Enabling Personalized Composition and Adaptive Provisioning of Web Services

Quan Z. Sheng[1], Boualem Benatallah[1], Zakaria Maamar[2], Marlon Dumas[3], and Anne H.H. Ngu[4]

[1] School of Computer Science and Engineering
The University of New South Wales, Sydney, Australia
[2] College of Information Systems
Zayed University, Dubai, U.A.E
[3] Centre for Information Technology Innovation
Queensland University of Technology, Brisbane, Australia
[4] Department of Computer Science
Texas State University, San Marcos, Texas, USA

Abstract. The proliferation of interconnected computing devices is fostering the emergence of environments where Web services made available to mobile users are a commodity. Unfortunately, inherent limitations of mobile devices still hinder the seamless access to Web services, and their use in supporting complex user activities. In this paper, we describe the design and implementation of a distributed, adaptive, and context-aware framework for personalized service composition and provisioning adapted to mobile users. Users specify their preferences by annotating existing process templates, leading to personalized service-based processes. To cater for the possibility of low bandwidth communication channels and frequent disconnections, an execution model is proposed whereby the responsibility of orchestrating personalized processes is spread across the participating services and user agents. In addition, the execution model is adaptive in the sense that the runtime environment is able to detect exceptions and react to them according to a set of rules.

1 Introduction

Web services are self-describing, open components that support programmatic access to Web accessible data sources and applications. Web services are also poised to become accessible from mobile devices [1], as the proliferation of such devices (e.g., laptops, PDAs, 3G mobile phones) and the deployment of more sophisticated wireless communication infrastructures (e.g., GPRS and UMTS), are empowering the Web with the ability to deliver data and functionality to mobile users. For example, business travelers now expect to be able to access their corporate servers, enterprise portals, e-mail, and other collaboration services while on the move.

However, several obstacles still hinder the seamless provisioning of Web services in wireless environments. Indeed, current Web service provisioning techniques are inappropriate because of the distinguishing features and inherent limitations of wireless environments such as low throughput and poor connectivity of wireless networks, limited computing resources, and frequent disconnections of mobile devices. In addition, the variability in computing resources, display terminal, and communication channel require intelligent support for personalized and timely delivery of relevant data to users [2]. Examples of issues that need to be addressed in order to make the service-oriented computing paradigm of real benefit to mobile users include:

– **Personalized composition of services.** Like their non-mobile counterparts (i.e., stationary users), mobile users also require an integrated access to relevant services. Indeed, the provision of Web services for mobile users tends to be *time* and *location* sensitive, meaning that the mobile users might need to invoke particular services in a certain period and/or a certain place. For example, a student will need the class assistant service only when she is attending a class. Service selection, composition, and orchestration should take in consideration the context of the service provisioning environment (e.g., CPU, bandwidth, state of the user) and the user preferences.
– **Limited resources and wide heterogeneity of mobile devices.** Mobile devices normally posses limited battery power and input capabilities. Therefore, mobile devices are better suited as *passive listeners* (e.g., receiving the service results) than as active tools for service invocation (e.g., searching for the service and sending the request) [3]. Furthermore, the provisioning of services should consider the location of the user. The Web services near her current location should be selected or only the highly customized content should be delivered (e.g., the closest restaurant).
– **Robust service execution.** Numerous situations could prevent a smooth execution of Web services in wireless environments. Indeed, obstacles can range from the dynamic nature of the Web services such as changes of Quality of Service (QoS) to the characteristics of mobile devices like frequent disconnections. We believe that services should be *self-managed* to support their adaptive execution over the Internet. To facilitate the robust execution of services, it is necessary to provide the capabilities for detecting the exceptions at run-time so that appropriate actions can be promptly taken.

The aforementioned challenges call for novel approaches to support dynamic and adaptive Web service provisioning in wireless environments. As a contribution toward this aim, this paper presents the design and implementation of *PCAP*: a framework for **P**ersonalized **C**omposition and **A**daptive **P**rovisioning of Web services. This framework provides a distributed, adaptive, and context-aware infrastructure for personalized composite service provisioning, which takes into account the needs of mobile users. The salient features of PCAP are:

– A personalized composite service specification infrastructure. Using this infrastructure, users specify their needs by reusing and adjusting existing process templates, rather than building their own services from scratch. Users locate process templates and annotate them with contextual information (e.g., execution time/place), thereby defining *personal composite services.*

– A self-managed and adaptive service orchestration model. Participating services and a user agent, a component that acts on behalf of the user, collaborate with each other for the smooth execution of the personalized composite services and interact with the user when and where she decides to do so, achieving the goal of "performing the right task at the right time and at the right place". The knowledge that the participating services and the user agent require is generated based on the context information, the data/control dependencies, and the user preferences. The model is complemented by the fact that user agents and services are able to adapt to runtime exceptions (e.g., service failures) according to exception handling rules.

