Date of acceptance Grade

Instructor

Service Composition on a Mobile Phone

Ville Mäntysaari

Helsinki November 26, 2007

M. Sc. Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Ville Mäntysaari

Työn nimi — Arbetets titel — Title

Service Composition on a Mobile Phone

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M. Sc. Thesis	November 26, 2007	61 pages

Tiivistelmä — Referat — Abstract

With the recent increase in interest in service-oriented architectures (SOA) and Web services, developing applications with the Web services paradigm has become feasible. Web services are self-describing, platform-independent computational elements. New applications can be assembled from a set of previously created Web services, which are composed together to make a service that uses its components to perform a certain task. This is the idea of service composition.

To bring service composition to a mobile phone, I have created Interactive Service Composer for mobile phones. With Interactive Service Composer, the user is able to build service compositions on his mobile phone, consisting of Web services or services that are available from the mobile phone itself. The service compositions are reusable and can be saved in the phone's memory. Previously saved compositions can also be used in new compositions.

While developing applications for mobile phones has been possible for some time, the usability of the solutions is not the same as when developing for desktop computers. When developing for mobile phones, the developer has to more carefully consider the decisions he is going to make with the program he is developing. With the lack of processing power and memory, the applications cannot function as well as on desktop PCs. On the other hand, this does not remove the appeal of developing applications for mobile devices.

ACM Computing Classification System (CCS):
C.2.4 [Distributed Systems]
D.2.12 [Interoperability]

Avainsanat — Nyckelord — Keywords
web services, composition, orchestration, mobile phone

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpula Science Library, serial number C-

Muita tietoja — övriga uppgifter — Additional information

Acknowledgements

When starting to write this thesis in January 2007 it didn't feel like I would ever be able to write the acknowledgements, it felt like a thing in the far future. But now after nine months it's starting to feel that soon I'm able to say this is done.

I wish to thank HIIT and especially Ken Rimey and Kimmo Raatikainen for the job opportunity, this thesis was made during the S4ALL project at HIIT. Big thank you for Ken for the comments and suggestions. I also want to give big credit to Pekka Kanerva for all the work he did with me for Composer and for the proxy. Also I want to thank Tero Hasu for the work he did for Composer. Also a thank you for Capricode for the cooperation is in place, namely Kimmo Kuusipalo, Harri Salokorpi and Maija Metso.

I also want to give a big thank you to Esa Pitkänen on the valuable comments he gave on this thesis.

Last but not certainly least. Thank you Kaisa, for everything.

I tried my best to follow Ken's advice on 'removing anything that seems to say nothing', but I'm not sure if I really was able to do it. This following quote is one of my favourite quotes, it has the same idea Ken tried to tell me. It's not about the length but the contents.

> La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever.
> -Antoine de Saint-Exupéry

In Helsinki, November 26, 2007
Ville Mäntysaari

Contents

1 Introduction

With the recent increase in interest in service-oriented architectures (SOA) and Web services, developing applications with the Web services paradigm has become more and more accepted. With the basic Web services paradigm developers are able to invoke Web services made by other developers or companies. Although this might be enough for simple development, more complex technologies are needed to build even more complex applications in the Internet.

To pursue this further technologies for Web service composition have been developed. In Web service composition applications can be assembled from a set of previously created Web services available in different service registries. These services together compose an application, which reaches its goal by invoking each individual service. Web service composition is a simple way to master complexity, by using elementary services as building blocks the developer does not need to worry how the the blocks are made or what programming languages are used to make them. This helps the developer to concentrate on different matters, if the building blocks are well defined and documented.

While application development for desktop PCs has been commonplace for a long time, mobile phones have been gaining more and more processing power lately, making them an interesting platform for new business scenarios and application development. In the research project S4ALL it has been envisioned to being able to compose Web services using mobile devices. Web service composition paradigm brings a lot of new possibilities for the devices. For this vision, I have created the Interactive Service Composer for Mobile phones which will I will describe in this thesis.

With technologies like BPEL [ES06] (Business Process Execution Language) and WS-CDL [RTF06] (Web Service Choreography Definition Language) developers are able to build Web service compositions. But these technologies are not viable for a mobile environment, where the processing power and the amount of memory available are low. These technologies are made to be used in environments, where the processing power is not a problem. On the other hand with Interactive Service Composer I am showing how we are able to do simple service composition without requiring for example the mobile device to transfer the composition to another machine for running. Admittedly Interactive Service Composer is not able to use the most sophisticated methods, but this doesn't reduce its abilities to perform

composition.

Interactive Service Composer is one of the many systems able to conduct service composition. Other interesting systems are eFlow [C+00] and SELF-SERV [BSD03]. What makes them interesting are the fact that both composition types, namely orchestration and choreography, are represented in these two. In orchestration there is a central entity that controls the execution of the service composition. This is different in choreography, where entities that take part in the composition are communicating together to execute the task. SELF-SERV is a system that uses choreography as a way to compose services, where services are collaborating together to implement a specific operation without a central entity. In eFlow services are composed as an orchestration, where a central entity controls the execution of the composition.

Interactive Service Composer (Composer in short) is a simple orchestrating service composer made with Python for mobile phones. The main subject of this thesis is to describe the work made for Composer. Before going too deep on the details of Composer I will present some background regarding Web services and service composition. Web services paradigm allows developers to find new services and access them. With service composition it is possible to build new services that include elementary services as building blocks. This composition then can be used as a normal Web service.

Composer has three distinct parts, the action store that is responsible for the actions (the building blocks) that are saved in the phone's memory, the workflow structure that is the base for building new service compositions and the user interface that keeps the user informed of the current state of the program. With Composer users are able to build new workflows and use them over and over again by saving the workflow. What allows more reuse is the possibility of using previously saved workflows as building blocks in a new workflow.

Simplicity, re-usability and generality (of the actions) were the main points considered when Composer was planned. Users should be able to make new services easily without requiring any knowledge of the implementation details. More advanced users were planned to be able to make actions of their own by coding a simple Python program which conforms with the format that the actions need to have. Generality in terms of actions was a tempting characteristic to have. The initial actions were planned to be usable in many different places so that users would have some general tools available to them before building their own workflows. Re-

usability was also a very much considered point. Existing workflows should be usable as part of new workflows and the saving and loading of workflows were considered to be important.

During the research project Composer was made in, it was used in a demo. In the demo Composer is used to execute device management tasks. These tasks are conducted by using a SOAP API that allows Composer to create device management tasks. This brought good experiences on how Composer could be made better by continuing the development. On the other hand good experience was not the only thing received from the demo. In addition it was noticed that some of the decisions made earlier in the development were not sufficient to tackle the problems we ran into in the demo.

What software development is for desktop PCs, is not the same as developing for mobile phones. Although with the recent increase in the processing power of the mobile devices and the increase of mobile platforms, there is a need for a mobile platform that is easy to develop for. Besides that the lack of processing power makes developing harder when additionally the developer needs to consider the problem of having small amount of memory and a slower processor than the desktop PCs have. Python for Series 60 is a promising environment to develop software. What makes it especially interesting is the simplicity and rapidity of developing for it.

The rest of this thesis is organised as follows: In chapter 2 I will cover some background of Web services and the different Web service technologies that are used in Interactive Service Composer. Web service composition and service-oriented architecture will have their own subsections. Chapter 3 is devoted to introducing the Interactive Service Composer and how it is made, what kind of decisions we had to make when developing it.

In chapter 4 I will cover some insights received from using the Interactive Service Composer in a demo. Interactive Service Composer was used to invoke device management tasks. Chapter 5 is about the experience gained from developing for mobile phones. Mobile phones having limited resources and processing power gives an extra consideration when developing for mobile phones. The libraries needed in the process are usually made for desktop PCs which makes the libraries unusable in the mobile device environment.

2 Background

The Web started as a technology for sharing information on the Internet. However, it quickly became the medium for connecting remote clients with applications across the Internet and more recently (with the arrival of Web services) a medium for integrating applications over the Internet, built on top of existing web protocols and based on open XML standards.

The growing trend in the industry is to build platform-independent software components, called *Web services*, which are available in the Internet. New applications can be assembled from a set of appropriate Web services and no longer must be written from scratch. Seamless composition of Web services has enormous potential in streamlining business-to-business or enterprise application integration [SK03]. The term Web services is very widely used nowadays, although not always with the same meaning. World Wide Web consortium (W3C) defines Web services as

> a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols. [A+03]

The W3C definition is quite accurate and also hints at how Web services should work. The definition stresses that Web services should be capable of being defined, described, and discovered thereby clarifying the meaning of accessible and making more concrete the notion of Internet-oriented, standards-based interfaces. It also states that Web services should be services similar to those in conventional middleware. Not only they should be up and running, but they should be described and advertised so that it is possible to write clients that bind and interact with them. In other words Web services are components that can be integrated into more complex distributed applications.

The W3C also states that XML (eXtensible Markup Language) [B+06] is part of the solution. In fact XML is so popular and widely used today that, just like HTTP and Web servers, it can be considered as being part of the web technology. It is likely that XML will be used as a data format for many web-based applications in the future.

There are two types of services: simple and composite services. The unit of reuse is

functionality that is in place and readily available and deployable as services that are capable of being managed to achieve the required level of service quality. Composite services involve assembling existing services that access and combine information and functions from possibly multiple service providers. More about the issue of service composition can be found in chapter 2.3.

2.1 Web Service Technologies

Leading standards when talking about Web services are *SOAP* [G+06] (Simple Object Access Protocol), *WSDL* [BL06b] (Web Service Description Language) and *UDDI* [OAS05] (Universal Description Discovery & Integration). Many applications today are made accessible to other applications using these three standards. These standards are so common that usually when talking about Web services these are presumed to be used [TBB03]. There are also other standards that can be used with Web services, namely *REST* [Fie00] (Representational State Transfer), which is in fact an architectural style and, *JSON* [Rub07] (Javascript Simple Object Notation), to name a few. However, these standards do not constitute the essence of Web services technology: the problems underlying Web services are the same regardless of the standards used.

2.1.1 SOAP

SOAP (Simple Object Access Protocol) is an XML-based communication protocol [G+06, A+04, C+02]. SOAP defines how information is organised using XML in a structured and typed manner so that the data can be exchanged between peers. As a communication protocol, SOAP is stateless and one-way. This means that SOAP is created by design to support loosely-coupled applications that interact by exchanging one-way asynchronous messages with each other. Any further complexity in the communication pattern such as two-way synchronous messaging or RPC-style interaction requires SOAP to be combined with an underlying protocol or middleware. Rather than defining a new transport protocol, SOAP works on existing transport protocols, such as *HTTP* (Hypertext Transfer Protocol) and *SMTP* (Simple Mail Transport Protocol). SOAP uses *XML Schema* [FW04] to define the document types.

Information exchange in SOAP is done using messages. The messages are used as an envelope where the application encloses whatever information needs to be sent. The

Listing 1: An example SOAP Message (taken from [C+02])

```
<SOAP:Envelope xmlns:SOAP=
  "http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    <et:eTicket xmlns:et=
    "http://www.acme-travel.com/eticket/schema">
      <et:passengerName first="Joe" last="Smith"/>
      <et:flightInfo airlineName="AA"
                     flightNumber="IIII"
                     departureDate="2002-01-01"
                     departureTime="1905"/>
    </et:eTicket>
  </SOAP:Body>
</SOAP:Envelope>
```

message format describes how information is packaged into an XML document. The envelope contains two parts: a header and a body. The header contains information which can be processed by intermediate nodes (they might be able to supply some added value services for the sender and receiver) between the sender and the receiver. If there are no intermediate nodes in a SOAP transmission, the header might not be necessary at all. That is why it is defined to be optional. The message body is the content the sender wants to send to the receiver. This follows the idea of the standard communication protocol approach. An example SOAP message is shown in listing 1, taken from [C+02]. In the example the optional SOAP header is not present and the message only includes the required SOAP body. This message includes data about a flight booking, details being the name of the booker and the date and time of the booked flight. A real example would also include the sender's credentials and other required information.

2.1.2 REST

Although it is often mistaken for a transport protocol, HTTP is really an application protocol. SOAP can be transported over HTTP for example, an act known as tunneling. In reality HTTP is much more than just transporter of bytes. In his doctoral dissertation, Roy Fielding [Fie00] introduced the term representational state transfer, i.e. REST, to describe the Web's architectural style. REST uses standard-

ised interfaces to promote stateless interactions by transferring representations of resources, rather than operating directly on those resources [Vin02a].

HTTP provides application-level semantics via its "verbs": GET, POST, PUT and DELETE. These verbs form a generic application interface that can be applied in practise to a broad range of distributed applications despite the fact that it was originally designed for hypermedia systems. This interaction model that REST defines is suitable for Web services as well. Web services are identifiable via URIs, and regardless of the wide variety of abstractions they might collectively represent, they can all be implemented using the same generic interface that HTTP's verbs provide [Vin02a].

REST style Web services are more lightweight to use, additional libraries are not needed as basic HTTP communication is usually available in every programming language by default. When developing applications with a mobile phone it is always easier to use services that do not require big libraries, because of the low computing power and low memory availability. SOAP might be the de-facto standard when talking about Web services, but when one is developing a lightweight software on a mobile phone it is not very usable. I will describe more experiences with developing on a mobile phone in chapter 5.

