed.

In the experiment, the assembly manager is packaged as a service, with a subservice Update that provides an interface to the epidemic update functionality. The assembly manager on the browser device is connected to all assembly managers on all other discovered devices. When the user edits and releases a new version of an assembly, the new version is sent to all other connected assembly managers. When one of these assembly managers is later connected to yet other assembly managers, the new versions are passed on, hence the term epidemic updates.

The version of an assembly is shown in the browser view. Several versions of an assembly can coexist and which ones are present is shown in the browser view. While the assembly versions automatically spread to devices, it is under user control to actually start using a new version. This initial implementation of epidemic updates is to be viewed as a proof of concept implementation. Filtering and security mechanisms need to be added to allow users to control the spreading of new versions.

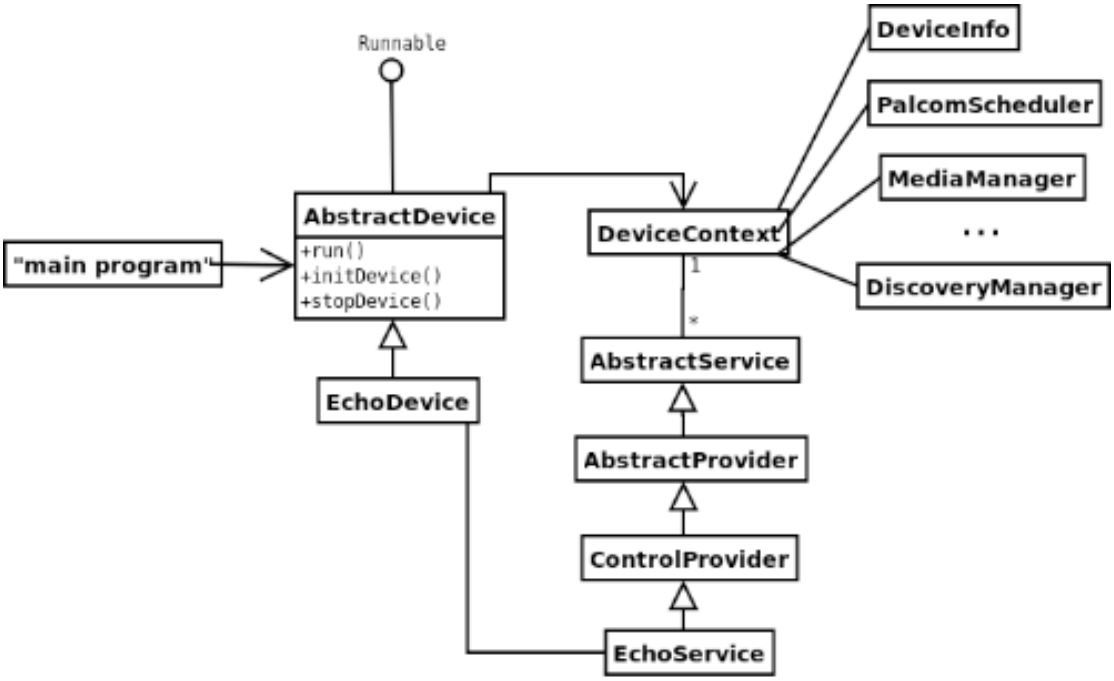
6 Supporting middleware software

The implementation of the Developer's Browser builds on a substantial amount of underlying middleware software. In this section we report on some of this middleware that is of particular importance to the browser. The middleware is developed in Pal-J, and is useful for any device that runs a JVM or a PalVM, and not only for browsing devices.

6.1 Service Framework

In order to easily build services and program devices, we have developed a software package called the Service Framework. It contains an abstract class *AbstractService* that captures what is common to all services, and specialized classes for *providers* and *customers*, and for predefined service types such as *control*. It also contains an abstract class *AbstractDevice* that models the device, and that keeps track of a the device's services and managers in a *DeviceContext* object.

A part of the service framework is shown in the figure below, including a simple application of the framework: the implementation of an EchoDevice that has an EchoService. The EchoService has an in-command for receiving a message with a text parameter, and simply sends a corresponding out-message every time it receives an incoming message. The example is further described in [WN112].