Section 2 presents the personalized composite service model. Section 3 describes an orchestration model for the distributed execution of personalized composite services, as well as the dynamic exception handling. The PCAP system architecture and its implementation are described in Section 4. Finally, Section 5 provides some concluding remarks.

2 Definition of Personal Composite Services

In this section, we first introduce the modeling of process templates and then describe the configuration of personal composite services.

2.1 Process Templates

Process templates are reusable business process skeletons that are devised to reach particular goals. For example, a class assistant template enables students to manage their class activities by composing multiple services like question posting and consultation booking. We specify process templates with statecharts [4]. It should be noted that the process templates developed in the context of statecharts can be adapted to other process definition languages such as BPEL4WS.

A statechart is made up of states and transitions. The transitions of a statechart are labeled with events, conditions, and assignment operations over process variables. States can be *basic* or *compound.* A basic state (also called *task* in the rest of the paper) corresponds to the execution of a service (which we call a *component service*) or of a member in a *Web service community.* A service community is a collection of Web services with a common functionality but different non-functional properties such as different providers and different QoS parameters. When a community receives a request to execute an operation, the request is delegated to one of its current members based on a selection strategy [5]. Compound states contain one or several statecharts within them. An example will be given in Section 2.3.

In process templates, a task t has a set of input and output parameters. We denote the input (resp., output) as Θ_i (resp., Θ_o) where $\Theta_i(t) = \{i_1, i_2, \ldots, i_m\}$ and $\Theta_o(t) = \{o_1, o_2, \ldots, o_k\}$. The value of a task's input parameter may be:

i) requested from user during task execution, ii) obtained from the user's profile, or iii) obtained as an output of another task. For the first case the following expression is used: i_j:=`USER`. For the other cases, they are expressed as queries: i_j:=Q_j. Queries vary from simple to complex, depending on the application domain and users' needs, and can be expressed using languages like XPath.

In our approach, values that users supply as input parameters are handled differently from the values obtained from user profiles. Indeed, because mobile devices are resource-constrained, values that can be obtained from user profiles should not be requested from users. However, in a process template specification, the template provider only indicates for which input parameters users have to supply a value. It is the responsibility of the user to specify, during the configuration phase, if the value will be provided manually or derived from her profile.

Similarly, the value of a task's output parameter may be: i) sent to other tasks as input parameters, and/or ii) sent to a user in case she wants to know the execution result of the task. Symbol $\rightsquigarrow$ is used to denote the delivery of output parameters. For instance, $o_j \rightsquigarrow$ {`USER`} means the value of o_j should be sent to the user. Note that the value of an output parameter can be submitted to multiple places (e.g., to a task and the user as well). Similar to input parameters, the provider of a process template does not decide which output parameters need to be returned.

2.2 Configuration of Personal Composite Services

Personalization implies making adjustment according to user preferences. Three kinds of user preferences are associated for each process template's task:

- *execution constraints* are divided into *temporal* and *spatial* constraints, which respectively indicate *when* and *where* the user wants to see a task executed,
- *data supply and delivery preferences* are related to supplying values to the input parameters and delivering values of output parameters of the task, and
- *execution policies* are related to the preferences on service selection (for communities) and service migration during the execution of a task.

Temporal and Spatial Constraints. We denote the temporal and spatial constraints of a task t as $\Theta_t(t)$ and $\Theta_s(t)$ respectively. Formally, a temporal constraint is specified as `TMP(op, tm)`, where `op` is a comparison operator (e.g., =, $\leq$, and `between`) and `tm` is either an absolute time, a relative time (e.g., termination time of a task), or a time interval in the form of [tm$_1$, tm$_2$]. `TMP(op, tm)` means that the task can be triggered only if the condition `ct op tm` is evaluated to true. Here, `ct` denotes the system time. For the sake of simplicity, we assume that all temporal values are expressed at the same level of granularity.

Similarly, Θ_s is a spatial constraint specified as `SPL(l)`, meaning that the task can be fired only when the condition `cl IS l` is evaluated to true. `cl` denotes the current location of the user, and `l` is a physical location. A location l_1 is considered the same as another location l_2 if the distance between two points of l_1 and l_2 does not exceed a certain value. We assume that all spatial values are expressed at the same level of granularity.

It should be noted that the temporal (resp., spatial) constraint can be empty, meaning that the corresponding task can be executed at anytime (resp., at anywhere). We also assume that the user's (mobile device) location is collected periodically by our system. In fact, with the advances in positioning technologies such as assisted-GPS (A-GPS) [6], we believe that obtaining mobile users location does not represent an issue anymore.

Data Supply and Delivery Preferences. As stated before, the values of some input parameters of a task can be obtained from a user's profile. The user proceeds in two steps: i) identify which input parameter values can be derived from her profile, and ii) supply the location of the profile and the corresponding attribute names. Similarly, for the output parameters of a task, a user may specify which parameter values need to be delivered to her.