2.1.3 WSDL

For Web services, SOAP is usually used in communication between the services, but SOAP is not able to supply the information what messages need to be exchanged to interact with a certain service. WSDL (Web Services Description Language) is an XML format language developed by IBM, Microsoft and Ariba [BL06b, A+04, C+02]. It was made to describe Web services as end points that can exchange certain messages. WSDL describes what messages need to be exchanged to use the service, also it describes where the service is located and what vocabulary is used in the messages. Vocabulary here is referring to the datatypes and message formats used in the Web service. WSDL uses external type systems to provide datatype definitions for the information exchange. Most services use XML Schema, but in general any type system can be used. XML Schema has built-in basic data types and it allows users to define more complex data types such as structures.

A complete WSDL service description provides two pieces of information: an application-level service description, in other words an abstract interface, and the specific

protocol-dependent details that users must follow to access the service at concrete service end points. The abstract part is made of port type definitions, where each port type is a logical collection of related operations. Each operation defines a simple exchange of messages. These are then used in the concrete part of the description. Message encoding and protocol bindings for all operations are specified within *InterfaceBindings*-element. These InterfaceBindings are then used in *Port*-definition, where ports combine the InterfaceBinding information to a network address, specified by a URI. And again Port is then used in Service-definition. This definition is the logical grouping of ports. An example WSDL description is shown in listing 2, taken from [C+02]. The example has all the necessary elements included. The abstract part is the message and port type definitions at the start. The concrete part is the binding and service definitions in the end of the example.

Listing 2: An example WSDL description (taken from [C+02])

```
<?xml version="1.0"?>
<definitions name="Procurement"
    targetNamespace="http://example.com/procurement/definitions"
    xmlns:tns="http://example.com/procurement/definitions" ... >
    <message name="OrderMsg">
       <part name="productName" type="xs:string"/>
       <part name="quantity" type="xs:integer"/>
    </message>
    <portType name="procurementPortType">
       <operation name="orderGoods">
          <input message = "OrderMsg"/>
       </operation>
    </portType>
    <binding name="ProcurementSoapBinding"
             type="tns:procurementPortType">
       <soap:binding style="document"
             transport="http://schemas.xmlsoap.org/soap/http"/>
       <operation name="orderGoods">
         <soap:operation
         soapAction="http://example.com/orderGoods"/>
         <input>
            <soap:body use="literal"/>
         </input>
         <output>
```

```
            <soap:body use="literal"/>
          </output>
        </operation>
    </binding>
    <service name="ProcurementService">
        <port name="ProcurementPort"
            binding="tns:ProcurementSoapBinding">
            <soap:address
            location="http://example.com/procurement/"/>
        </port>
    </service>
</definitions>
```

The difference relative to normal middleware platforms is the need to define the location at which the service is available [A+04]. In conventional middleware, the service provider can simply implement an interface and register the implemented service with the middleware. The absence of centralised platform in Web services means that the client should be able to identify the location at which the service is made available. This problem can be addressed for example using centralised registries, like UDDI registries.

2.1.4 UDDI

The UDDI (Universal Description Discovery & Integration) specifications [OAS05, C+02, A+04] offer users a unified and systematic way to find service providers through a centralised registry of services that is roughly equivalent to an automated online phone directory of Web services. Accessing the registry is done using a standard SOAP API for both query and update operations. UDDI provides two basic specifications that define a service registry's structure and operation. First one is the definition of the information to provide about each service and how it is encoded. The second one is the API for querying and updating the registry, which describes how this information can be accessed and updated. Being about Web services, UDDI APIs are also specified in WSDL with SOAP binding, so that the registry can itself be accessed as a Web service and also its characteristics can be described in the registry itself, just like any other Web service.

The information in UDDI registries can be simply categorised by the way what each type of information is used for. This categorisation is analogous to a telephone

directory. The white pages of the registry provide a listing of organisations, their contact information and the listing of the services these organisations have. In the yellow pages there are classifications of both companies and Web services according to systematics that can be either standardised or user-defined. It is possible to search through the yellow pages for a service that belongs to a certain category. The green pages of the registry provide information on how a given Web service can be invoked. It is provided by means of pointers to service description documents, typically stored outside the registry, for example at the service provider's site.

There is a free UDDI registry available on the Internet provided by XMethods [XMe07]. The site provides both programmable interface to the services the registry has listed (using UDDI version 2 specification) or one can search for services using their web based interface. XMethods' registry is the registry that is used in our service composer when a user wants to search for new Web services. XMethods also has an interface for testing the Web services with a web browser, using MindReef's SOAPScope service [Min06]. There are a multitude of services available at XMethods' registry, both free and commercial services.

2.2 Service Oriented Architecture

Although the Web was intended from the start to be used by humans, most people have agreed that it will have to evolve - for example through the design and deployment of modular *services* - to have a better support for automated usage. When talking about Service-Oriented Computing (SOC) [HS05, BL06a, PG03], services are used as the fundamental element while developing new applications. These services are platform- and network-independent operations that clients or other services invoke, as they were described in the previous chapter. Since services can be offered by different organisations and since they communicate over the Internet, they provide a distributed computing infrastructure for both intra- and cross-enterprise application integration and collaboration. SOC concept has evolved from earlier component-based software frameworks such as J2EE, CORBA, and DCOM. Web services standards - including SOAP, WSDL, UDDI and BPEL - are based on the readily and openly available Internet protocols XML and HTTP, and thus are easier and cheaper to adopt [BL06a].

In order to build the service model, SOC relies on the Service-Oriented Architecture (SOA) [A$^+$04, Pap03]. In SOA a set of previously created software applications and support infrastructure are organised into an interconnected set of services, each ac-

cessible through standard interfaces and messaging protocols. When all the pieces of an enterprise architecture are in place, existing and future applications can access these services as necessary without the need of complicated point-to-point solutions based on proprietary protocols. This architectural approach is particularly applicable when multiple applications running on varied technologies and platforms need to communicate with each other. SOA represents the big picture of what you can do with Web services.

A good explanation of the SOA concept can be found in [Pap03], see also figure 1. SOA is defined there as being

> [...] a logical way of designing a software system to provide services to either end-user applications or other services distributed in a network. SOA defines an interaction between software agents as an exchange of messages between service requesters (clients) and service providers. Clients are software agents that request the execution of a service. Providers are software agents that provide the service. Agents can be simultaneously both service clients and providers. Providers are responsible for publishing a description of the services they provide. Clients must be able to find the descriptions of the services they require and must be able to bind (in other words to be able to use the service) to them. [Pap03]

SOA is not only a service architecture, it is in general a relationship between three kinds of participants: the service provider, the service discovery agency, and the service requester (client). The interactions involve the publish, find and bind operations, as described in figure 1. These participants and operations act upon the service artifacts (the representations of a service): the service description and the service implementation.

As a consequence of the dynamic binding capability there is a loose coupling model between the applications. Loose coupling means that the requester has no knowledge (or does not require any knowledge) of any internal structures or conventions the service might have, for example programming language or deployment platform. In loosely coupled systems, interactions between parties take place via messages in an asynchronous environment with messages exchanged over separated communication sessions. This idea reminds of the late binding [JM76] in programming theory. Loose coupling allows software on each side of the conversation to change without impacting the other, provided that the message schema stays the same. Loose

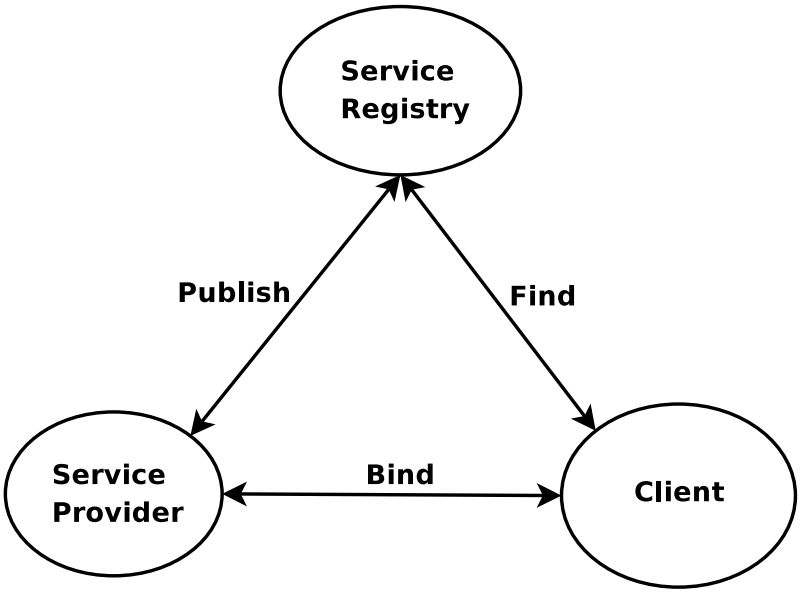

Figure 1: Service-Oriented Architecture

coupling is often cited [MJS06] as a defining and desirable characteristic of service-oriented computing as it keeps the interacting parties independent.

2.3 Service Composition

To move further from the basic Web service architecture (describe, publish, interact) mechanisms for *service composition* are required to be applied. The composition of Web services to handle more complex problems is gaining a lot of attention in enabling business-to-business collaborations [BSD03]. New applications can be assembled from a set of appropriate Web services and no longer written from the scratch. Service composition can be seen as a way to master complexity, where complex services are incrementally built out of services at a lower abstraction level. Web service composition has a lot of potential to streamline the development of new applications and to make enterprise application integration easier. Service composition is not a new idea at all. It has been proposed before and is in fact what *EAI* (Enterprise Application Integration) [Lin00] and workflows are about. The main big difference between workflows and service composition is the fact that the actions are Web service invocations, not normal application operations.

In service composition new services are made from existing services composing them

into a single service, which takes care of the required invocations for the services which it is composed of. Several standards and specifications have been proposed in this area, including Business Process Execution Language for Web Services [AC+03] (BPEL for short), proposed by IBM, Microsoft and BEA, and Web Services Choreography Description Language [RTF06] (WS-CDL) proposed by W3C.

BPEL is a language which is used to build compositions in the form of business processes, which are composed of Web services. BPEL is essentially a layer on top of WSDL, with WSDL defining the operations allowed and BPEL defining how the operations can be sequenced. In BPEL there is a single entity coordinating all Web service invocations. This type of composition is called *orchestration*. WS-CDL is a language for specifying peer-to-peer protocols where each party wishes to remain autonomous and in which no party is a master over another, such that there is no central entity. Each involved party can describe its part in the interaction by itself. This type of composition is called *choreography*. In choreography messages sequences are tracked between the entities in order to follow the execution of the composition.

2.3.1 BPEL

BPEL (Business Process Execution Language for Web Services) [ES06, AC+03, Pel03] supports a process-oriented form of service composition. Each BPEL composition is a business process or workflow that interacts with a set of Web services to achieve a certain task. BPEL is simply a flow language that combines together basic and structured activities to create the logic of a business process. BPEL compositions are called processes, the services the process interacts with are called partners and the message exchange or intermediate result transformation is called an activity. With these terms a process contains a set of activities. A process, like any other Web service, supports a set of WSDL interfaces that enable it to exchange messages with its partners. BPEL brings the notion of two-level programming to Web services: programming in the small for implementing the basic services used by a composite service itself and programming in the large for specifying the composite service. Programming in the small is done using the usual programming languages, for example Java and C#. Programming in the large is done based on a business process language, e.g. BPEL. BPEL's development came out of the notion that programming in the large [DK75] and programming in the small required different type of languages.

The process interacts with partners by invoking the operations they support and receiving messages through the process service interface. BPEL also includes activities that allows it to perform actions such as signalling faults, terminating the process execution and manipulating data. These activities can be combined into complex algorithms. These are for example the ability to define an ordered sequence of steps and to define a loop. These structured activities are derived from a combination of several activities, either basic or other structured activities.

BPEL supports both executable and abstract business processes. An executable process describes the participants behaviour in a particular business interaction, in fact describing a private workflow. An abstract process, also called a business protocol, specifies the public messages exchanged between the participants. Business protocols are not executable and do not carry a process flow's internal details [Pel03]. In fact these process types describe the two different service composition methods: executable processes describe orchestration and abstract processes describe the choreography of services.

An example BPEL process is shown in listing 3, taken from [Kha02]. The example is about a loan approving process. The process has two parties involved, a customer and a financial institution, both defined under the <partners> -tag.

Listing 3: An example BPEL process (taken from [Kha02])

```
<process name="loanApprovalProcess"
   targetNamespace="http://acme.com/simpleloanprocessing"
   xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
   xmlns:lns="http://loans.org/wsdl/loan-approval"
   xmlns:loandef="http://tempuri.org/services/loandefinitions"
   xmlns:apns="http://tempuri.org/services/loanapprover">
   <partners>
      <partner name="customer"
         serviceLinkType="lns:loanApproveLinkType"
         myRole="approver"/>
      <partner name="approver"
         serviceLinkType="lns:loanApprovalLinkType"
         partnerRole="approver"/>
   </partners>
   <containers>
      <container name="request"
         messageType="loandef:CreditInformationMessage"/>
```

```
    <container name="approvalInfo"
      messageType="apns:approvalMessage"/>
  </containers>
  <sequence>
    <receive name="receive1" partner="customer"
      portType="apns:loanApprovalPT"
      operation="approve" container="request"
      createInstance="yes">
    </receive>
    <invoke name="invokeapprover"
      partner="approver"
      portType="apns:loanApprovalPT"
      operation="approve"
      inputContainer="request"
      outputContainer="approvalInfo">
    </invoke>
    <reply name="reply"
      partner="customer"
      portType="apns:loanApprovalPT"
      operation="approve" container="approvalInfo">
    </reply>
  </sequence>
</process>
```

2.3.2 WS-CDL

WS-CDL (Web Services Choreography Description Language) [RTF06, MH05] is an XML-based language that can be used to describe the behaviour of multiple services that need to interact in order to achieve some goal, the interaction between the services and the accepted rules that need to be satisfied in the interaction. WS-CDL describes this behaviour from a global or neutral perspective rather than from one perspective of any one party. The complete WS-CDL description is called a global model. The term common collaborative observable behaviour is used in WS-CDL to describe the behaviour of system of services, from a global perspective. Each service has a behaviour of its own which can be described with WSDL for example.

