Configurable Composition and Adaptive Provisioning of Web Services

Quan Z. Sheng, *Member*, *IEEE*, Boualem Benatallah, *Member*, *IEEE*, Zakaria Maamar, and Anne H.H. Ngu

Abstract—Web services composition has been an active research area over the last few years. However, the technology is still not mature yet and several research issues need to be addressed. In this paper, we describe the design of CCAP, a system that provides tools for adaptive service composition and provisioning. We introduce a composition model where service context and exceptions are configurable to accommodate needs of different users. This allows for reusability of a service in different contexts and achieves a level of adaptiveness and contextualization without recoding and recompiling of the overall composed services. The execution semantics of the adaptive composite service is provided by an event-driven model. This execution model is based on Linda Tuple Spaces and supports real-time and asynchronous communication between services. Three core services, coordination service, context service, and event service, are implemented to automatically schedule and execute the component services, and adapt to user configured exceptions and contexts at run time. The proposed system provides an efficient and flexible support for specifying, deploying, and accessing adaptive composite services. We demonstrate the benefits of our system by conducting usability and performance studies.

Index Terms—Web service, service composition, service-oriented architecture, exception handling, event-based service execution.

1 Introduction

Web services, and more in general Service-Oriented Architectures (SOAs), are gathering a considerable momentum as the technologies of choice to implement distributed systems and perform application integration. Although tremendous efforts and results have been made and obtained in Web service composition area [1], [2], [3], [4], the technology is still not mature yet and requires significant efforts in some open research areas [4], [5], [6], [7], [8].

A current trend is to provide *adaptive* service composition and provisioning solutions that offer better quality of composite Web services [5], [6], [8], [9], [10], [11]. The pervasiveness of the Internet and the proliferation of interconnected computing devices (e.g., laptops, PDAs, 3G mobile phones) offer the technical possibilities to interact with services anytime and anywhere. 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. Since the contexts of users, either human beings or enterprises, are varied, it is essential that service composition embraces a *configurable* and *adaptive* service provisioning approach (e.g., delivering the right service in the right place at the right time). Configuration allows the same service to be reused in different contexts without low-level recoding and recompilation of the service.

We have developed the CCAP system (*C*onfigurable *C*omposition and *A*daptive *P*rovisioning of composite services) [8], [12] that provides a system infrastructure for distributed, adaptive, context-aware provisioning of composite Web services. CCAP is a fully functional prototype system that allows service designers to focus more on specifying service composition requirements at a high level of abstraction such as business logic of applications, generic exception handling policies, and contextual constraints, rather than on low-level deployment and coordination concerns. The innovative aspect of our work is the composition model that provides distinct abstractions for service context and exceptions, which can be embedded or plugged into the process schemas through simple interaction with end users. This differs from existing service composition approaches, which rely on scripting languages or process modeling notations (e.g., state diagrams, Petri nets, process algebra) to specify low-level exception handling and contextual awareness, resulting in monolithic composite services that are unintuitive, hard to reuse, and maintain [13]. The salient features and contributions of CCAP are as follows:

- A configurable service composition model that decouples the contextual and exceptions specification from the business logic of the actual service. We use the concept of *process schema*, which is a reusable and extensible business process template devised to reach a particular goal for the modeling of the business logic of the composite service. End users can then customize this process schema by assigning users' contextual constraints to the process schema. This avoids hard-coding users' contextual

- *Q.Z. Sheng is with the School of Computer Science, the University of Adelaide, Adelaide, SA 5005, Australia. E-mail: qsheng@cs.adelaide.edu.au.*
- *B. Benatallah is with the School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia. E-mail: boualem@cse.unsw.edu.au.*
- *Z. Maamar is with the College of Information Technology, Zayed University, PO Box 19282, Dubai, United Arab Emirates. E-mail: zakaria.maamar@zu.ac.ae.*
- *A.H.H. Ngu is with the Department of Computer Science, Texas State University, TX 78666-4616. E-mail: angu@txstate.edu.*

Manuscript received 13 Feb. 2008; revised 4 Nov. 2008; accepted 22 Dec. 2008; published online 15 Jan. 2009.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSC-2008-02-0009.
Digital Object Identifier no. 10.1109/TSC.2009.1.

constraints as control flow in the service which will quickly lead to an overly complex process schema that is hard to understand, reuse, maintain, and schedule. Generic exception behaviors (e.g., service failures, network errors) that may happen during the service provisioning are specified as policies that can be reused across different process schemas. By adopting *policy-based* approach for modeling exceptions, end users can dynamically add, remove, and modify policies on how exceptions can be handled without changing composite service functionalities.