Execution Policies. The execution policies include the *service selection policy* and the *service migration policy*. For a specific task, users can specify how to select a service for this task. The service can be a fixed one (the task always uses this service), or can be selected from a specific service community or a public directory (e.g., UDDI) based on certain criteria (e.g., location of the mobile user). Furthermore, users can specify whether to migrate the services to their mobile devices (e.g., if mobile devices have enough computing resources) or to the sites near the users current location for the execution. Our works on service selection and migration are described elsewhere [7, 8].

2.3 An Example

The example introduced here is inspired by two recent ubiquitous computing applications: UIUC's Gaia [9] and, to a greater extent, UCSD's ActiveClass [10]. ActiveClass is a novel computing application for enhancing participation of students and professors in the classrooms via wireless mobile devices such as PDAs. ActiveClass provides several distinct features including: i) students are encouraged to ask questions anonymously without exposing themselves to the class, thereby avoiding the problems associated with the traditional practice of *raise-hand-up* asking where those students who are diffident are unlikely to ask any questions; ii) professors are able to choose the questions which are worth to be answered; and iii) students can vote the questions asked by other students, which helps the professors to identify the questions of most concern.

Figure 1 is the statechart of a simplified `classAssistant` process template that helps students manage their class activities. In this template, an attendance reminder notifies students about the time and place of the lecture. During the lecture, when a student wants to ask a question, she first browses the questions asked by other students using her PDA. Then she decides either to vote for an already posted question (if her question was already asked by someone else) or to post her question (if no one has asked a similar question). The student may ask several questions during the lecture. After the class, a consultation booking is performed if not all of her questions are answered. In both cases, feedback for the lecture is provided by the student.

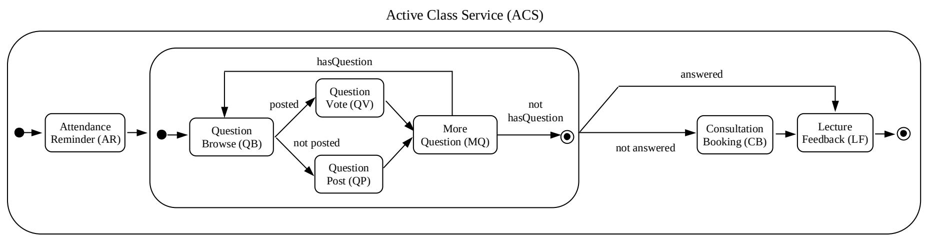

Fig. 1. classAssistant process template

Task	Input Parameters & dependencies	Output Parameters & dependencies

AR	string subjectID:=doc(`PROFILE`)/subject1,
	string studentID:=doc(`PROFILE`)/studentid,
	integer remindTime:=`USER`
		string lectureTime⇝{`USER`},
		string lecturePlace⇝{`USER`},
		string subjectID⇝{QB, QV, QP, CB, LF}
		string studentID⇝{CB}, string professor⇝{CB}

QB	string subjectID=doc(rcv(QB))/subjectID,	XMLDoc questions⇝{`USER`}
QV	string subjectID=doc(rcv(QV))/subjectID, string questionID:=`USER`	XMLDoc voteDetails⇝{`USER`}
QP	string subjectID=doc(rcv(QP))/subjectID, string question:=`USER`	XMLDoc postDetails⇝{`USER`},
MQ	boolean newQuestion:=`USER`	
CB	Date preferredDate:=`USER`, string subjectID=doc(rcv(CB))/subjectID, string professor=doc(rcv(CB))/professor string studentID=doc(rcv(CB))/studentID	Date consultationDate, XMLDoc consultationDetails⇝{`USER`}
LF	string subjectID=doc(rcv(LF))/subjectID, string comments:=`USER`	XMLDoc commentDetails⇝{`USER`}

Table 1. Data dependencies of the `classAssistant` process template (see Figure 1)

Now, assume a student, Andrew, is interested in using the `classAssistant` process template. First, Andrew has to personalize this template by indicating his preferences for each task. For instance, because the lecture will be held from 9am to 11am at `Quad01A`, Andrew sets the temporal and spatial constraints of tasks `QB`, `QV`, `QP`, and `MQ` to be `TMP(between, [9:00 01/01/04, 11:00 01/01/04])` and `SPL(Quad01A)` respectively.

Table 1 describes the input and output parameters of the personal composite service. To describe the data supply and delivery preferences, the following additional notations are used:

– `USER` denotes an end user (e.g., a student), while `PROFILE` denotes the XML document where the user's profile is stored,
– `doc(rcv(CB))/professor` is an XPath query and `rcv(CB)` stands for the XML document that includes the outputs of other tasks received by `CB`. Attribute `professor` is extracted using this query.