Individual service behaviours can be used in a composition in which a set of services with their own behaviours could be effectively used. In order to do so a global model

that describes the peer to peer interactions of such a set of services is required to ensure that the services will in fact cooperate to a commonly understood script. That script is the global model and that script is what WS-CDL is used to describe. An example of a WS-CDL description is shown in listing 4, taken from [MH05]. The example is thoroughly explained in the article, interested readers can refer to it for explanations. Although the example is fairly long already, I have left out some parts of the description which denoted by [...]'s. Details available in the referenced article.

The WS-CDL choreography description is contained in a package (as shown in the example) and it is a container for a collection of activities that may be performed by one or more participants. The three main types of activities that are defined in WS-CDL are called control flow activity, work unit activity and basic activity. The activities in control flow are sequence, parallel and choice. A work unit activity describes the conditional and repeated execution of an activity. Sequence, parallel, choice and work unit activity of WS-CDL represent the basic control flow structure such as sequence, while and switch in a typical programming language. The third type of WS-CDL activity is the basic activity. Basic activities in WS-CDL are for example interaction, no action or assign.

The information sent or received during an interaction is described by a named variable. Variables in WS-CDL are used to represent three different types of information: application-dependent information (for example product code), state information (for example order sent) and channel information. Variables contain values and have an information type. These variables are accessed using WS-CDL XPath 1.0 extension functions.

Listing 4: An example WS-CDL description (taken from [MH05])

```
<package name ="AnnualStatementService" ...>
<informationType name="correlationId" type="string"/>
<informationType name="annualStatement"
                 type="annualStatement.xsd"/>
 <roleType name="ServiceProviderRole">
  <behavior name="ReceiveAnnualStatement"
          interface="TaxAdvisor.wsdl"/>
 </roleType>
 <roleType name="ServiceRequesterRole">
  <behavior name="ReceiveTaxAssessment"
            interface="TaxAdvisor.wsdl"/>
 </roleType>
 <relationshipType name ="ClientTaxAdvisor">
   <role type="ClientRole" />
   <role type="ServiceProviderRole"/>
 </relationshipType >
   [...]
 <participantType name ="TaxAdvisor">
  <role type="ServiceProviderRole"/>
  <role type="ServiceRequesterRole"/>
 </participantType >
   [...]
 <channelType
  name="SubmitAnnualStatementChannel"
```

```
   action="request">
   <passing action= "respond"
            channel = "ReturnProcessedTaxAssessmentChannel"/>
    <reference>
     <token name ="taxAdvisorRef"/>
    </reference>
    <identity>
     <token name ="processId"/>
    </identity>
  </channelType>
    [...]

<choreography name ="AnnualStatementSubmission"
               root="true">
  <relationship type ="tns:ClientTaxAdvisor"/>
  <relationship type ="tns:TaxAdvisorMunicipality"/>
   [...]
  <variableDefinitions >
   <variable name = "AS"
             mutable = "true"
             free= "false"
             informationType = "annualStatement"
             silent= "false"/>
             roleTypes ="Client,TaxAdvisor"
   [...]
  </variableDefinitions >
  <sequence>
   <interaction name ="AnnualStatementSubmission"
                channelVariable ="tns:SubmitAnnualStatementChannel"
                operation ="ReceiveAnnualStatement" initiate="true">
    <participate relationshipType ="ClientTaxAdvisor"
                 fromRole="tns:ClientRole"
                 toRole="ServiceProviderRole"/>
    <exchange name = "AnnualStatementSubmissionExchange"
              action= "request"
              informationType = "annualStatement">
     <send variable = "AS"/>
     <receive variable = "AS"/>
    </exchange>
   </interaction >
   [...]
  </sequence >
</choreography >
</package>
```

2.3.3 Related Work

There is a lot of related work in the field of service composition. Here I will present two different service composition systems, that represent a choreography approach and a orchestration approach. The Self-Serv environment is building compositions with choreography approach. It is developed by a research project in University of New South Wales and Queensland University of Technology. eFlow is a platform that is building compositions using orchestration approach. It is developed by HP.

Self-Serv

Self-Serv [BSD03, SBDM02] aims to enable the declarative composition of new services from existing ones, the multi-attribute dynamic selection of services within

a composition and peer-to-peer orchestration of composite service executions. The Self-Serv architecture features a service manager and a pool of services. In Self-Serv a composite service is an umbrella structure that brings together other composite and elementary services that collaborate to implement a set of operations. Elementary services provide access to Internet-based applications. In contrast composite services are made of multiple component services. The system expresses the business logic of a composite service operation as a state chart that encodes a flow of invocations to component service operations. A state chart is made up of states, which can be either basic or compound, and transitions, which are labelled according to a set of rules.

In order to support scalable execution of composite services over the Internet, services should be self-orchestrating: they should be capable of executing composite services without relying on a central scheduler. Accordingly, Self-Serv adopts an orchestration model based on peer-to-peer interactions between software components hosted by the providers participating in the composition. The execution of a composite service in Self-Serv is coordinated by several peer software components called coordinators.

Coordinators are attached to each state of a composite service. They are in charge of initiating, controlling, monitoring the associated state, and collaborating with their peers to manage the service execution. The knowledge required at runtime by each of the coordinators involved in a composite service is statically extracted from the service's state chart and represented in a simple tabular form called routing tables. Routing tables contain preconditions and post conditions. They are used to determine when the service should be executed and what should be done after the execution. This way coordinators do not need to implement any complex scheduling algorithms.

eFlow

In eFlow [C$^+$00, CS01] a composite service is modelled as business process, enacted by a service process engine. A composite service is modelled by a graph, which defines the order of execution among the nodes in the process. The graph may include service, decision and event nodes. Service nodes represent the invocation of basic or composite service. Decision nodes specify the alternatives and rules controlling the execution flow, while event nodes enable service processes to send and receive several types of events. Arcs in the graph may be labelled with transition predicates

defined over process data, meaning that as a node is completed, nodes connected to outgoing arcs are executed only if the corresponding transition predicate evaluates to true. A service process instance is the process schema instance. The same service process may be instantiated several times, and several instances may be running at the same time.

In order to manage and even take advantage of the frequent changes in the Web service environment, service processes need to be adaptive, i.e., capable of adjusting themselves to changes in the environmental conditions with minimal or no manual intervention. eFlow provides dynamic service discovery, multi service nodes and generic nodes in order to achieve this goal. With dynamic service discovery service selection can be made at run-time, i.e. selecting the service that best fits the customers' need. Multi service nodes allow eFlow to invoke multiple instances of the same type of services, in order to request information from multiple services. A generic service node is a service node, that is not statically bound or limited to a specific set of services. Instead, it includes a configuration parameter that can be set with a list of actual service nodes either at process instantiation time or at runtime. Generic nodes are resolved each time they are activated, in order to allow maximum flexibility and to cope with processes executed in highly dynamic environments.

Process instances are run by the eFlow engine. The main function of the engine is to process messages notifying completions of method nodes, by updating the value of case packet variables accessed by the nodes and by subsequently scheduling the next method node to be activated in the instance, according to the method flow definition. When a method flow (i.e., an interaction with a given service) is completed, then the service node is also considered completed. The engine then determines the next service node to be activated (according to the service node definition), selects the service to be executed, and eventually starts invoking the methods on the new service. The engine also processes events, either internal events or external events, by delivering them to the requesting event nodes.

3 Interactive Service Composer for Mobile Phones

In the ITEA (E!2023) project S4ALL (Services for All) the goal is to have a world of user-centric services that are easy to create, share and use. In the project it is visioned that mobile terminals are used to access these services and compose services the way the end user wants to use them. Adhering to these goals of the project I have created Interactive Service Composer for Mobile Phones (later on referred as Composer), which is essentially a simple service composer for mobile phones.

Interactive Service Composer is a simple application that is able to do service composition. Composer is an orchestrating service composer, being only able to do simple service composition. It was designed to be a very simple application from the start. First ideas were borrowed from Apple's Automator [App07], which allows users to make simple linear workflows, where each workflow is composed of multiple simple tasks. These workflows can be saved and reused multiple times. These same features are available in Composer, there are a library of simple actions available, the workflow is composed of these actions and the workflows can be saved for later use. In Automator user is able to drag and drop actions from the defined list of actions into the workflow. Composer was never planned to be able to support dragging and dropping, but it is generally made to be simple and easily understood and used.

The building blocks of the workflows in Composer are called actions. The actions can be normal Web services, which are described with WSDL, or they can be small Python programs which use the phone's own resources to run a simple task. There are a lot of features in the Python runtime, which allows to make simple actions that for example send an SMS message to another person. The actions have a specified form which they have to be in to work with Composer, see chapter 3.5. This allows the automated handling of the actions, like inspecting the data field and running the action. The Python actions need to have only one function, named run, which is called when the action is executed in the workflow. Other functions are optional and can be made to make the reading of the action easier for other developers.

3.1 Architecture

Interactive Service Composer is a program made fully with Python running on Python for Series 60 (later on referred as PyS60) software on a series 60 software platform on a mobile phone. Python was chosen because it is a fast language to develop with. Together with PyS60 it was easy to make a choice what to use because

building simple user interfaces and developing prototypes is fast.

PyS60 is a Python runtime made for series 60 software platform. It is a full port of the Python programming language and it also provides access to many of the phone's smartphone functions, like camera, contacts and bluetooth communications. Because of the simple usage of the phone's own functions it is very easy to make simple yet powerful software with Python on the series 60 software platform. Although PyS60 does not include all the libraries that are available in the basic Python distribution, it is fairly easy to port the necessary libraries for the phone. Composer uses external libraries for SOAP handling and UDDI requests. The libraries are made for desktop computers, which brings some problems because of the small memory and low processing power of the mobile phones. I will describe more of this library size problem in chapter 5.2.

Composer has been from the start divided into three main elements. First of all there is the repository of actions which takes care of all the actions that has been saved into the phone's memory, be it actions coded with Python or Web services described with WSDL. Second part is the workflow-structure that takes care of the running the workflow, adding and removing actions from the workflow. The third main part of the program is the user interface, which is responsible for displaying the necessary elements on the screen in different parts of the program. The general architecture is shown in figure 2.

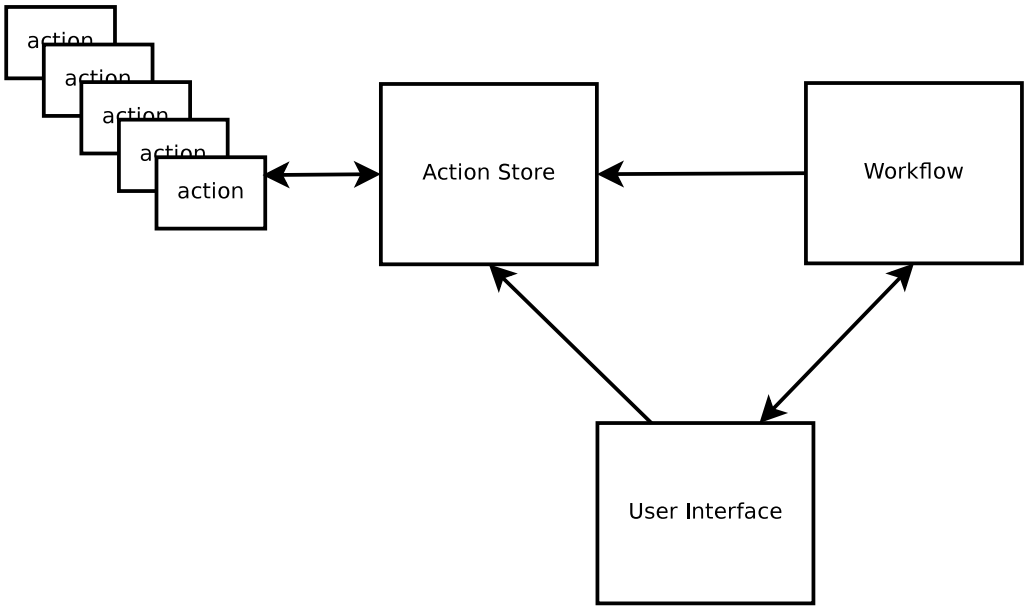

Figure 2: The general architecture of ISC

In order to handle the SOAP requests and responses Composer is using a SOAP library called SOAPpy. SOAPpy is one of the few SOAP libraries available for Python. At the time SOAP libraries were looked into there were no other good implementations available, at least not so simple as SOAPpy. SOAPpy does not handle complex types in WSDL specifications very well, which makes using more complex Web services difficult. The complex types in this case refer to for example collections of items, where the items are a simple type elements (string, number etc). SOAPpy is not able to construct a correct request message when these kinds of elements are in the service specification.

UDDI queries are handled with UDDI4py library. It is a fairly simple UDDI library, which implements the UDDI 2.0 functionality. The library is also using SOAPpy for sending the SOAP requests, which made the choice easier when selecting what library to use. The UDDI library also includes an API for adding new entries to the UDDI registry. This feature is not very feasible for a mobile phone, where the amount of required information is big. This information includes for example the URI where the WSDL file is found. I will describe more about the library problems with Composer on chapter 5.2.