- An event-driven service execution model that provides the execution semantics of adaptive composite services. The most significant benefit of translating process models into an event-based model, by means of *control tuples* in the form of Event-Condition-Action (ECA) rules, is what we called *knowledge independence*, in the sense that control tuples are stored separately from a composite Web service specification. This provides the possibility of: 1) distributing control tuples to participating Web services, thereby realizing a fully distributed execution of composite services and 2) adding or removing control tuples to overlay behavior on top of already *deployed* composite services in response to unforeseen situations or particular requirements. The execution semantics is enforced by means of three core generic services: *coordination service*, *context service*, and *event service*. These basic infrastructure form the backbone of a middleware that provides deployment and automatic execution of the adaptive composite service in a robust and scalable manner.
- A user-based service composition evaluation. CCAP has been implemented using a number of state-of-the-art technologies. To validate the feasibility and benefits of our approach, we not only evaluated the system performance such as scalability and adaptability, but conducted a *usability study* to evaluate the learnability, efficiency, and user adoption of our approach. We presented the system to a number of volunteers, asked them to use the system, and to report their experience by answering a questionnaire. To the best of our knowledge, this is one of the few works that provide evaluations of Web services composition approaches using real users.

The remainder of the paper is organized as follows: Section 2 overviews our configurable service composition model and Section 3 elaborates the proposed configuration approach including user contexts and exception handling policies. Section 4 gives details about our event-driven composite service execution model and the algorithms for generating control tuples from composite service definitions. Section 5 focuses on the implementation, usability study, and performance evaluation of the proposed system. Finally, Section 6 overviews related work and Section 7 provides some concluding remarks.

2 Configurable Composition Model

Traditional service composition model typically relies on process modeling notations to choreograph the component services. Service designers statically design and chart every aspects of the composite service. This mode of composition requires the context that the service will be executed in specified a priori. It also requires *exhaustive* enumeration of all the possible exceptions in the composed services if robust execution is desired. However, the environments that users interact with are *dynamic* by nature (e.g., changing locations, changing environmental factors, changes in services, and evolving user requirements). It is not possible to enumerate all the possible contexts and exceptions at service design time. This calls for more agile approaches where services can be composed from reusable process template that can be configured to accommodate the slight variation needs of every individual user in different contexts, and can be transparently adapted to changes with minimal or no user intervention.

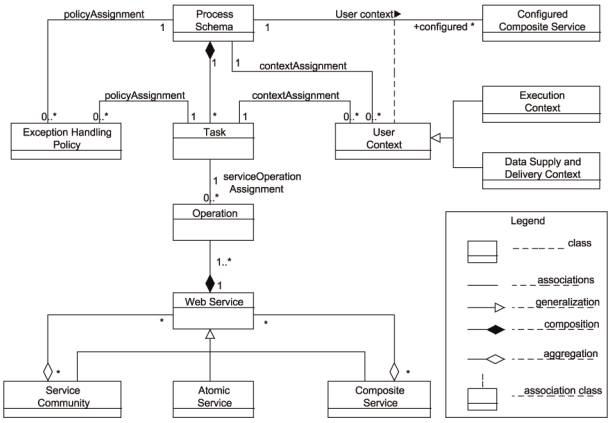

Fig. 1. UML class diagram for configurable service composition model.

2.1 Design Overview

Fig. 1 shows our configurable and adaptive service composition model, described in Unified Modeling Language (UML) [14]. A *process schema* is a reusable and extensible business process template devised to reach a particular goal (e.g., travel planning). Each process schema has one or more tasks and each task belongs to exactly one process schema. The relation is denoted by a composite aggregation (i.e., the association end with a filled diamond). Each task is associated with a service operation where the service can be either an atomic service, a composite service, or a service community (details will be introduced in Section 2.2).

A process schema can be *configured* by assigning a number of *user contexts* to the process schema's tasks and the schema itself. The relation is denoted as an association class (dashed line). We distinguish two main kinds of user contexts, namely *execution contexts* and *data supply and delivery contexts*. Process schemas allow tasks to be parameterized, i.e., individual users can customize process schemas to meet their particular requirements by assigning contexts to those parameters.