The values of some input parameters are supplied by the user. For instance, in order to give lecture feedback (task `LF`), Andrew must input his comments (e.g., `comments`). On the other hand, the value of input parameter `subjectID` of `QB` can be derived from the value of output parameter `subjectID` of `AR`, which in fact, is also used to provide the values of the same input parameter of other tasks (i.e., `QV`, `QP`, `CB`, and `LF`). Further, Andrew specifies that the input parameter `studentID` and `subjectID` of task `AR` will be derived from his profile. Andrew also would like to receive the detailed result (e.g., `postDetails`) of each task. It should be noted that there are six conditions in the statechart transitions. Conditions are modeled as boolean variables, whose values are provided by the user at runtime.

3 Personal Composite Service Orchestration

During the execution of a composite service, the involved component services need to coordinate with each other and with the client device in order to ensure that the business logic of the composite service is enforced. This process is often termed *orchestration*. Existing orchestration models [11, 12] assume that the connection between the central scheduler and the component services is continuously available, and that it has the same characteristics (e.g., bandwidth, reliability) as a connection between two component services. This assumption is not valid in the case of personal composite services, where executions are initiated and followed up by, and specifically for, a given (possibly mobile) client. Accordingly, we advocate that in order to achieve robust and smooth execution of personal composite services in mobile environments, these composite services should be *self-orchestrating*: they should be able to coordinate their actions in an autonomous way, without having to continuously synchronize with the client, which could lead to idle periods and timeouts due to disconnections.

In our approach, *self-orchestration* is achieved by encoding the interactions between services in the form of *control tuples* which are placed in and retrieved from *tuple spaces*. The tuple space model has its origins in the Linda [13] language for distributed programming and is recognized as an attractive model for managing interactions among loosely coupled entities in wireless environments [2]. Tuple spaces have the advantage of providing direct support for *pull-based asynchronous interactions* in which the “sender” (e.g. the client device) and the “receiver” (e.g. a component service) are separated in both space and time. This enables mobile users to disconnect at any time, and re-synchronize with the underlying infrastructure upon reconnection.

3.1 Control Tuples and Compound Transitions

In this section, we define two concepts used in the rest of the paper: *control tuple* and *compound transition*.

Definition 1. *A control tuple is a rule of the form Event-Condition-Action (*`E[C]A`*) where:*

- `E` *is a conjunction of execution events. The conjunction of two events* e_1 *and* e_2 *is denoted as* $e_1 \wedge e_2$ *and the semantics is that if an occurrence of* e_1 *and an occurrence of* e_2 *are registered in any order, then an occurrence of* $e_1 \wedge e_2$ *is generated.*
- `C` *is a conjunction of conditions on execution states including event parameter values and service information (e.g., inputs and outputs of tasks).*

– `A` *is a sequence of execution actions* $a_1; a_2; \ldots; a_n$. *The actions are executed in the order specified by the sequence. Some selected events and actions supported in our approach are given in Table 2.* □

As discussed earlier, a statechart can have compound states and therefore, there can be multiple direct and indirect ways of transitioning from a given basic state to another one. In other words, when exiting a given state, there are a number of transitions that can be taken, some of which are simple (e.g., the transition between `QV` and `MQ` in Figure 1) and others are compound (e.g., the transition between `MQ` and `LF`). Hence, in order to determine how to route control-flow notifications and data items between basic states, we need to introduce a concept of *compound transition.*

Definition 2. *A compound transition CT is a sequence of transitions* tr_1, tr_2, *...,* tr_n *belonging to a given statechart, such that:*

– *source(*tr_1*)*[5] *is a basic state,*
– *target(*tr_n*) is a basic state, and*
– *for all i in [1..n-1], either target(*tr_i*) is the final state of a region belonging to the compound state source(*tr_{i+1}*), or source(*tr_{i+1}*) is the initial state of a region belonging to the compound state target(*tr_i*).* □

3.2 Service Orchestration Tuples in PCAP

In this section, we present four types of control tuples used to coordinate personal composite service executions in PCAP, namely *precondition tuples*, *postprocessing tuples*, *context awareness tuples*, and *exception handling tuples.*

Precondition and Postprocessing Tuples. Determining when should a given task be activated during the execution of a personal composite service requires answering the following questions: i) what are the *preconditions* for executing this task? and ii) once an execution of this task is completed, which entities (e.g., other tasks or the user agent) need to be notified of this completion? The knowledge needed to determine the moment(s) of activation of a task during an execution of a personal composite service can therefore be captured by two sets: i) a set of *preconditions* to be checked before the task is executed; and ii) a set of so-called *postprocessing actions* capturing which other tasks may need to be notified when the execution of a given task is completed. Below, we provide formal definitions of the concepts of precondition and postprocessing of a task.