6.2 Initial Resource Manager

The *Resource Manager* is responsible for providing assemblies with resources such as suitable devices, services, and connections. An initial very simple Resource Manager has been implemented that supports the current requirements from the Assembly Manager, including handling of alternative services. This initial Resource Manager will be replaced by a more advanced Resource manager and a Contingency Manager that are being implemented within WP 5, see [Del42]. The current version supplies assemblies with connections, and informs the relevant assemblies when a connection is broken. The Resource Manager is implemented as a subclass of ControlProvider in

the Service Framework, making itself a service that can be accessed via a Remote view or another assembly. By making the Resource Manager a service, smaller devices do not need to have their own resource manager, even if they run assemblies, but can rely on a nearby device with a Resource Manager Service.

6.3 The Assembly Manager

The *Assembly Manager* is responsible for storing, loading and running assemblies. It is of course used by the Developer's Browser, but it can also be used on its own, on a smaller device, e.g., a camera or a gps, provided that the device provides a JVM or a PalVM. To support this latter case, the Assembly Manager is packaged as a service (it is implemented as a subclass to the ControlProvider in the Service Framework). The commands in the Assembly Manager service allows assemblies to be loaded and run, and also to be edited, e.g., to change individual connections. The packaging of the Assembly Manager as a service allows it to be controlled by other services and also to be controlled remotely from other devices. For example, the user can bring up a Remote view of the Assembly Manager of a small device and load and run assemblies on the device.

6.3.1 Representation of assemblies

An assembly description is stored as an XML string, currently in a file in the local file system on the device. This use of files will soon be replaced by storing the XML strings in the *Storage Component* that is currently developed in WP 9 [Del44]. The Storage Component will furthermore be made into a ControlProvider service. This will allow small devices to store their assemblies on other devices. The Storage Component Service will also be useful for assemblies since an assembly can use the Storage service for storing data persistently.

Internally, in the Assembly Manager, the assembly is represented as an *abstract syntax tree,* an *AST*. The abstract grammar of the assembly is shown in appendix A. The main parts of an assembly are the following:

- Devices – references to particular devices
- Services – references to particular services on those devices
- Connections – description of how the services connect to each other and to the assembly.
- Script – description of local state (variables) and of what actions should be taken when the assembly receives a message from one of its connected services (eventhandler).
- Synthesized service – a service description for the assembly, allowing the assembly to be viewed as an ordinary service, e.g., to let another assembly interact with it.

6.3.2 Loading and running assemblies

The assembly manager can be asked to load and run an assembly. When given an XML representation of an assembly, it performs the following actions:

- Parse the XML string into an AST
- Find the relevant services and devices described in the AST (by asking the Resource Manager)
- Set up the connections between the services (also via the Resource Manager)
- Start an interpreter for the assembly script. The interpreter is essentially an event-based loop that waits for messages on the incoming connections and acts according to the script logic.

The Assembly Manager can also be asked to stop and unload the assembly. The interpreter will then be stopped, and the connections closed.

6.4 Discoverable Component Manager

The Discoverable Component Manager is responsible for storing and loading discoverable components, i.e., services that are not tied to the device hardware, and which can therefore be copied between devices. When loading a discoverable component it starts to run as a service on the device. The Discoverable Component Manager is implemented as a service (a subclass to ControlProvider in the Service Framework), with commands for loading a service, and for starting and stopping it. This is used by the Assembly Manager: if an assembly requires a discoverable component that is currently not running, the Assembly Manager will connect to the Discoverable Component Manager and ask it to start the component. Furthermore, the use of the Discoverable Component Manager as a service opens for future support for copying Discoverable Components between devices.

6.5 Simulated devices

In a development situation is it valuable to be able to simulate devices, i.e., to run their services on a general-purpose computer rather than on the device itself. This is useful in order to prototype the functionality that will later be built for real in the actual device: building a simulated device is much less work than building an actual device. In order to support this, we have developed a graphical user interface framework for easily developing the user interface of a simulated device. This has been used for simulating a number of devices, e.g., cameras, gps-devices, etc. When running simulated devices they run as separate processes or threads, typically on a laptop. This allows easy debugging in the development situation, as well as more realistic simulations of scenarios, if the simulated devices are placed on different physical laptops. This piece of supporting software needs full Java to run, and is part of the Utilities layer of the PalCom toolbox infrastructure, see [Del39].

7 Interaction with other workpackages

The work in WP6 has been carried out in close cooperation with the other workpackages on specific design: The middleware software has been implemented in Pal-J, using the tools and software developed in WP3 (Runtime Environment) and WP4 (Communication and Components), and providing feedback and requirements on those tools and libraries. There have been design meetings and discussions with WP5 (Resource and Contingency Management) resulting in a design and implementation of the initial resource manager.