3.2 User Interface

The user interface was kept simple throughout the development of the program. The simple user interface is both a nuisance and a benefit. The main thing the UI is working with are simple lists, which is also restricted by PyS60 which has a limited number of options on what to show on the screen. Composer does not need more complex elements than lists to show the workflow, but in some cases the lists were not able to show enough information. The workflows are shown as lists of actions, which also tells how they are run. Workflows are always run linearly one action at a time. It is possible to do simple loops in the workflow, though during the development they were not needed except for testing.

Building user interfaces on a mobile phone usually involves using a lot of menus. In Composer too most of the functionality is in the different menus that are related to different views. For example in the main screen the menu has functionality for creating a new workflow or loading a previously saved workflow, as shown in figure 3.

The workflow view has one workflow showing at a time. The view consists of a list of actions that have been added into the workflow, or alternatively a note saying that

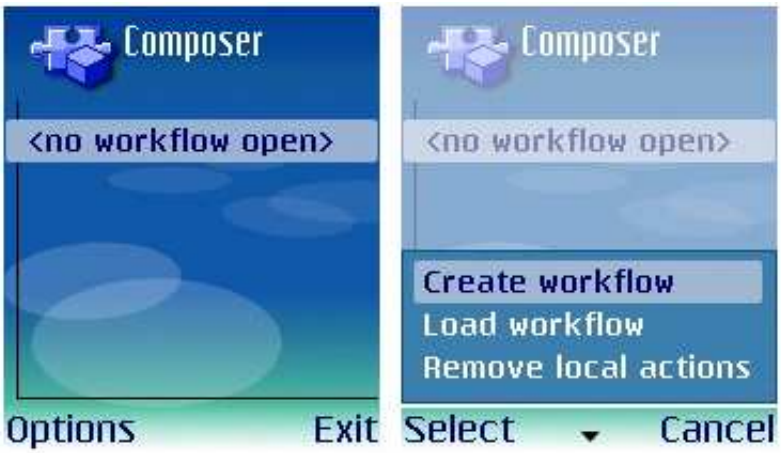

Figure 3: Main view - Main view menu

there are no actions in the current workflow. The heading shown in the top of the screen shows the name of the currently open workflow. The list shows the names of the actions that have been added into the workflow. In case of WSDL-files the action naming is two-fold. The name can be a generic name or it can be a name taken from the WSDL-file, because all WSDL files do not necessary include the name in a place where the SOAP library could find it. The longer documentation of the action is shown as a text-field on its own, also the configuration is done outside the basic list structure. There are a lot of items in the workflow menu, as the figure 4 shows. The workflow view supports many different operations, like adding new actions, removing old actions, running the currently open workflow or saving it.

Figure 4: Workflow view - Workflow view menu

3.3 Action Store

The repository of actions, or just action store, is responsible for the actions that are saved in the phone's memory. The module is a simple storage/retrieval class for the actions used in Composer. Action store takes care of the actions for the workflow module and delivers the required information to the user interface about actions. The first thing it is responsible for is when Composer is started to inspect all the actions (actions made with Python) that are present on the phone and make a list of them. Also at the same time the WSDL files (which describe Web services) are read and inspected. The actions are not loaded fully when the program starts, but only partially read, so that startup would be faster.

The actions made with Python have a special data-field in the file. This data-field is read when Composer is started. The data-field includes administrative information for Composer on how to deal with the action. This information is one of the central parts of the action. As the data-field has been read and the content stored in the action store model, the information can be used when adding new actions into the workflow. More information about the structure of the actions can be found in chapter 3.5.1.

Action store is accessed by the workflow module frequently when it is dealing with actions. As new actions are added to the workflow the input and output parameter types are checked. The types of the previous action and the action that the user is adding must match. These parameter types are available in the data-field of the action. Composer is also able to use previously saved workflows as actions in other workflows. The saved workflows are stored in the same place as the Python actions and are loaded to the action list when starting the program. More information about the workflow structure can be found in section 3.4.

3.3.1 Action Handling

Composer was planned from the start to be simple and generic tool. This same ideology can be seen in the action store. Although there are three types of actions, the handling of those three types is made to be as much similar as possible. The user interface requires a lot of information from the action store regarding the actions, what is the full name of the action, what are its parameters and so on. Action store can supply the required information for the user interface. The three types of actions are the firstly planned and made Python actions, the WSDL actions

(i.e. Web services described with WSDL) and the last are flow control actions. Previously saved workflows can also, as stated earlier, be used as actions, though they are composed of multiple actions.

To support the handling of all actions in the same way, the WSDL files are handled using a special wrapper that is able to supply the required information to the other modules. The module references (that are required to run the actions) in action store in the case of WSDL actions point to the wrapper class. The wrapper takes care of calling the instantiated WSDL proxy, made by the SOAPpy library, which in turn sends the request to the Web service. Both the wrapper and the proxy are generated only when the workflow is run. This saves time when some Web service is added into the workflow but the workflow is not run.

The Python programming language requires all files imported into the environment before they are usable. This is due to Python being an interpreted language. That is why all actions must be imported to the environment before running a workflow. Importing the Python actions dynamically to the PyS60 environment is fairly straightforward, as the file name is already known it is easy to use the dynamic importing method that is built in to the Python environment, which imports a specified file to the environment as a module. WSDL-actions are not used that way, they have to be given to the SOAP-library, which makes a WSDL proxy out of the file, which then can be accessed via its API. Generating a proxy of the WSDL-file makes accessing the Web service easier, although the SOAP library used in Composer is not able to understand all of the more sophisticated sides of SOAP.

3.3.2 Storage and Retrieval

The actions are stored in a hashtable, where they are indexed with the file name of the action without its file type extension. Besides the data-field the table stores a reference for the loaded module of the action. Only when the workflow is run the actions are imported to the Python environment and module references are updated to point to the loaded modules. If the action is a Web service described with a WSDL-file, the reference to the module is a reference to the WSDL wrapper class, created by the action store. The hashtable is kept current by refreshing the data in it after certain events. These events include saving a workflow to the phone's memory and searching and downloading a new Web service definition file from a UDDI server. By refreshing the list at these events it is possible for the action store to know at all times what actions are present.

When WSDL files are seen for the first time, Composer will generate a special cache-file out of them to make the loading faster. In the cache file Composer stores same kind of information that the Python actions have in their data-field. For WSDL files the input and output types are not known when the files are being added into a workflow, because it would require either initialising the WSDL proxy or parsing the XML. In the case of certain more complex Web services it might be that input and output types are not available even when the proxy is made. This is due to the library not being able to return more specific parameter list than just the complex type name required as the input. This will cause problems with services of which parameters the user does not know beforehand.

When user wants to add a new action into the workflow it is action store's responsibility to conduct the search on the actions. User can search for actions with a name of the action or a certain tag that has been given to the action by the developer of the action. Tags can be for example related to what the action does or is it an action made with Python or a Web service. Problems with searching are the limited possibilities to show the matching actions on the screen. In its simplest case the results are only shown as names of actions that match the criteria. This might make the selection of actions hard, because the user cannot see any longer documentation at that point. User can also search for new actions from an UDDI registry. More about this can be found in the chapter 3.3.3.

3.3.3 UDDI search

Action store is responsible in conducting a search for new actions from a UDDI (Universal Description Discovery & Integration) registry. When a user makes a UDDI search, the action store retrieves the matching actions (matching by name or matching by a certain tag) from the registry, which are then presented to the user. The list possibly is big so it might be hard for user to select the service that suits him/her the best. Also because the user interface has so few options in showing the results the list only has the names of the services that are available in the UDDI registry (matching the search criteria). A more sophisticated result list would include a some kind of description of the services that were found, this would make the choice for the user much easier.

After the user has selected the service that suits his/her needs, the WSDL of the service is downloaded into the phone's memory. At this point Composer does not know anything about the service, except that the WSDL file is available and it

can be added into a workflow. Like said previously, from all WSDL-files Composer will generate a special cache-file, which helps in handling the Web services. The cache file in this case is done after the downloading finishes and the list of actions is refreshed. After the WSDL file has been downloaded and saved into the phone's memory they can be added into workflows like usual.

3.4 Workflow Structure

Workflows are the base for building new service compositions in Composer. Workflows are simple lists of actions, be it Python coded actions, Web services or other workflows functioning as actions. Workflows are run linearly starting from top, currently workflows are not supporting conditional execution (if-then). There is a possibility to also do simple loops, but they are not used in many occasions. There is always only one workflow open at any time. The workflow model represents a single workflow that has any number of actions which can be run, saved and used in other workflows. The workflow module is responsible for loading, saving, running and populating the workflows.

The actions, that are available from the action store, are stored in a simple list. The actions are in the list in the same order that they are shown on the screen. When a user wants to add a new action to the bottom of the list or insert one between two actions the list is modified according to the users commands. Workflow list stores the whole action modules, so that when referring to an action it can be done straight, not by referring to the action store first.

Actions made with Python usually have some configurable parameters. Workflow module takes care of allowing the user to see and change the parameters. Parameters can be either freely enterable text or a list of pre-defined values from which user can choose the fitting one. The parameters are shown to the user in a form-type list where the parameter names and values are first shown in their default values. When the workflow is run, the parameters are passed to the action, be them changed by the user or not. More information about the parameters can be found in section 3.5, where the actions are described.

3.4.1 Populating the Workflow

When new actions are added into a workflow, the action is added to the list of actions in this workflow. At this point the action has not been loaded fully (imported

into the environment), the information available about the action is its name, description, input and output types and some administrative information. The input and output types are necessary to have, as they are checked to fit with the previous and next actions in the workflow. Input and output types in Composer are basic programming language types, numbers and strings. The input type checking is done by just checking that the actions that are going to pass parameters to each other have matching types defined. The parameter type checking is especially hard when considering Web services. As Web services have the types defined only in their WSDL files, it's hard to make the check work with the current SOAP library, especially if the output type is composed of multiple simple types. If the input and output types do not match, then the action is not added into the workflow.

Removing an action from the workflow does not require any special operation. The action that user wants to remove is removed from the workflow list so there is no reference to the action anymore. After that the list is reorganised due to one action possibly being removed from the middle of the list. When the action is removed from the middle of the list Composer does not check the input and output types of the remaining actions that were neighbouring the removed one. This might cause problems later when running the workflow, because the input and output types might not be compatible anymore. This modified workflow is considered to be different than the previous workflow with one more action, so the workflow can be saved as a new workflow, different than the previous one with one action less.

Like mentioned before, workflows can be reused after saving them. A saved workflow lists all the actions that was added to the workflow when it was made. This means that when loading a workflow Composer checks that all the actions are still available in the phone's memory. If one or multiple actions are not available the workflow cannot be loaded. Saved workflow also stores the parameter values that the actions have. These are loaded and inserted into the workflow model, which holds the actions and their parameters. When the necessary actions are loaded and the parameters are set, the workflow is runnable.

3.4.2 Running the Workflow

Before workflows can be run the workflow module has to conduct some administrative tasks. Because workflows can include workflows that can include workflows and so on, the module has to make sure there are no endless loops in the workflow. This is done by checking all the actions in the workflows and sub-workflows and mak-

ing sure there are no loops. As the actions are not imported to the system at the starting phase of Composer, before starting to run the workflow all of the Python made actions are imported, so that the modules are runnable. If there are any Web services in the workflow the required libraries are also loaded and the proxies are instantiated before being able to run the Web services.

Figure 5: Running a Workflow

The running of the workflow (see figure 5) starts with calling the first action with an empty input. There is no way of giving an input to the first action in the workflow in the current version of Composer. It might turn out to be necessary to give input to the first action, so this might change in the future. Input is specified to be a list, which helps in handling the different types of outputs that different actions return. Also during the demo preparations, we discovered that we needed a better way to store outputs than just a basic list. Because of this the demo actions save their more complex outputs into a hashtable which is given to them as an input. More about the demo and the experience received from it can be found in chapter 4. This information can then be accessed by actions later in the workflow. The pre-specified function in the first action is called and the action runs and returns something at the end. The output that the action returns is given to the next action and the same thing happens again. The whole workflow is run through the same way and in the end the result (or the returned item from the last action) is shown to the user. This is no different to actions being workflows, the inner workflow gives the input to its first action as it would have come from a previous action.

There are also a few types of special actions that require different to normal handling. These are the actions that are used in the device management demo and flow control

actions that are related to loops. Difference to normal actions with the demo actions are that they are given a bit different input list. This is because the actions in the demo need to store more information that we are able to pass in the normal input-lists. This includes, but does not restrict to, login information to the server where the Web services are hosted. The flow control actions require different to normal handling because the looping requires workflow to observe when the loop is ending. Workflow does not itself conduct the looping, but it passes the responsibility to the action. Workflow is then required to continue the running after the looping is done.

3.4.3 Saving the Workflow

Workflows can be saved for later use after user has composed the actions he/she wants it to have. The main point from the start of the development was to ease the reuse of existing workflows. This follows the ideas from Apple's Automator, which was made to help users to do tasks that repeat multiple times. Users are able to use their workflows over and over again, also as part of more complex workflows. Workflows are saved in the same place as actions so that they are easily found.

Every workflow has some actions added into it, be it actions or other workflows, empty workflows can't be saved. When saving a workflow Composer only saves the information what actions are used in the workflow and what are the parameters they have. This follows the idea used in BPEL [ES06], in a BPEL process it is specified what Web services are used and where they are found. In Composer the actions are locally available, either the WSDL file must be present or the action file must be present, otherwise the workflow cannot be loaded.