Definition 3. *The precondition of task t of a personal composite service S is a set of control tuples such that:*

[5] Here, source(tr) denotes the source state of transition tr, while target(tr) denotes the target state of tr.

◦ Events and Descriptions
entered(location *l*): the user has entered the location *l*.
disconnected(device *d*): the device *d* has disconnected. For example, the user's mobile device may be switched off or can not be reached in an uncovered area.
unpresentable(serviceResult *r*, device *d*): the service result *r* is evaluated to be unpresentable in the user's device *d* due to the limited capabilities of the device.
QoSDegraded(service *s*, QoS *q*): the QoS *q* of service *s* has deteriorated. For example, the execution time of the service becomes longer.
◦ Actions and Descriptions
notify(task *t*): send a notification of completion to task *t*.
transform(serviceResult *r*, tranformService *s*, device *d*): transform service result *r* using the transformation service *s* according to the capabilities of the user's device *d*.
informNewLocation(location *newL*, serviceSet *SC*): inform the location *newL* of a user to the relevant Web services *SC*.
reassign(service *s*): delegate the invocation of a service to service *s*.

Table 2. Selected events and actions supported in PCAP

- `E` *is a conjunction of events of the form* `ready(`$\Theta_i(t)$`)` *and* `completed(`t'`)` *where* t' *is a task for which there exists a compound transition* `CT` *such that source(*`CT`*)=*t' *and target(*`CT`*)=*t*. The event* `completed(`t'`)` *is raised when a notification of completion is received from the controller of* t'*.*
- `C` *is a conjunction of temporal and spatial constraints of* t*,* $\Theta_t(t)$ `and` $\Theta_s(t)$*. If* t *does not have any constraint,* `C` *is interpreted as* `true`*.*
- `A` *is an execution action* `execute(`t`)`*, which invokes the task* t*.* □

In Andrew's classAssistant service (see Section 2), the precondition of `CB` is expressed as: {`ready(`Θ_i`(CB))`$\wedge$`completed(MQ)[TMP(`$\geq$`, 11:10 01/01/04)]execute(CB)`}, where Θ_i`(CB)` is the set of the input parameters of task `CB`. This tuple indicates that when all the values of input parameters of `CB` are available and `MQ` is finished, if current time is later than `11:10 01/01/04`, the invocation of `CB` will start. Note that Andrew did not specify a spatial constraint for `CB`.

Definition 4. *The postprocessing of* t *of* S *is a set of control tuples such that:*

- `E` *is an event of the form* `completed(`t`)`*. The event is generated when the execution of task* t *is completed.*
- *There exists a compound transition* `CT` *where source(*`CT`*)=*t *and target(*`CT`*)=*t'*.*
- `C` *is Cond(*`CT`*), which is the conjunction of the conditions labeling transitions of* `CT` *(i.e.,* tr_1*,* tr_2*, ...,* tr_n*), expressed as* $c_1 \wedge c_2 \wedge \cdots \wedge c_n$*, where* c_i *is the condition labeling transition* tr_i*.*
- `A` *is a set of actions of the form* `notify(`t'`)` *and* `sendResult(`o`, `r`)`*, where* o *is an output parameter whose value needs to be delivered to a receiver* r*, which could be the user or another task of* S*.* □

In the example, the postprocessing of task `QP` is: {`completed(QP)[true]notify(MQ);sendResult(postDetails, Andrew)`}. This tuple indicates that when the execution of `QP` is completed, task `MQ` should be notified of this completion and the value of output parameter `postDetails` should be sent to Andrew.

The concepts of precondition and postprocessing of a task as defined above possess two advantageous features. Firstly, the knowledge (i.e., precondition and postprocessing) of the execution of a task is expressed in the form of tuples, which provide the possibility to store and operate the knowledge using powerful coordination models such as tuple spaces. Life cycle information can be associated with tuples, indicating how long a tuple should be made available in the tuple space. Thus, the potential overhead of tuple space can be avoided because the tuples will be removed automatically when they expire. Secondly, the design of precondition and postprocessing tuples considers both the control flow and data dependencies of the personal composite services. In particular, when the execution of a task is completed, only the output parameters whose values are needed by other entities (e.g., the user or other tasks of the composite services) are dispatched.

Context Awareness Tuples. There are two major pieces of context information relevant for the execution of personal composite services: *current time* and user's *current location*. It is assumed that the current time is known by all the entities participating in execution (i.e., derived from their system clock). The user's location, on the other hand, is only known and maintained by the *user agent*: a component whose role is to facilitate the orchestration of personal composite services on behalf of mobile users. In order to achieve context awareness, and in particular, to take into account the above two pieces of context information, control tuples encoding context awareness rules are placed in the user agent's tuple space at the beginning and during the composite service execution. For example, the following control tuple can be placed in the user agent's tuple space to capture the fact that when the user agent detects that the user enters a given location, this location needs to be communicated to the services participating in a personal composite service execution: `entered(`$newL$`)[true]informNewLocation(`$newL$`,` SC`)`, where $newL$ is the new location of the mobile user and SC is the set of involved Web services.