The work in WP6 has also benefited much from the work in the application prototypes workpackages, WP7-WP12. At the WP5-6 workshop held in Lund 6-7/11 there were representatives from all these workpackages. The coordinated work has resulted in both identifying requirements needed for mechanisms in the assemblies, and for experiments with implementing assemblies for partial scenarios. A number of experiments have been made within WP6. Feedback from these experiments is used continually to improve both the concepts and the software. The most extensive experiments so far have been with the GeoTagger and the Tiles scenarios (parts of WP7 and WP11, respectively).

GeoTagger is a part of the OnSite scenario, developed in WP 7. Experiments with programming this scenario started already with the MUI browser for handhelds, and has been the most important driving scenario for development of the current assembly concept. The GeoTagger assembly combines a camera, a gps device and a storage device in order to automatically store gps-tagged images when a landscape architect takes pictures with a camera. This scenario has been successfully implemented and run with the assembly mechanism in the Developer's Browser.

The work on *Tiles* is part of the Active Surfaces scenario developed in WP11 on Care Community. Based on the assembly concept, a solution to the Tiles games was developed in WP11 [Gr+06], and a simulated implementation using the Developer's Browser has been developed as part of WP6 [BrMaSv07]. See also [Del44].

A first internal release of the software was done in November 2006, and besides WP11, other workpackages are now starting to use the software, in particular WP9 for implementing a simulated prototype of the Stone, and WP5 for implementing a more advanced Resource Manager.

8 Conclusion

This deliverable reports on the software support for assemblies. An interactive browser has been implemented which supports browsing of devices and services, remote interaction with services, and support for editing and running assemblies. The main functionality for managing and running assemblies can also be run on smaller devices without an interactive browser.

The PalCom application prototypes have been used for identifying requirements on the assembly mechanism, and major and important parts of these requirements have been implemented and tested for partial scenarios from the prototypes, most notably GeoTagger and Tiles. Important assembly mechanisms that have been implemented include connections that assemble services, scripts with variables and eventhandlers, and synthesized services for assemblies.

The work is still preliminary in that there are many aspects of assemblies that need further development, both concerning the concepts and the software. Yet, the main design is stable and working, and we are in a situation where refined ideas can be implemented and incrementally improve the working tool set.

The technical work reported here can be put into perspective by relating to the PalCom challenges (treated in more depth in [Del39]). It is interesting to see that the work contributes to solving all of these challenges, while of course it is far from providing a complete solution to any of these rich concepts:

An end-user can construct an ad-hoc application by combining services (*construction*). The combination is represented as an assembly description that can be inspected (*visibility*) and taken apart and changed (*deconstruction*). The inner structure of the assembly can also be ignored, and the assembly used as a higher-level service by constructing a synthesized service (*invisibility*). The assembly remembers its connections and automatically connects its assembled services (*stability*). But it is also possible to change the combination of services, e.g., in order to cope with changes in the environment (*change*). The assembly concept is simple to understand (*understandability*), yet has the potential to scale because of its hierarchical nature (*scalability*). Assemblies support combination of heterogeneous devices and services, they only need to support a simple and very general service protocol that is both domain independent and platform independent (*heterogeneity*). At the same time, there is coherence due to the underlying common service protocol (*coherence*). There is certain very limited initial support for autonomy in that by defining alternative service bindings, an assembly can automatically switch an assembled service if the highest priority service fails (*autonomy*). But the limits for this autonomous behavior is completely under user control and explicitly declared in the assembly descriptor (*user control*).

Future work will focus on refining the assembly concept and tools in order to improve the support for the PalCom challenges. In particular, we will implement additional partial scenarios and simultaneously update the assembly language and browser capabilities to support their needs. The tools and supporting middleware will be provided as open-source software. Work will also continue on visual browsers for ordinary end users and small browsers for devices with limited interaction capabilities.

9 References

[Del25] PalCom External Report 34: Deliverable 25 (2.6.1): *End-user Composition Tool*. Technical report, PalCom Project IST-002057, October 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-25-[2.6.1]-end-user-composition-tool.pdf

[Del39] PalCom External Report 50: Deliverable 39 (2.2.2): Open architecture. Technical report, PalCom Project IST-002057, December 2006. http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-39-[2.2.2]-open-architecture.pdf

[Del41] PalCom External Report 55: Deliverable 41 (2.4.3): Components & communication. Technical report, PalCom Project IST-002057, December 2006. http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-41-[2.4.3]-components-communication.pdf

[Del44] PalCom External Report 58: Deliverable 44 (2.7.2): Prototypes status after Year 3. Technical report, PalCom Project IST-002057, January 2007. http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-44-[2.7.2]-prototype-status-after-year3.pdf

[WN112] Palpable Working Note #112. Service framework. David Svensson, January 2007.