An important part of saving the workflow is the parameters the actions have. Parameters are a good way to customise the actions. So in order to ease the reuse of the workflows, the parameters need to be saved also. Parameters are stored in the workflow module as a list of key - value pairs. The parameters are saved with the workflow so that only the values are saved. The keys are then available from the actions itself, they are not changed so it is not sensible to save them.

Below is an example of a workflow that is saved to a phone's memory, shown in listing 5. It includes 4 different actions. Some of the actions have parameters saved. First lines include the name of the workflow, type of the action, which in this case refers to the action being in fact a workflow and the longer documentation of the workflow. Beginning on line 5 starts the list of the actions. Each action has the

Listing 5: An example of a saved workflow

```
1  data = {
    'doc': u'A composite action.',
3   'type': 'wflow',
    'name': u'DailyDilbert',
5   'actions': [
    {'paramvals': [([u'DailyDilbertImagePath', u'DailyDilbertImage
        '], 1)],
7     'file': 'dailydilbertaction.wsdl'},
    {'paramvals': [], 'file': 'base64_decode_action.py'},
9   {'paramvals': [u'c:\\file.jpg'], 'file': 'save_to_file_action.
        py'},
    {'paramvals': [], 'file': 'show_image_action.py'}
11  ],
    'tags': ['string', 'show', 'web service', 'save file',
13   'base64', 'decode', 'phone', 'screen', 'WSDL', 'image']
  }
```

parameter values and the filename of the action on the phone saved. On line 12 are saved the tags of the workflow. Workflows get all the tags of the individual actions, so that when searching for actions, it is also possible to find workflows that contain actions with certain tags.

3.5 Actions and Web Services

As stated before, actions are the building blocks of service compositions in Composer. Actions are referring to both Python-made actions that use the phone's services and Web services described with WSDL. From the start actions were planned to be as generic as possible so that they could be used in many places as possible. They were also planned to be easily configurable so that it would be easy to users to change the behaviour and to use them in different places. These goals were partly attained, but with the use cases in the device management demo it was not possible to try to stay on the generic action path.

The third type of actions is the flow control actions. They are not used in many occasions, the only one available at the moment is an action that allows users to make simple loops. The handling of these actions is not different to normal actions,

the looping action runs the specified actions for the required amount of loops and then the execution continues on the next action after the loop. This requires the workflow to do some extra administrative tasks when the loop ends, but otherwise the handling of the flow control actions is the same than the normal Python-made actions.

3.5.1 Actions

The Python-made actions have a certain format that must be followed in order to use them in Composer. This is required to have so that all actions can be handled similarly. This format defines where the necessary data is to be put and what function is called when the action is run. There is an example action in listing 6, which probably is the simplest action of them all. The purpose of this action is to show a text box on the screen, in where user can input text. The action has parameters that can be configured to show the default text and the title for the text box. On lines 1-10 is first the data-field that holds specific information for action store and workflow on how to handle the action and what is its name for example. This data field is loaded when Composer is started. It has all the necessary information for the action store to be able to find the actions and give them to workflows.

At line 13 starts the run-function that includes the code which is run when the action is called. Every Python-made action needs to have this function so that it can be run. It does not matter if the action has a number of other functions, it just needs to have this one in order to work properly. In the example the function is not very long nor very difficult to understand. That is what makes the action a good example.

Listing 6: An example of an action

```
data = { 'name': u'Query text',
2          "type": "python",
           'doc': u'Request a text from the user',
4          'parameters': [ ('label', 'text', u'my label'),
                           ('default', 'text', u'value') ],
6          'tags': ['text', 'request', 'dialog'],
           'input': 'anything',
8          'output': 'string'
         }
```

```
10
#---
12
def run(input, p):
14
        import appuifw
16      output = appuifw.query(p["label"], 'text', p["default"])
        input[0] = output
18      return input
```

Previously introduced parameters are also important to actions. With them it is possible to change how the action behaves, depending of course what the parameters are made to affect. On line 4 is the parameter field in the example action. The parameters are formatted already so that they can be, without modifications, shown in the parameter configuration screen in the workflow. The configuration form takes a list of three items, the name of the field, the field type and the field default value. The format is specified to be this because the form that shows the parameters requires all fields to be in this format.

The example is missing a couple of extra definitions that tell Composer what kind of input list to use and if the action is using a certain type of parameters. The first definition is for the previously mentioned demo-actions, they need a more powerful input list than just a list, so if the parameter is set, the action gets a hashtable as an input (that includes the possible outputs from the previous actions), not a list. The other definition is related to having a list of pre-defined values in one of the parameters. Composer changes the pre-defined list of parameter values to a simpler form when the definition is present, this helps the usage of the parameter value in the action.

3.5.2 Web Services

The second main type of actions are the Web services. They are WSDL files, that describe one or multiple different Web services. Composer does not itself handle WSDL files. Instead, they are given to the SOAPpy library, which makes a proxy out of them which allows the calling of the services from Python. The proxy on the other hand is not directly called from Composer, but the proxy is given to a wrapper class, that makes the Web services function like normal Python-made actions. There are some problems with input and output types with Web services.

If the Web services use complex datatypes as their input or output types, it is hard to find out what really are the types that the complex type is composed of. The problems mainly are because of the library because it is not finished and it would require more work to better handle the complex SOAP types.

The WSDL wrapper class has also a run-method which gets called when the workflow is run. The run method includes only the function call to the proxy. If the WSDL file has more than one service available, it is the users job to select the most appropriate service by configuring the entry in the workflow. Configuring the entry shows the list of the available services to the user. There are no other parameters for Web services available. The proxy is then invoked with the information what service is called and the parameters that the previous action has returned.

The returned output can be many things. A list of simple datatypes, a complex datatype that includes multiple simple datatypes and so on. This makes the handling of output a bit complex in some cases. The easiest way (that is used at the moment) is to just return the whole output, be it a list of some simple types or a single string, number or what ever the service happens to return. This way is not necessarily the best way, because it might be that the next action in the workflow does not need all the different items in the list. Also the generality is immediately lost if somehow only one specific part of the output is needed and it is necessary to be separated from the list. Although returning the whole output is the Composer way to do it, there might be a need to try and break up the output.

4 Application to Device Management

As a practical use case for Interactive Service Composer I will describe the experiences and insights received from using Composer in a demo. In this demo Composer is used to send commands to a device management server in order to allow an administrator to use the server without his laptop or his desktop computer. This demo will be presented to the Finnish partners in the S4ALL consortium. During development the examples never were too complex because I'd want to test out the newest feature implemented and move on. But in this demo the use cases are complex and it was interesting to note that Composer was able to deal with them.

We are working with S4ALL partners Capricode and Nokia to build this demo. In the demo the work is divided into three distinct elements. Our job is to handle the sending of device management (DM) [L+06] commands from Composer to the device management server made by Capricode. The server sends commands to the mobile terminals, that have a device management client made by Nokia. These interactions can be seen in figure 6.

The interaction with Composer and the device management server is in fact done via a proxy. This proxy was fully made by Pekka Kanerva, we worked together to build this demo. More about the proxy can also be found in [Kan07]. It was noted early in the demo planning that the Python SOAP library that Composer is using will not be able to communicate with the server in the correct way. Like stated previously, the SOAP library is not finished and it is not able to handle all parts of the SOAP specification. To tackle this we are using a proxy software coded with Ruby to handle the SOAP requests and responses. Ruby has a library that supports SOAP better than the Python libraries available at the time of writing. Like Python, Ruby has numerous powerful libraries in the basic distribution which makes developing small applications fast and easy.

Composer uses REST-calls to send commands to the proxy. These calls are then transformed into SOAP requests which are then delivered to the device management server. The transformation is simple because from the REST-calls it is easy to separate the needed parameters for the SOAP request. This way the complex SOAP requests can be populated and sent using the proxy. The data that the device management server returns is mediated to Composer on the HTTP response of the REST-call. This way Composer does not need to make more than one connection to the proxy. This though might be problematic when the SOAP calls take a long

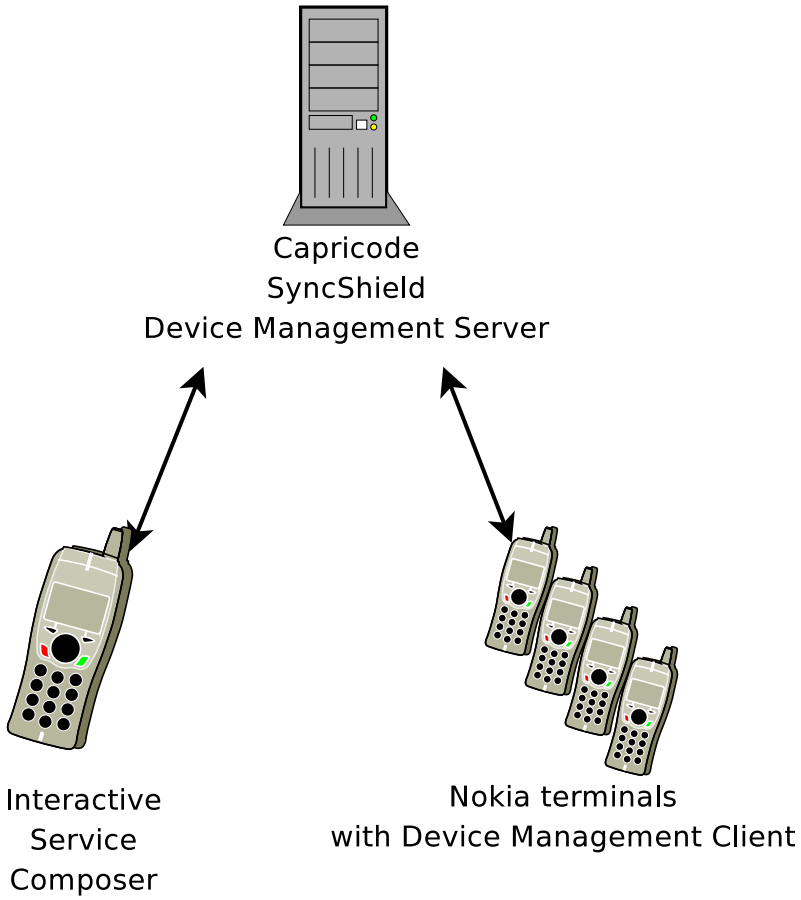

Figure 6: Interactions in the device management demo.

time to execute and Composer has to wait for the proxy to send a response back. The proxy does not have sessions, each command runs and ends in its own space and no information is stored between the commands.

Each command for the device management server has its own REST-call. This makes easier to parse the parameters and set them in the SOAP request. The device management jobs on the server are accessible through Web services described with WSDL. The proxy uses the WSDL files to access the services. The device management server is thus functioning as a normal Web service, where there are multiple services on one server. Difference to normal Web services here is the fact, that in order to use the services, an authentication service must be first used to be able to use the rest of the services. After this the proxy is able to send the SOAP requests to the other Web services. When the server sends the SOAP response to the proxy, the response is first parsed in the SOAP library. The data in the response

is given to the proxy from the library and it is put without any modifications to the HTTP response. The response to Composer is an XML message without the SOAP headers, the SOAP library removed them in the proxy. From the XML response Composer parses the necessary information and stores it. After this like with all actions the output is then passed to the next action. The general architecture of the proxy is shown in figure 7, picture taken from [Kan07].

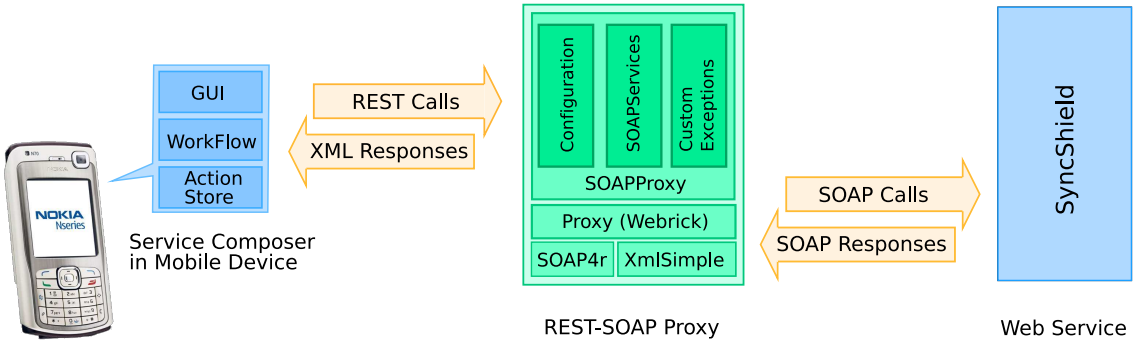

Figure 7: Proxy Architecture (taken from [Kan07])

4.1 Status

At the time of the writing we have kept the demo with two of the three specified use cases. I have described here what experiences we gained mostly from the first use case, where we had most of the problems. The second use case was not problematic, but most of the biggest problems were solved while developing the first use case. The second use case was implemented few months after the first use case was ready. This was due to one of the partners being busy. The third use case was left out from the demo, because we did not have enough time to implement it. I will describe it in any case how it was planned to be implemented.

4.2 Device Management

Device management (DM in short) refers to administering remote devices from servers. An administrator can remotely handle configurations, install and remove programs and detect problems on devices. The administrators can as well be wireless operators, service providers or administrators on a firm responsible for the mobile phones. With the increased interest in device management and the number of available products there are various proprietary device management implementations

that have evolved, which makes a threat for the interoperability among the devices and servers. To tackle this OMA (Open Mobile Alliance) Device Management specification [OMA04, L+06, AMV04] aims to unify them. The OMA DM specification has evolved from SyncML DM.