Exception Handling Tuples. There are numerous situations that could prevent a smooth execution of a personal composite service. Indeed, obstacles are multiple, ranging from the dynamic nature of the Web to the reduced capabilities of mobile devices. To support adaptive execution of personal composite services over the Internet, services should be *pro-active*: they should be capable of adapting themselves in reaction to run-time exceptions.

We distinguish two levels of exceptions: *user level* and *service level*. The user level exceptions are due to the characteristics of mobile devices (e.g., display size) or changes of the personal composite services launched by users. A mobile device can be disconnected due to discharged battery, alignment of antennas, or lack of coverage area. As a result, service results can not be delivered to the user. Further, a service result may not be able to be displayed on a mobile device because of lack of appropriate facilities (e.g., device cannot display graphics). Other exceptions at the user level are changes of personal composite services. For example, Andrew may want to change his preferences on a specific task (e.g., spatial constraint of `QB`, `QV`, and `QP`) because the lecture room is rescheduled (e.g., the lecture will be held at `Quad01C` instead of `Quad01A`).

During a service execution, different exceptions can occur. In particular, the selected Web service that executes a task of a personal composite service may become unavailable because it is overloaded or its respective server is down. The QoS parameters[6] of a service may be changed (e.g., the service provider increased the execution price). For a specific task of a personal composite service, some new Web services with better QoS may become available.

An *exception handling tuple* acts as an instruction to execute actions if specific *exception events* occur and specific conditions hold. Exception events are generated in response to changes of service execution states. Examples of such events are: mobile device disconnection, services failure, and violation of QoS constraints. The following is an example of exception handling tuples:

– `arrived(a)`$\wedge$`unpresentable(r,d)[true]transform(r,TS,d)`, where `TS` is a transformation service and `a` is the user agent. Note that the description of transformation services is outside the scope of this paper for space reasons. This rule, specified by the user, indicates that if the result `r` of a service can not be displayed in the user's current device `d`, the result will be sent to service `TS` for adaptation before being forwarded to `d`.

3.3 Control Tuples Generation

The creation of control tuples of a personal composite service occurs at various stages. First, the process template designer defines control tuples at design time to capture failure handling policies (see Section 3.2) and other behavior which cannot be personalized by the user. The tuples created in this way are injected into the control tuple spaces of the relevant entities (e.g., component services, service communities) before the process template is made available to users. Later, when a process template is personalized and the resulting personal composite service is executed, the user agent automatically generates and injects control tuples (i.e., precondition and postprocessing tuples) from the preferences specified by the user and the information encoded in the process template. Finally, once the personalized service is being executed, the user agent keeps adding tuples into the tuple spaces of the participating services, according to the information that it receives from the user, and the tuples (e.g., context awareness tuples) that exist in its own tuple space (a user agent has its own tuple space).

The generation of precondition and postprocessing tuples of each task of a personal composite service is complex and challenging because the information encoded in the statechart (e.g., control flow and data dependencies) of the personal composite service needs to be extracted and analyzed. In what follows, we therefore describe the algorithms for the generation of postprocessing and precondition tuples.

The algorithm for generating postprocessing tuples (namely `PostProc`) for a task takes as input a task t, and produces a set of postprocessing tuples. The algorithm analyses the data dependencies of the output parameters (`OD`) and

[6] Detailed description of QoS parameters can be found in [8].

the outgoing transitions of t (`TR`). From `OD`, a set of actions is created indicating which outputs should be delivered to which receivers. The postprocessing set of t is the union of the postprocessing tuples associated to `TR`.

The postprocessing tuples for each outgoing transition of a task are generated by a function named `PostProcT`, which takes as input a transition tr, and returns a set of postprocessing tuples including the postprocessing actions associated with this transition. Various cases exist. When tr leads to a basic state (say t'), the tuple `completed(source(`tr`))[c]notify(`t'`)` is created, meaning that after the execution of the task is completed, if the condition `c` is true, a notification must be sent to t'. If tr points to a compound state, one postprocessing tuple is generated for each initial transition of this compound state. Finally, if tr points to a final state of a compound state, the outgoing transitions of this compound state are considered in turn, and one or several postprocessing tuples are produced for each of them.

Similarly, the algorithm for generating precondition tuples of a task (namely `PreCond`) relies on the personalized attributes of the task (e.g., temporal and spatial constraints, input parameters), and control flows associated with the task. The task's *incoming* transitions are analyzed and the precondition is generated for each incoming transition of the task. For space reasons, we omit the description of the algorithm.