A process schema can also be configured to handle particular type of exceptions. An *exception handling policy* is a rule that prescribes the knowledge on the appropriate response to a particular execution exception. Policies are assigned to the process schema and its tasks by the relation *policyAssignment*, indicating which task (process schema) has

what kinds of exception handling policies. This assignment is done at design time by the service designer. A task (process schema) can have multiple exception handling policies. End user can modify the assigned policies by adding, changing, and deleting the exception handling policies.

In the rest of this section, we will introduce the process schema model. Details on configuring process schemas (i.e., user context model and exception handling model) will be described separately in Section 3.

2.2 Process Schema

A *process schema* is an umbrella structure that aggregates other atomic or composite services. Following our previous work [15], process schemas are modeled as statecharts [16]. It is worth noting that the process schemas developed in statecharts can be mapped onto other process definition languages such as BPEL [17].

A statechart is made up of states and transitions. States can be *initial*, *end*, *basic*, or *compound*. A basic state (also called *task*) corresponds to an invocation of a service operation, whether an atomic service, a composite service, or a *community*. The concept of *Web service community* [15] is proposed to handle the large number and dynamic nature of Web services (e.g., emergence of new services and retraction of old ones) in a flexible way. A service community is a collection of Web services with a common functionality but different nonfunctional properties such as different providers and different Quality of Service (QoS) parameters (e.g., reliability). Service communities provide descriptions of a desired functionality without referring to any potential service. When a community receives a request to execute an operation, the request is delegated to one of its current members based on selection strategies [15].

Compound states provide a mechanism for nesting one or several statecharts inside a (larger) statechart. This is a useful abstraction that is needed to nest subprocesses within a process. There are two types of compound states: OR and AND states. An OR state contains a single statechart whereas an AND state contains several statecharts (separated by dashed lines). OR states are used as a decomposition mechanism for modularity purposes, while AND states are used to express concurrency.

Fig. 2 is the statechart of a simplified process schema of the digital class assistant [8] that helps students manage their class activities. In this example, an attendance reminder notifies students about the lecture's time and place. The task is followed by an AND state, in which an attendance guide is performed in parallel with the outlining of the lecture notes. The former provides a detailed guidance on how to get to the lecture room from a student's current location, while the latter searches the lecture note slides and provides the student an outline 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 a posted question (if her question was already asked by someone) 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 about the lecture is provided by the student.

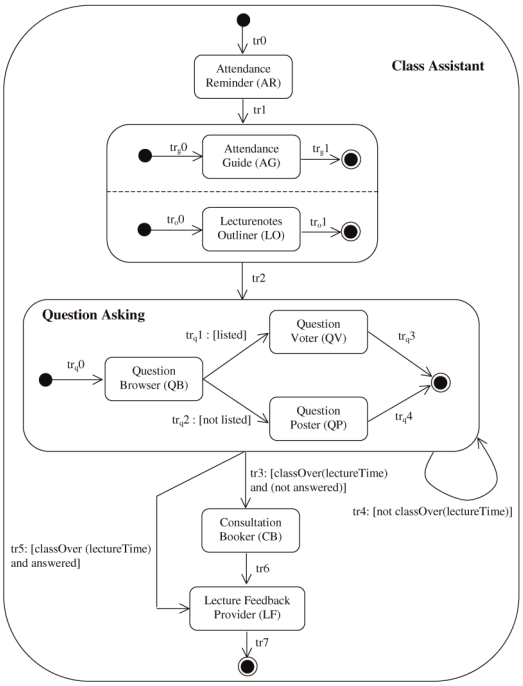

Fig. 2. The class assistant process schema.

2.2.1 Data Dependencies

In a process schema, each task t has a set of input and output parameters. We denote its input and output as $\Theta_i = \{i_1, i_2, \ldots, i_m\}$ and $\Theta_o = \{o_1, o_2, \ldots, o_k\}$, respectively. The value of a task's input parameter may be: 1) requested from user during task execution, 2) obtained from user profile, or 3) obtained as an output of another task. For the first case, the following expression is used: $i_j := \texttt{USER}$. For the other cases, they are expressed as queries: $i_j := \mathcal{Q}_j$. Queries vary from simple to complex, depending on the application domain and user needs. Queries can be expressed using languages like XPath [18].

During the process schema specification, the service designer only indicates which input parameters end users have to supply values. It is the responsibility of an end user to specify, during the configuration phase, if the value will be provided manually or extracted from her profile.

Similarly, the value of a task's output parameter may be: 1) sent to other tasks as input parameters and/or 2) sent to an end user in case she wants to know the execution result of the task. Symbol $\rightarrow$ denotes the delivery of output parameters. For instance, $o_j \rightarrow \texttt{USER}$ means that the value of o_j should be sent to the end user. Note that the value of an output parameter can be submitted to multiple places (e.g., a task and the user). At process schema specification time, the service designer only indicates which output parameters require end users attention.