In general terms the OMA DM specification consists of three distinct parts. These are the protocol and mechanism specification, the data model and the policies. The protocol and mechanism specification describes the protocol used in the communication between the device management server and a mobile device. Data model describes the data made available in the mobile device, for example access point settings and email settings. The policies define who can manipulate a particular parameter or update a particular object in the device.

4.2.1 Protocol and Mechanism Specification

The protocol is defined based on SyncML Representation protocol and SyncML Synchronisation protocol. It includes packet elements that construct the device management messages, message transfer mechanism, and treatments that servers and clients should perform. To successfully perform a complete device management task, servers and clients go through two phases: the setup phase (figure 8) and the management phase (figure 9), figures taken from [L+06]. During the setup phase clients and servers authenticate each other and send device information. Packet 0 in setup phase is an optional package which can be used to start a device management session. After the client successfully sends the authentication and its device information in packet 1, the server sends its credential and information about management operations or user interaction commands in packet 2.

After completing the setup phase the management phase begins. This phase could have several iterations in single session until the required management tasks are completed. In this phase the client first sends its response to the server in packet 3 to the command in packet 2. After this the server can send another operation to the client in packet 4. After this there can be more iterations or otherwise the phase can end here.

About mechanisms, OMA DM specifies *bootstrap* and *notification initiated session*. Bootstrapping is a process where a clean device is provisioned to a state where it can initiate device management sessions. It is also possible to further bootstrap a device to be able to initiate sessions with another device management server. The

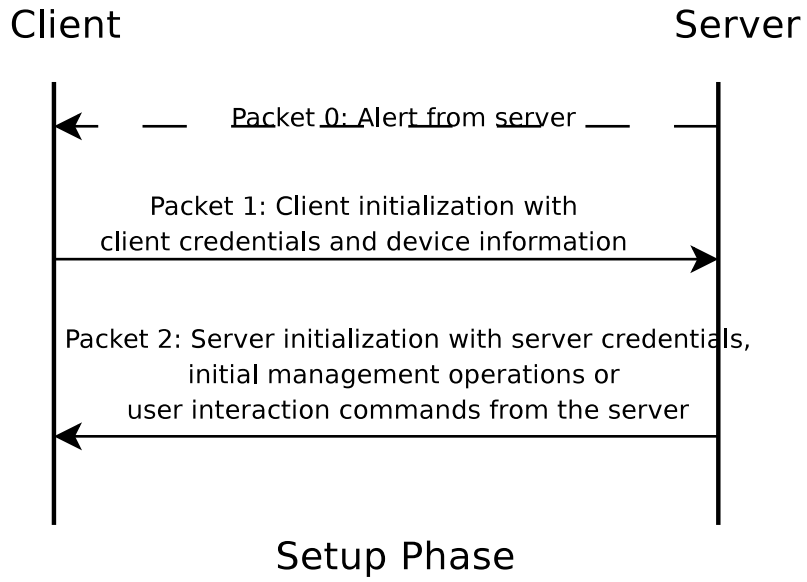

Figure 8: Setup Phase in Device Management (taken from [L+06])

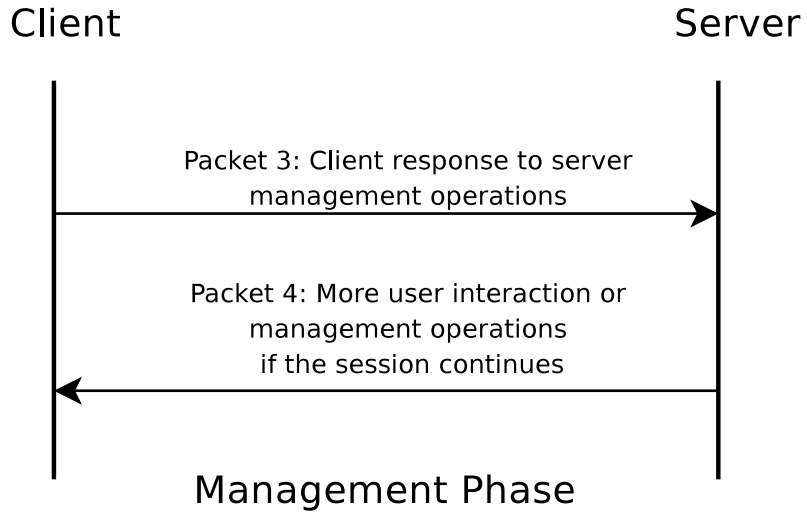

Figure 9: Management Phase in Device Management (taken from [L+06])

bootstrapping can be done as a server-initiated process, from smart cards, or through customised bootstraps, which may be done during the device manufacturing.

The second mechanism, the notification initiated session mechanism, actually specifies the packet 0 in figure 8. Normally clients don't have the resources or they are not otherwise consistently listening to servers' notifications because of security reasons. This mechanism provides a way for servers to notify the client to initi-

ate a new device management session. These occasions are for example when an administrator triggers a request on the device management server user interface to conduct some tasks. One occasion might also be that there are faults that require repairing. Security in device management is implemented with TLS 1.0 or SSL 3.0 as the protocol between servers and clients. MD5 ensures the integrity of device management messages.

4.2.2 Data Model

Each device that supports OMA DM contains a *management tree*. The management tree contains and organises all the available management objects, so that the nodes can be accessed directly through a unique URI. Nodes are entities that can be manipulated using the OMA DM protocol, for example the settings for e-mail or information about a certain application. An interior node can have infinite number of child nodes, while a leaf node must contain a value, null being a possible value. There are a set of associated run-time properties in each node. The properties are only valid for the associated node, except for the Access Control List (ACL). The ACL property can be inherited from a parent, so this property is not necessary associated only for this node. The ACL of a certain node tells which server can manipulate that node. The manipulation can be one of the following: adding a child node, getting node's properties, replacing the node, or deleting the node. An example management tree is found in figure 10, taken from [L+06].

Trying to perform device management tasks on OMA DM conformance devices would be insufficient without any common objects. Therefore OMA DM requires both clients and servers to implement three mandatory management objects and one optional object for clients and servers. The mandatory objects are OMA DM account management object, DevInfo management object and the DevDetail management object. The optional management object is called Inbox management object. These objects are defined in device description framework (DDF). The DDF provides servers the information that is necessary in order to manage the client devices. Device manufacturers can publish descriptions of their devices so that organisations operating device management servers can update the new descriptions to their servers. The servers can utilise the descriptions to manage the new functions the existing devices have or manage totally new devices.

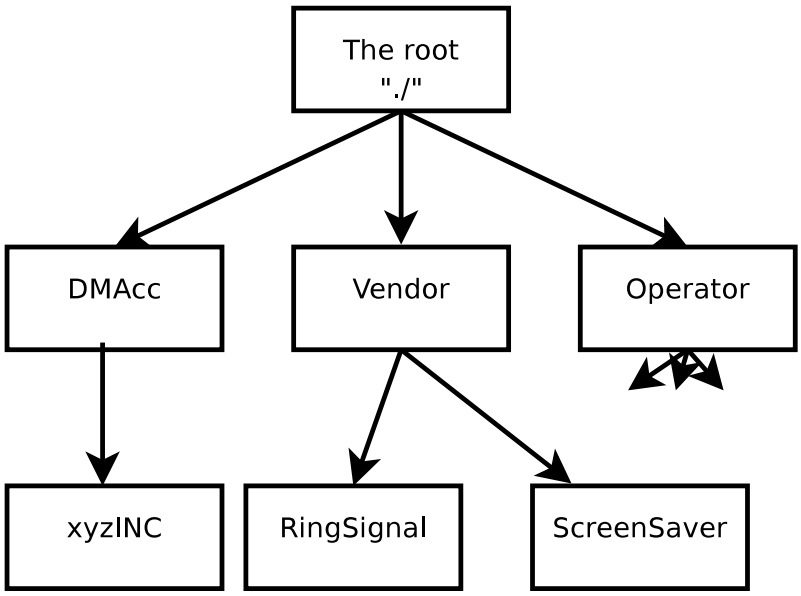

Figure 10: Example Management Tree (taken from [L+06])

4.3 Use Cases

We have planned three different use cases for the demo, which show some of the technologies present in device management. These features are not the only features present in the server and there are a lot of possibilities of what can be done with device management. The general scenario of the demo is that there is an administrator who needs to access the device management server from different places other than his office in which case he must use his mobile phone to do the things. The mobile phone in this case has Composer installed in it.

Including these three use cases, Composer is used to log in to the device management server and also it is possible to request a status of a certain job that is added to the server. These additional tasks were necessary to be implemented. Without the ability to query the status of a certain task Composer (and in this case the user) would not be able to know anything about it after the job has been added. The server does not contact Composer (or the user) after the task creation is done. Also because SOAP only allows request-response type interaction, it is easier for Composer to query the server about a certain task. The need for this extra functionality became quickly clear, it is a good feature in testing the demo and the feature is useful for administrators too.

The logon procedure was required to be implemented so that it would not be possible

for everyone to send commands to the device management server. Only if the user has a valid token included in the request the server will run the commands. This token is received after successfully authenticating with the server. The token is present in every request that Composer makes to the server, in order to keep the session secure.

4.3.1 Locking a Device

In the first use case we are demonstrating how it is possible to lock a mobile phone remotely. For example when someone has lost his phone or it has been stolen, it is possible for the administrator to access the device management server and send a command to the phone to lock itself. The locking is done on the mobile terminal by using the phone's own mechanisms. Different types of terminals have different ways to lock the phone. Usually locking means that the user cannot access any functionality of the phone without first entering an unlock code. After entering the code the user is able to use the terminal again. This schema is helpful when the phone might be stolen. If the person who stole the phone does not know the unlock code, it renders the phone useless to him. And the quicker the locking can be done, the less damage the stealer can make to the phone and the subscription. The process speeds up with Composer because the administrator does not need a laptop or a desktop computer to do the job.

In Composer the first use case is fairly simple. In order to lock the device, the device management server only needs the phone number of the device to be locked. This can be fetched from the phone's contact database. Also the user can type the number by himself. This phone number is then sent to the proxy, which delivers the SOAP request to the device management server.

When the device management server receives the request, it first checks that the device has been bootstrapped. This means that the device is present in the servers device database and it is able to initiate device management sessions when the server sends commands to the terminal. If the terminal is found in the database, the lock command is sent to the terminal. Before this the device management server returns a message to Composer via the proxy, which tells that the device management job has been added to the queue and will be run when other jobs before it are completed. After this the phone is locked in due course. Composer does not get any other status reports about the job without asking. By using the status query after the job has been added the administrator can check the situation, has the job been completed

or not.

4.3.2 Initialising a Device

The second use case is about adding a new terminal to the device management server, i.e. bootstrapping a device. With different device management implementations there might be a bit different ways to do this. Usually when the device phone number (or generally the information about the phone) is added to the device management server, the terminal that possesses the phone number receives a message, where it is instructed to fetch a special terminal software, which understands the device management commands that the server sends. This usually requires some user interaction. It also might be that the device does not require any terminal software. This is the case with the Nokia mobile terminals used in the demo. Bootsrapping in this case is done by registering the server with the built in device management client. This might require exchanging credentials or requiring user to allow the bootsrapping to be done.

In this use case Composer has the following tasks. The user is required to enter the phone number of the phone that will be initialised. This can be found on the contact database, but in this case it is more likely that the user will have to enter the number by hand. It was planned on the use case scenario that the added phone will be for a new employee, which is starting on the workplace. The workplace's phones include all the numbers of the workers there, so in this case the phone number can't be found in the contact database. In the scenario the phone number could be given to the administrator via phone by the new employees nearest superior for example.

The phone number is then sent to the device management server (via the proxy of course). The device management server adds the job to the queue and the server sends Composer a response to the request, response including the identification number of the job added. This job is then run in its time and like stated previously, the terminal that is to be initialised gets a request to either fetch the required terminal software or configure the built-in client to use the company's device management server.

In the API there are separate calls for adding a new device to the server and specifically starting the bootstrapping process. What Composer is required to do after adding the device to the server is to invoke another method, which will start the job to send the service message instructing the user to fetch the terminal software. For

this invocation the server requires a phone number, which the user must input again in Composer. Adding the device in Composer is in this case a two step process, in which the steps are separate workflows. When the phone has been initialised and added to the server the use case scenario also includes a task of sending of a basic software package to the terminal.

This package includes some necessary software in order to be used in the workplace. The terminal fetches the software package from the device management server when the server sends the command. This package is then automatically installed on the terminal, requiring only minor user interaction. There can be multiple different software packages on the device management server, the required software package can be selected in Composer when the phone number is sent. Optionally the software package might also include all the required settings for the new phone, which will be installed with the package. These settings might include access point settings, email settings and so on.

4.3.3 Sending Settings to a Device

The third and last planned use case is a situation where user needs the settings for Internet access to be sent to the phone. For example a computer illiterate person has somehow deleted the settings from the phone and now wants to access the Internet using his/her phone. The settings can be sent from the device management server, administrators might have added a set of common settings to all the phones in the company. These settings can then be easily sent to the phone so that the user can again access Internet.

This is implemented in the following way. Composer requires the user to supply a phone number that the settings will be sent to. This phone number is available either on the contact database on the phone or the user can input it by himself. The phone number is then sent to the device management server via the proxy. The server checks if the phone is present in the database and then sends the request to initiate a device management session to the terminal.

It might be that a company has a special set of settings for the access points and that is why it is necessary to use the settings that are present on the server, not the basic settings that the mobile operator supplies. The new settings are then fetched to the terminal and installed. After this the terminal can be again used to access the Internet. Composer will again only receive a response from the server

that the device management job has been added to the queue, but after this more information need to be requested from the server. It is possible to query the status of the job after it has been added, like stated before.