4 Implementing PCAP

In this section, we overview the status of the PCAP prototype implementation. The prototype architecture (see Figure 2) consists of a *user agent*, a *process manager*, and a *pool of services*, all implemented in Java. Below, we describe the implementation of the process definition environment (also called the process manager) for specifying and managing process templates, and a set of pre-built classes that act as a middleware for enabling the self-orchestration of personal composite services.

4.1 Process Definition Environment

The process definition environment consists of a set of integrated tools that allow template providers to create process templates: *templates/service discovery engine* and *process template builder*. The process template builder assists template providers in defining the new templates and editing the existing ones. A template definition is edited through a visual interface. The template builder offers an editor for describing a statechart diagram of a process template (an extension of our previous work in [14]).

The template/service discovery engine facilitates the advertisement and location of processes and services. In the implementation, the Universal Description, Discovery and Integration (UDDI) is used as process/service template repository. Web service providers can also register their services to the discovery engine. The Web Service Description Language (WSDL) is used to specify Web services.

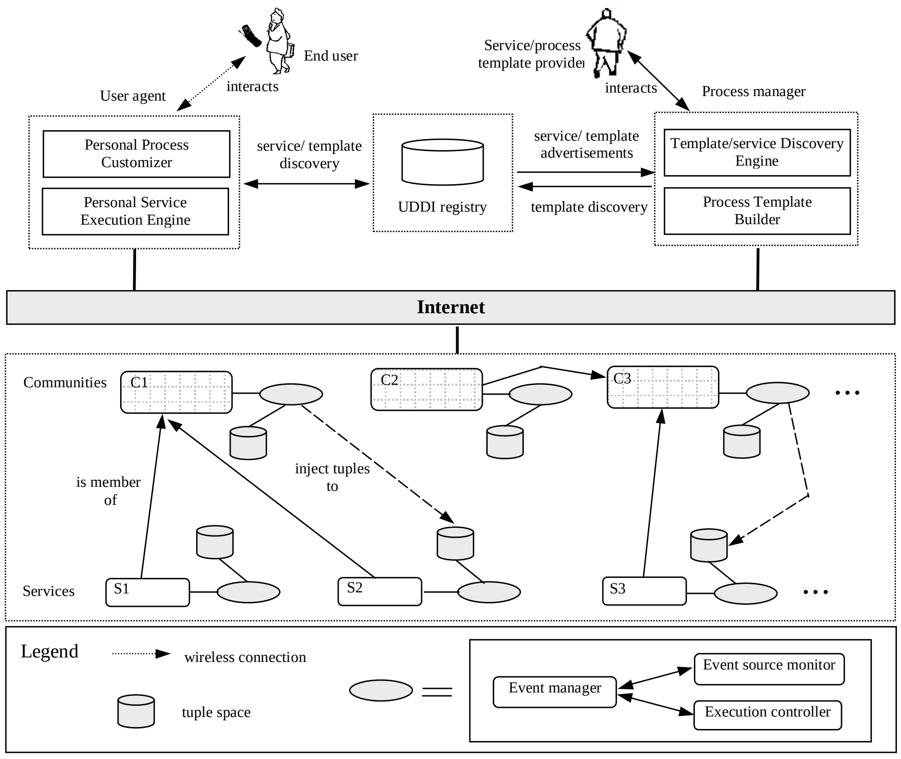

Fig. 2. Architecture of PCAP prototype

4.2 Pre-built Classes

For any user (resp., service) wishing to participate in our platform, the user (resp., the administrator of the service) needs to download and install a set of pre-built class, namely the *user agent* (resp., the *task controller*), the *event manager*, and the *event source monitor*.

User Agent. User agents are used for specifying and executing personal composite services. A user agent consists of a *personal process customizer*, and an *execution engine*. The process customizer generates personal services for users by customizing process templates based on user preferences. It provides an interface for the mobile user, implemented using Pocket PC Emulator. Currently, kXML 2 (`http://kxml.enhydra.org`), an open source designed to run in an MIDP (Mobile Information Device Profile) environment (e.g., PalmOS), is used to parse XML documents on mobile devices. The personalized service is then translated into an XML document for subsequent analysis and processing by the execution engine. The user agent maintains the user profile. An interface is also provided for the mobile user to create and modify her profile information.

The execution engine provides a method called `deploy()` that is responsible for generating control tuples of each task of a personal composite service, using the approach presented in Section 3. The control tuples are then uploaded into the tuple spaces of the corresponding selected Web services, as well as into the tuple space of the user agent (not shown in the architecture).

Task Controller. The functionalities of a task controller are realized by a pre-built Java class, called `Controller`. This class provides services with capabilities to participate in service management including exception handling. It provides a method called `coordinate()` for receiving messages, managing service instances (i.e., creating and deleting instances), registering events to the event manager, triggering actions, tracing service invocations, and communicating with other controllers. There is one controller class per service. The controller relies on the control tuple space of the associated service to manage service activities. Each tuple space is represented using a local XML repository. Controllers monitor and control service activities by creating and reading tuples of their associated space as well as injecting (uploading) tuples in spaces associated to their peers.