[WN113] Palcom Working Note #113. Eclipse-based browser and assembly editor: User's guide. Sven Gestegård Robertz, January 2007.

[Eclipse] www.eclipse.org

[BrMaSv07] Jeppe Brønsted, Boris Magnusson, David Svensson, The Tiles Simulator, Dept. of Computer Science, Aarhus University, 2007, In Preparation.

[Gr+06] Erik Grönvall, Alessandro Pollini, Alessia Rullo, and David Svensson. Designing game logics for dynamic Active Surfaces. In Proceedings of MUIA '06, Workshop on Mobile & Ubiquitous Information Access, Espoo, Finland, September 2006.

[MaJa07] Boel Mattsson and Brice Jaglin. Implementing the PalCom protocol in an Axis network camera. Master's thesis, Dept. of Computer Science, Lund University, January 2007.

[SvMaHe05] David Svensson, Boris Magnusson, and Görel 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, pages 153-164, Erfurt, Germany, September 2005

[SvHeMa06] David Svensson, Görel Hedin, and Boris Magnusson. Pervasive applications through scripted assemblies of services. In Proceedings of SEPS 2006, 1st International Workshop on Software Engineering of Pervasive Services, Lyon, France, June 2006.

[Sv06] David Svensson, Support for Ad-Hoc Applications in Ubiquitous Computing. Licentiate thesis, Lund University, November 2006.

10 Appendix. Abstract grammar for assemblies

```
abstract InfoRoot : Info ::= <Format:String> <Name:String> <Version:String>;

abstract Decl : AbstractXMLRepresentable ::=;

//Add device ID and time stamp here
AssemblyDescriptor: InfoRoot ::= AssemblyInfo*;

AssemblyInfo: InfoRoot ::= <Released:boolean> Devices:DeviceDeclList
               Services:ServiceDeclList Connections:ConnectionDeclList
               [EventHandlerScript] [ServiceDescription];

DeviceDeclList    : Decl ::= DeviceDecl*;
ServiceDeclList   : Decl ::= ServiceDecl*;
ConnectionDeclList: Decl ::= ConnectionDecl*;

DeviceDecl : Decl ::= Name:Identifier URN;

ServiceDecl : Decl ::= LocalName:Identifier Decl:AbstractServiceDecl;
abstract AbstractServiceDecl : Decl ::= ;
SingleServiceDecl : AbstractServiceDecl ::=
       ServiceName:Identifier DeviceUse <URNSuffix:String>;
AltServiceDeclList : AbstractServiceDecl ::= ServiceDecl:AltServiceDecl*;
AltServiceDecl : SingleServiceDecl ::= <Prio:String>;

ConnectionDecl : Decl ::= Provider:ServiceExp Customer:ServiceExp;

Identifier : Decl ::= <ID:String>;

DeviceUse : Decl ::= Identifier;

abstract ServiceExp : Decl;

ServiceUse: ServiceExp ::= Identifier;
ThisService : ServiceExp;

EventHandlerClause*;
EventHandlerScript : Decl ::= Variables:VariableList EventHandlers:EventHandlerList;

VariableList:Decl ::= VariableDecl*;
EventHandlerList:Decl ::= EventHandlerClause*;


VariableDecl : Decl ::= VariableType Identifier;
abstract VariableType;
MimeType: VariableType ::= <TypeName:String>;

EventHandlerClause : Decl ::= <CommandName:String> ServiceExp [CommandInfo] Action*;

abstract Action : Decl ::= ;
AssignAction : Action ::= VariableUse ParamUse;
abstract ActionWithParams : Action ::= <Command:String> ParamValue:Use*;
SendMessageAction : ActionWithParams ::= ServiceExp;
InvokeAction : ActionWithParams ::= ;

abstract Use : Decl ::= ;
VariableUse : Use ::= <Name:String>;
ParamUse  : Use  ::= <Name:String>;
ConstantUse : Use ::= <Constant:String>;
MissingUse : Use ;
```