4.4 Experience

Using Interactive Service Composer in a demo gave a lot of new perspectives. It is a fairly different thing to test Composer with a few actions than trying to build a demo that interacts with a real product. It requires a bit more from the program and from the start of the demo planning it was clear that it would require more work put on Composer in order to be able to use it in the demo. But including these insights we noticed some problems with Composer, relating to a certain library and relating to the implementation of inputs and outputs between the actions.

Maybe one of the biggest problems with Composer were problems with the SOAP library. I have already previously stated how the library is not finished and it does not handle all parts of the SOAP specification well. When we first got the WSDL descriptions for the device management servers' services we tested them with the SOAP library that Composer uses. By using debugging features present in the library it quickly became obvious that the requests sent to the server were not formed correctly, the parameters were incorrectly formed.

The WSDL descriptions included a list of simple parameters (which corresponds to a complex type in SOAP) as an input to the services. In the description the items in the list are named so that the service can easily get the correct parameters the clients are sending, see listing 7 for example. Also in this case the list itself is named. These parts of the description seem to be problematic for the SOAPpy library. SOAPpy library was not able to correctly name the items in the list, which was not accepted by the service.

Like stated previously, to tackle this problem we built a proxy that changes the REST calls Composer is sending to SOAP calls that the Web services are able to understand. The proxy of course brought more work, but it really was a relief. Being able to use REST style Web services from Composer is easier, because not having to load the SOAP library to the environment. This makes using Composer a lot faster too. In [Kan07] there is a lot more talk about SOAP libraries and problems with them in this project, interested readers should refer to it for more information.

Another thing that could be listed more of an insight than problem was with param-

Listing 7: An Example Complex Type

```
<xsd:element name="Login">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="domain" type="xsd:string"/>
   <xsd:element name="username" type="xsd:string"/>
   <xsd:element name="password" type="xsd:string"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

In a SOAP message it should look like

<Login>
 <domain>domain</domain>
 <username>uname</username>
 <password>pass</password>
</Login>
```

eter passing between actions. Before the demo parameters passed between actions in Composer were plain lists, which had one or multiple items. When we learnt that in order to use the services Composer needs to authenticate to the server and carry a token that is given after successful authentication, the simple list was not perceived to be enough. To save a token in the list to be used somewhere later in the workflow will not work. It might be that the workflow has some actions that don't keep the list intact, although our policy was that actions only use the amount of parameters they need from the list and keep the rest intact. But 'maybe' is not good enough.

The first ideas of Composer were borrowed from Apple's Automator, as described in chapter 3. This simple list idea might work with a closed environment that Apple has, but with actions that has been made by someone else than you, it is more likely that something will go wrong or it might be that someone has made an action that has only intention to do harm. Trying to store special inputs somewhere in the list to save them for actions somewhere later in the workflow will not work. We wanted to have a more precise solution for this problem.

This ultimately lead to changing the input parameter list type to be different when running the demo actions. It gives more options to the actions to save the necessary items. The new type of input list is a hashtable that in fact has the basic input list

in it and the demo actions have the token saved there. The hashtable is not given to all actions, only actions specified to be able to handle it get it. This is saved us from changing all the actions to support the new type of input list. Although at first I was going to change all actions according to it, but when I realised I could use the data-field for it, the change only affected actions that needed the new input list.

While we were able to tackle some of the parameter passing problems with a more complex list of parameters, the amount of parameters in some of the SOAP calls were so big that it is not feasible for Composer anymore. Especially while implementing the second use case I was struck by the big number of parameters required in some calls. While being able to supply the parameters by asking them from the user, in my opinion it is not feasible anymore, when the first ideas were to keep input and output to be simple and only one item. Some of the SOAP calls required six to seven different parameters, supplying all these parameters is seriously crippled by the unavailability of good user interface elements in Python for Series 60. On the other hand parameters can be supplied via the action parameter interface, but this is not necessarily a good way. That is because people might forget to look at the configurable parameters and the default values someone has defined are not necessarily good. One point is also that configurable parameters were not specified to use in this way when the first plans for Composer were made, they were specified to alter the behaviour of the action.

The experiences generally gained from the demo were good. We were happy how Composer was able to perform with the demo. Also making the proxy gave us good experience about REST-style Web services, of which we didn't have a lot of experience beforehand. Although there were some problems at first, I think we managed very well and we were able to overcome the problems by making Composer better.

5 Experience Developing for Mobile Phones

With the recent increase in the mobile phones' processing power it has become feasible to develop software for mobile terminals. This is a very interesting subject for service providers and mobile phone operators as more and more people are carrying mobile phones. These mobile terminals are with people where ever they go, which gives a huge amount of possible new usage scenarios and business potential.

Developing applications for mobile phones has been possible for several years now with mobile operating systems like Symbian OS [Sym07], Windows Mobile [Mic07] and Qtopia [Tro07]. Albeit the languages needed to learn to develop for these operating systems are not very easy to learn (the case with Symbian C++, which has some special conventions to learn), there has been people developing for these platforms for quite a some time now. But when Nokia brought Python for Series 60 environment on its Series 60 phones developing became a fairly large amount easier, at least when looking on the viewpoint of language complexity.

Like introduced earlier, Python for Series 60 is a full Python implementation running on the Series 60 smartphones. The Python programming language is simple to learn, yet it has some powerful features built in. With PyS60 users are able to access the smartphone features and build applications without having to learn the special features of the specific language, Python is the same on desktop PC and on the mobile phone. With simple user interface elements it's easy to build applications on the mobile phone with Python, especially prototyping can be done quickly and easily.

Composer was first developed for a Nokia 6600. It has 6 megabytes of memory and it has a 32 megabyte memory card. Developing for this mobile phone became quickly a tedious task, because the phone is fairly slow, being one of the early Series 60 mobile phones. When adding new features it became clear that there will be a need to get a better mobile phone in the near future. The last straw was when we started to use the SOAP-libraries which took a really long time to load. Amount of time required to load the library on the PyS60 version back then was one minute and few seconds.

The next mobile phone we developed for was a Nokia N70, which worked much more smoothly. It has 22 megabytes of memory and a 64 megabyte memory card. With this mobile phone it was more smooth and the library loading did not take that long time. And when the loading of actions was changed it was fairly smooth to use

Composer on the N70. With the latest development on PyS60, running Composer has become even smoother, see next chapter for more details.

5.1 Resource Limitations on Mobile Phones

The amount of memory and processor speed are not problems when planning and implementing a piece of software for a desktop PC. With big amounts of memory and fast processors even more and more complex pieces of software are being able to run on desktop PCs. But this is not the case with mobile phones. Mobile phones have only recently been gaining more processing power and more memory, still having only a small amount of what desktop PCs have.

It would not be feasible to put a processor that powerful on a mobile phone that desktop PCs have. It would drain the battery instantly and the advantage gained from this would be very small. Mobile phone users would much rather have mobile phones that have batteries that last long. Also it most certainly is what companies are aiming for. Users have similar needs, users want the mobile phones to be smaller and smaller the batteries can't be big sized. This on the other hand means that the batteries have less power. There is a clear contradiction here. Mobile phones need to have more processing power but at the same time keeping the size small, keeping the battery long living and fairly small.

A clear example of the resource limitations on mobile phones is the following case. We are running fairly large libraries on Composer, which are needed to handle the Web service invocations. The library consists of multiple files with over 5500 lines of code. This does not include all the other libraries it is using. When running Composer on the mobile device, loading this library takes a long time. Some of the slowness is because of the Python environment and its memory handling. In the start of the June of 2007 Nokia released a new version of the environment which made the library loading two times faster than it used to be. I have provided some measurements about this in listing 8. Clearly the Python distribution has become faster, but this also shows that there is still a huge gap between how the desktop PC performs related to mobile phones.

Another example of having limited resources is the case with Composer inspecting the actions when starting. The actions are not loaded fully when Composer starts, but only partially read, so that starting the application would not take so long. Even with the amount of actions nearing 20 the loading of Composer became quickly slow,

Listing 8: Timings on Library Loading

Hardware and Platform	Time in seconds to import library	Time to build SOAPpy WSDL proxy
Linux 2.6.20 P4 3ghz, 2gb Python 2.4	0.08 seconds	0.01 seconds
Nokia N70 22mb memory		
PyS60 1.3.17	40 seconds	15 seconds
PyS60 1.3.22	21 seconds	2 seconds
Nokia 6600 6mb memory		
PyS60 1.3.17	58 seconds	19 seconds
PyS60 1.3.22	31 seconds	3 seconds

while all actions were imported to the environment at start time. Because all of the actions are not used in workflows, there is no need to load them all at startup. This change to reading only a part of the file made the starting of Composer faster. Accessing files from a memory card on the mobile phone is not a problem in this case. It is as fast as reading from the internal memory.

I don't exactly recall the amount of time that it took to start up composer anymore, because it has been at least seven to eight months since it was changed. But I think it was close to being thirty seconds. This probably partially also depended on the Python for Series 60 environment, because it has had a lot of problems with importing files to the environment being slow. It seems that now with the recent development the problem has been fixed it and the importing works a lot faster than it used to be. We have never had much more than 20 actions in Composer so it's hard to say how the current solution would work with amounts like hundred or more actions. I do believe though that it would work better than the first solution.

These two things clearly show what things need to be addressed when implementing a piece of software for mobile phones. Big libraries are a problem, they usually are planned to be used with desktop PCs that are a lot faster than mobile phones. And

when something is repeated multiple times it is usually smart to stop for a while and think if some part of it could be done faster. Or alternatively can it be done with less memory, or with generally less resources.

5.2 Libraries and Python for Series 60

When developing with a mobile phone one has to pay attention to the size of the programs because the resource restrictions of the mobile terminals. This does not fit to the picture where developed programs need to use libraries that are made for desktop computers. This is the usual case when developing something with PyS60. There are a lot of libraries available for Python in general, but they are big or require other libraries to work. Libraries that use other libraries very quickly make a pile of libraries that all need to be ported to PyS60. It might require some extra work to get the libraries working on the mobile phone, because the Python distribution is not complete. For example the weak reference implementation is missing from the distribution, this relates closely to the implementation of XML.

When considering specifications like SOAP it is clear that the library is big in order to support all the features in the specification. This might then make using the library on a device with less memory and lower processing power slower. With the size of the library also usually the amount other libraries needed increase. This makes the job of trying to use the library on a mobile phone harder, because more libraries are needed to checked to be working with the mobile phone. Our experiences with SOAP libraries for Python were not encouraging. The libraries we tested were not complete nor fully functional. In [Kan07] there is a lot more discussion about SOAP libraries for Python and Ruby and the experiences gained from this project.

Another big specification is UDDI. The library we are using to conduct UDDI searches in fact also uses SOAPpy SOAP library, which was chosen before thinking about UDDI. It was a relief that there was no need to provide another SOAP library for the UDDI search. Still the library in this case is fairly big too. In Composer we are only using the search functions of the UDDI specification. The specification also includes functionality for adding new entries to the registry. This functionality is not needed in Composer and it would make the library size smaller if it would be removed. On the other hand it would not be sensible to just implement some part of the specification and leave some parts out. When porting the library for the mobile phone, we tried to remove some of the imports in order to keep the size of the imported library smaller. This same operation was done for SOAPpy which also

included some server-side functionality, those parts are not loaded when using the library in Composer.

5.2.1 Problems with Libraries

We had most of our library problems with either big libraries or big amount of libraries that were needed to be supplied in order to make the library work. This made porting the libraries to the mobile phone harder. Usually problems relating to big amount of library references were because some of the libraries were specific to certain operating system and if the device's Python distribution did not have the library included, we needed to supply a library taken from the desktop PCs distribution with some changes. These changes were not necessarily big, but in any case it required some work to make the changes and move the library to the mobile phone.

It didn't help the developing and working with libraries when at first we didn't know how we could send all files from one library at once and also to put them in a directory in the mobile phone where we wanted them to go. We eventually came out with a notion of making a SIS-package (Symbian Installation Source packages) out of the library files so that they can be easily copied to the mobile phone. The libraries need to be at a certain directory on the mobile phone in order to be able to use them. Single files can be of course be copied automatically to the library folder with the help of PyS60, but after that they need to be moved to another folder to keep all libraries organised. This also gave a requirement to find a way to copy all files at once.

As we were developing with Linux, it was not easy to find the right tools which are able to make SIS-packages. In this case it helped that some of my co-workers were been developing for PyS60 earlier and had been able to find a program that was able to build the packages. Eventually we were able to automate the packaging process so, that we had a script file which was able to build all necessary packages if there had been any changes in the source files. This made transferring the files a lot easier when one needs to send only one or two files to the mobile phone.

5.3 Problems and Solutions

The extra difficulty when developing for mobile phones is the fact that one has to first transfer the files to the mobile phone in order to test out the newly implemented

feature. This became a nuisance in one point while trying to fix some errors and it required continuously to send a new version of the files to the mobile phone. One of my co-workers used a script to transfer the specified files to the mobile phone with a serial cable. This would have helped in some point when we had not yet come up with the system to create SIS-packages out of the required files. When transferring files to a phone and trying out the changes it requires you to concentrate on a different thing and you might lose thoughts of what you were about to do next.

Possible solution for this problem is using an emulator. This would be possible if not using Linux or Mac OS for development because for these operating systems there are no emulators available. Nokia has emulators only for Microsoft Windows. With emulators it is possible to test out the simplest features and see if they are working or not. Nonetheless it is not possible to test the features available in Python for Series 60, like the camera functionality or sending an SMS-message. The installation also requires a fair amount of work to be able to use them properly.