Event Manager and Event Source Monitor. The event manager and the event source monitor are attached to a service or the user agent. The event source monitor detects the modifications of the *event sources*. For example, the event source of the event `entered` is the mobile user's current location. The event manager fires and distributes the events. These two components are mapped into classes called `eventManager` and `eventSourceMonitor` respectively. The `eventManager` provides methods for receiving messages, including subscribing messages from controllers and event source information from `eventSourceMonitor`, and notifying the fired events to the subscribed controllers. In particular, the class `eventManager` implements a process that runs continuously, listening to incoming messages. The messages are either `subscribe` or `monitor`. The former are the messages for subscribing to events, while the latter are the messages notifying the detected event source information. When the `eventManager` receives a message, it first examines the identifier of the message and proceeds as follows: i) if it is a subscribing message, extracts the controller and the subscribed event, then add the controller to the array of the event, which maintains all the subscribers of this event, ii) if it is a monitor message, extract the event source information and fire the corresponding event.

5 Conclusion

In this paper, we have presented the design and implementation of PCAP, a framework of enabling personalized composition and adaptive provisioning of Web services. While much of the work on Web services has focused on low-level standards for publishing, discovering, and provisioning Web services in wired environments and for the benefit of stationary users, we deemed appropriate to put forward novel solutions and alternatives for the benefit of mobile users. Main contributions of PCAP include: i) personalized composition of Web services by considering users' preferences (e.g., temporal and spatial constraints), ii) distributed execution of personal composite services that is coordinated by tuple space based orchestration model, and iii) run-time exceptions handling.

So far, we have implemented a prototype that realizes the specification and execution of personal composite services. This implementation effort has served to validate the viability of the proposed approach. Ongoing work includes assessing the performance and scalability of PCAP. Another direction for future work is to add more flexibility to PCAP (beyond its exception handling capability) by supporting runtime modifications to the schema of a personal composite service (e.g., removing a task).

References

1. Chen, Y., Petrie, C.: Ubiquitous Mobile Computing. IEEE Internet Computing **7** (2003) 16–17
2. Mascolo, C., Capra, L., Emmerich, W.: Mobile Computing Middleware (A Survey). In: Advanced Lectures on Networking (NETWORKING 2002), Pisa, Italy (2002)
3. Burcea, I., Jacobsen, H.A.: L-ToPSS-Push-oriented Location Based Services. In: Proc. of the 4th VLDB Workshop on Technologies for E-Services (VLDB-TES03), Berlin, Germany (2003)
4. Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM Transactions on Software Engineering and Methodology **5** (1996) 293–333
5. Benatallah, B., Sheng, Q.Z., Dumas, M.: The Self-Serv Environment for Web Services Composition. IEEE Internet Computing **7** (2003) 40–48
6. Giaglis, G., Kourouthanasis, P., Tsamakos, A.: Towards a Classification Network for Mobile Location Services. In Mennecke, B., Strader, T., eds.: Mobile Commerce: Technology, Theory, and Applications. Idea Group Publishing (2002)
7. Maamar, Z., Sheng, Q.Z., Benatallah, B.: On Composite Web Services Provisioning in an Environment of Fixed and Mobile Computing Resources. Information Technology and Management Journal, Special Issue on Workflow and e-Business, Kluwer Academic Publishers (forthcoming) **5** (2004)
8. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality Driven Web Services Composition. In: Proc. of the 12th International World Wide Web Conference (WWW'03), Budapest, Hungary (2003)
9. Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrstedt, K.: A Middleware Infrastructure for Active Spaces. IEEE Pervasive Computing **1** (2002) 74–83
10. Griswold, W.G., Boyer, R., Brown, S.W., Truong, T.: A Component Architecture for an Extensible, Highly Integrated Context-Aware Computing Infrastructure. In: Proc. of the 25th International Conference on Software Engineering, Oregon, Portland (2003)
11. Schuster, H., Georgakopoulos, D., Cichocki, A., Baker, D.: Modeling and composing service-based and reference process-based multi-enterprise processes. In: Proc. of the 12th Int. Conference on Advanced Information Systems Engineering (CAiSE'00), Stockholm, Sweden, Springer Verlag (2000)
12. Casati, F., Shan, M.C.: Dynamic and adaptive composition of e-services. Information Systems **26** (2001) 143–162
13. Ahuja, S., Carriero, N., Gelernter, D.: Linda and Friends. Computer **19** (1986) 26–34
14. Sheng, Q.Z., Benatallah, B., Dumas, M., Mak, E.: SELF-SERV: A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment. In: Proc. of the 28th Very Large DataBase Conference (VLDB'02), Hong Kong, China (2002)