There is also one Python library [Rim06] available that is able to show the user interface elements of the mobile phone on a desktop computer by using another library to build frames and lists. The library includes the same API as the Python for Series 60 user interface and it shows a same kind of view what the software would look like on the mobile phone. This is also a good option for testing out features that don't use any of the smartphone's features. It is also easy to test features that require Internet connection because Python's normal libraries for network access can be used. This library was made here in HIIT, but only late in the project I started to use it. It helped in a few situations where I needed to make some fairly large changes to the software.

Generally debugging on mobile phones is harder because of the extra cycle with sending new version of the file to the mobile phone. Also if you run into a problem with libraries that you have ported you have to first check the libraries on the desktop pc to see what might have caused the problem and then again send the new files. Debugging at its worst case usually meant that I needed to include printing of debugging information in the problematic places to see where the execution went wrong. This usually lead to many cycles of trial and error, which were a nuisance while developing Composer.

5.4 Conclusion

While developing Composer there has been a lot of problems, be it smaller or bigger problems. These problems helped to see the development for mobile phones being a fair amount of different than for desktop PCs. I'm happy to see what we were able to develop but naturally there were some ideas that were not pursued further. I was hoping Web services would work better with Composer, but when I realised there were not many usable Web services available (because of the library) I was a bit disappointed how it turn out.

I see problems related to libraries as the biggest problems we had with Composer. Problems with SOAP library are a big thing in itself, but also when trying to bring new libraries to the mobile phone there were already some problems. Another big problem we had was with SOAP. There still is no really good, working SOAP libraries for Python. There has been a lot of development lately in one library which unfortunately does not fit the needs of Composer. Composer needs to have a proxy-like structure, where inputs and outputs are easily handled, the library recently being worked on offers a script that transfers WSDL-files to runnable Python scripts. This kind of setting does not really fit to Composer.

When developing for mobile phones it is necessary to first think what needs to be done. After the thinking you should then see how it performs on the device you are developing for. It is also necessary to think what libraries are needed in using the program. Are they available in the specific Python distribution (Python for Series 60 does not have all the standard libraries)? Do the libraries need other libraries? And when solving a problem, the easiest solution might not always be the best solution. These are the main points I got from developing for mobile phones.

6 Conclusion

With increasing interest in Web services, service oriented architectures and service composition, the thought of developing applications with the service composition paradigm has become more and more accepted. With service composition users are able to build new applications by combining existing services into more complex applications. Service composition is a way to master complexity, combining simpler building blocks into a complex application.

The dominant technologies in Web services architecture are SOAP for communication, WSDL for description and UDDI for the registries. These are not the only technologies available for Web service architecture, but usually these are the ones which are connected with Web services. This architecture is based on three components: the service requester, the service provider and the service registry. This is a similar way to how conventional middleware are built.

For the S4ALL vision I have created and described the Interactive Service Composer for Mobile Phones in this thesis. With Composer users are able to create service compositions, which are built from web services or actions functioning on the phone's own resources. By combining different Web services users are able to build applications that suit their needs. By reconfiguring, saving and reusing workflows users have a lot of possibilities to develop for their own needs.

While Composer is able to build simple compositions, I would have liked to see Composer being able to use all kinds of Web services, currently the SOAP library is constraining the choice of services. I would also liked to be able to develop some of the more sophisticated sides of the plans that were made in the S4ALL project. This would have required a lot of time and resources than we had in our use.

While developing Composer the best way to test its capabilities was to use it in a demo. In the demo Composer is used to build service compositions which send commands to a device management server. All problems and errors are not usually found when developing the program and trying to realise the plans made out for it. By building the demo with the partners we learnt a lot of Composer and its defects and good sides.

The whole demo process was not only about fixing the problems Composer had, but it was also very useful for us to see what Composer was able to do. Mainly because our partners were busy with other things, one of the three use cases is still undone. While the first use case gave us a lot of good experiences I'm wondering what other

experiences we could have received from them.

When developing for mobile phones the developer needs to take account different things than while developing for desktop PCs. While developing Composer I had problems with libraries on the Python for Series 60 environment. These problems originated from the fact that the libraries were either not finished or while trying to bring the required libraries to the mobile phone the dependencies (on other libraries) caused problems. A problem of its own was the size with certain libraries, which is a problem for resource scarce mobile phones.

Though SOAP is one of the dominant technologies in Web services architecture, the library problems (size, dependencies) render it hardly usable on mobile devices. While not trying to defame SOAP itself, the libraries we tried to work with were not very well fit to be used in a mobile device with low processing power.

Finally there were few interesting things I learnt from designing and building Composer. First of all the service-oriented architecture is an interesting concept. Service composition is also very interesting topic. By using services already built to make totally new services is an interesting idea, which could ease the development of new applications. At the start of the project it was a bit hard to understand all the technologies related to SOA, but after a while they started to seem clear.

What comes to developing with mobile phones the biggest insight I received was the problems with libraries. It seemed to follow all the way through the whole development process. First in the start when we were trying to find suitable libraries for the development and in the end while developing the demo. These problems are not only about libraries being broken, but also library size problems, dependency problems and so on. Another thing learnt was the lack of processing power on mobile phones. It became very quickly obvious how the lack of processing power affected the development.

The environment is so much different and interesting than with desktop PCs that although the lack of processing power is clear, the appeal is so big that there will a lot of development for mobile phones. And the trend seems to be continuing. As is the trend of developing software with service-oriented architecture paradigm.

References

A+03 Austin, D. et al., Web Services Architecture Requirements, October 2003. URL http://www.w3.org/TR/wsa-reqs.

A+04 Alonso, G. et al., *Web Services; Concepts, Architectures and Applications*. Springer, 2004.

AC+03 Andrews, T., Curbera, F. et al., Business Process Execution Language for Web Services version 1.1, May 2003. URL http://www.ibm.com/developerworks/library/ws-bpel/.

AMV04 Adwankar, S., Mohan, S. and Vasudevan, V., Universal Manager: Seamless Management of Enterprise Mobile and Non-mobile Devices. *IEEE International Conference on Mobile Data Management*, 2004.

App07 Apple, Apple's Automator, 2007. URL http://www.apple.com/macosx/features/automator/.

B+06 Bray, T. et al., Extensible Markup Language (XML) 1.0, August 2006. URL http://www.w3.org/TR/2006/REC-xml-20060816/.

BDFR03 Benatallah, B., Dumas, M., Fauvet, M.-C. and Rabhi, F., Towards Patterns of Web Services Composition. In *Patterns and Skeletons for Parallel and Distributed Computing*, Springer-Verlag, 2003, pages 265–296.

BL06a Bichler, M. and Lin, K.-J., Service-Oriented Computing. *IEEE Computer*, 39,3(2006), pages 99–101.

BL06b Booth, D. and Liu, C. K., Web Services Description Language WSDL Version 2.0, March 2006. URL http://www.w3.org/TR/wsdl20-primer.

BSD03 Benatallah, B., Sheng, Q. Z. and Dumas, M., The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7,1(2003), pages 40–48.

C+00 Casati, F. et al., Adaptive and Dynamic Service Composition in eFlow. *Proc. of 12th International Conference on Advanced Information Systems Engineering (CAiSE)*, 2000.

C+02 Curbera, F. et al., Unraveling the Web Services Web: An introduction to SOAP, WSDL and UDDI. *IEEE Internet Computing*, 6,2(2002), pages 86–93.

C+03 Curbera, F. et al., The Next Step in Web Services. *Communications of the ACM*, 46,10(2003), pages 29–34.

Col04 Colan, M., Service-Oriented Architecture expands the vision of Web services, part 1, April 2004. URL `http://www-128.ibm.com/developerworks/library/ws-soaintro.html`.

CS01 Casati, F. and Shan, M.-C., Dynamic and Adaptive Composition of E-Services. *Information Systems*, 21,3(2001), pages 143–163.

DK75 DeRemer, F. and Kron, H., Programming-in-the-large Versus Programming-in-the-small. *Proceedings of the International Conference on Reliable Software*, 1975, pages 114–121.

ES06 Ezenwoye, O. and Sadjadi, S. M., Composing Aggregate Web Services in BPEL. *ACM-SE 44: Proceedings of the 44th annual Southeast Regional Conference*, 2006, pages 458–463.

Fie00 Fielding, R., *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine, 2000.

FW04 Fallside, D. C. and Walmsley, P., XML Schema Second Edition, October 2004. URL `http://www.w3.org/TR/xmlschema-0/`.

G+06 Gudgin, M. et al., SOAP Version 1.2, December 2006. URL `http://www.w3.org/TR/soap12`.

HS05 Huhns, M. and Singh, M. P., Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9,1(2005), pages 75–81.

JM76 Jones, N. D. and Muchnick, S. S., Binding time optimization in programming languages: Some thoughts toward the design of an ideal language. *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, 1976, pages 77–94.

Kan07 Kanerva, P., State-of-the-Art of SOAP Libraries in Python and Ruby. Technical Report 2007-2, Helsinki Institute for Information Technology, 2007. URL http://www.hiit.fi/node/87.

Kha02 Khalaf, R., Business Process with BPEL4WS: Learning BPEL4WS, Part 2, 2002. URL http://www-128.ibm.com/developerworks/webservices/library/ws-bpelcol2/.

KS06 Kobti, Z. and Sundaravadanam, M., An Enhanced Conceptual Framework to Better Handle Business Rules in Process Oriented Applications. *ICWE '06: Proceedings of the 6th international conference on Web engineering*, 2006, pages 273–280.

L+06 Lin, S. et al., An introduction to OMA Device Management, October 2006. URL http://www.ibm.com/developerworks/library/wi-oma/index.html.

Lin00 Linthicum, D. S., *Enterprise Application Integration*. Addison-Wesley, 2000.

MH05 Mendling, J. and Hafner, M., From Inter-Organizational Workflows to Process Execution: Generating BPEL from WS-CDL. In *Lecture Notes in Computer Science*, volume 3762, Springer-Verlag, 2005, pages 506–515.

Mic07 Microsoft, Microsoft Windows Mobile, June 2007. URL http://www.microsoft.com/windowsmobile/default.mspx.

Min06 MindReef, MindReef SOAPScope, 2006. URL http://www.mindreef.com/mindreef/soapscope.php.

MJS06 Molina-Jimenez, C. and Shrivastava, S., Maintaining Consistency between Loosely Coupled Services in the Presence of Timing Constraints and Validation Errors. *ECOWS '06 4th European Conference on Web Services*, 2006.

MM04 Milanovic, N. and Malek, M., Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8,6(2004), pages 51–59.

OAS05 OASIS, UDDI version 3.0, February 2005. URL http://www.uddi.org/.

OMA04 OMA, OMA Device Management Specification version 1.1.2, January 2004. URL http://www.openmobilealliance.org/release_program/dm_v112.html.

Pap03 Papazoglou, M. P., Service-Oriented Computing: Concepts, Characteristics and Directions. *Proc. of the Fourth International Conference on Web Information Systems Engineering (WISE'03)*, 2003.

Pel03 Peltz, C., Web Services Orchestration and Choreography. *IEEE Computer*, 36,10(2003).

PG03 Papazoglou, M. P. and Georgakopoulos, D., Service-Oriented Computing. *Communications of the ACM*, 46,10(2003), pages 25–28.

PL03 Perrey, R. and Lycett, M., Service-Oriented Architecture. *Symposium on Applications and the Internet Workshops*, 2003.

Rim06 Rimey, K., PyS60 Emulation Library, 2006. URL http://sourceforge.net/projects/pys60-compat/.

RTF06 Ross-Talbot, S. and Fletcher, T., Web Services Choreography Description Language: Primer, June 2006. URL http://www.w3.org/TR/ws-cdl-10-primer/.

Rub07 Rubio, D., An Introduction to JSON, February 2007. URL http://dev2dev.bea.com/pub/a/2007/02/introduction-json.html.

SBDM02 Sheng, Q. Z., Benatallah, B., Dumas, M. and Mak, E. O.-Y., SELF-SERV: A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment. *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China*, August 2002, pages 1051–1054.

SK03 Srivastava, B. and Koehler, J., Web Services Composition - Current Solutions and Open Problems. *ICAPS 2003 Workshop on Planning for Web Services*, 2003.

Sym07 Symbian, Symbian OS Mobile Operating System, June 2007. URL http://www.symbian.com/symbianos/index.html.

TBB03 Turner, M., Budgen, D. and Brereton, P., Turning Software into a Service. *IEEE Computer*, 36,10(2003), pages 38–44.

Tro07 Trolltech, Qtopia, June 2007. URL `http://trolltech.com/products/qtopia/`.

VBS04 Vidal, J. S., Buhler, P. and Stahl, C., Multiagent Systems with Workflows. *IEEE Internet Computing*, 8,1(2004), pages 76–82.

vdADtH03 van der Aalst, W., Dumas, M. and ter Hofstede, A., Web Service Composition Languages: Old Wine in New Bottles? *Proc. of the 29th EUROMICRO Conference New Waves in System Architecture*, 2003.

Vin02a Vinoski, S., Putting the "Web" into Web Services - Web Services Interaction Models Part 2. *IEEE Internet Computing*, 6,4(2002), pages 90–92.

Vin02b Vinoski, S., Web Services Interaction Models - Part 1: Current Practice. *IEEE Internet Computing*, 6,3(2002), pages 89–91.

XMe07 XMethods, XMethods UDDI Registry, 2007. URL `http://www.xmethods.net`.