Table 1 describes the input and output parameters of the class assistant process schema, together with their data dependencies. To describe data dependencies, the following notations are used: 1) `USER` denotes an end user (e.g., a student) and 2) `doc(rcv(QB))/subjectID` is an XPath query where `rcv(QB)` stands for the XML document that includes the outputs of other tasks received by

TABLE 1
Data Dependencies of the Class Assistant Process Schema

Task	Input parameter & dependency	Output parameter & dependency
AR	string subjectID:=USER, string studentID:=USER	string lectureTime→{USER}, string lecturePlace→ {AG, USER}, string subjectID→{LO, QB, QV, QP, CB, LF}, string studentID→{CB}, string professor→{CB}
AG	string lectureP=doc(rcv(AG))/lecturePlace, string studentID=doc(rcv(AG))/studentID	XMLDoc guideDetails
LO	string subjectID=doc(rcv(LO))/subjectID	XMLDoc lectureOutline
QB	string subjectID=doc(rcv(QB))/subjectID	
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}
CB	Date preferredDate:=USER, string subjectID=doc(rcv(CB))/subjectID, string professor=doc(rcv(CB))/professor, string studentID=doc(rcv(CB))/studentID	XMLDoc consultBookingDetails→{USER}
LF	string subjectID=doc(rcv(LF))/subjectID, string comments:=USER	XMLDoc commentDetails→{USER}

QB. A similar explanation applies to other XPath queries given in the table.

Control flow between states are modeled as transitions. There are five conditional transitions in the process schema shown in Fig. 2. Conditional transitions are modeled as Boolean variables whose values are provided by an end user at runtime, or as Boolean functions that take as parameters queries involving input and output parameters of tasks. For example, the condition `classOver(lectureTime)` is a function call whose parameter is directly obtained from one of the outputs of task AR. Meanwhile, `listed` and `answered` are two Boolean variables.

3 Configuring Process Schemas

Process schemas that correspond to recurrent user needs (e.g., booking rooms, financial planning, travel planning) are defined by service designers based on, for example, common usage patterns, and are stored in schema repositories. These schemas specify tasks (e.g., booking a flight), data/control-flow dependencies between tasks, and generic exception handling policies. Once defined and deployed as services, end users can locate, configure with their contexts, and execute them. The configuration is done by specifying a number of *user contexts*[1] and *exception handling policies* and assigning them to the process schema's tasks and/or the process schema.

3.1 User Context

Within current UML statecharts model, there is no provision for modeling and managing user contexts. Typically, user contexts are either coded as specific tasks in the process schema or as special conditional control flow. The intermix of user context and business logic in a process schema quickly leads to an overly complex process schema that is hard to reuse, maintain, and schedule. We provide the following two abstractions for modeling of user contexts in CCAP: *execution contexts* and *data supply and delivery contexts*.

3.1.1 Execution Context

An execution context specifies that certain conditions must be met in order to perform a particular operation. Formally, an execution context is modeled as a predicate (i.e., Boolean function) that consists of an operator and two or more operands. While the concept of execution context is generic enough to be applicable in specifying a wide range of contextual constraints, we focus on two constraints in our service composition model: *temporal* and *spatial* constraints:

- *Temporal constraint* $\Theta_\tau(t)$: Formally, a temporal constraint is specified as $\Upsilon(\mathcal{P}, \mathcal{T})$, where $\mathcal{P}$ is a comparison operator (e.g., $=$, $\leq$, and $between$) and $\mathcal{T}$ is either an absolute time, a relative time (e.g., termination time of a task), or a time interval in the form of $[\mathcal{T}_1, \mathcal{T}_2]$. The $\Upsilon(\mathcal{P}, \mathcal{T})$ constraint of a task means that the task must be triggered if the condition $\texttt{ct}\ \mathcal{PT}$ is evaluated to true. Here, `ct` denotes system time.
- *Spatial constraint* $\Theta_\zeta(t)$: Similarly, a spatial constraint is specified as $\Gamma(\mathcal{L})$. $\mathcal{L}$ is a physical location given by a user. The $\Gamma(\mathcal{L})$ constraint prescribes that the task must be fired when the condition $\texttt{cl} = \mathcal{L}$ is evaluated to true, where `cl` denotes the current location of the user. A location $\mathcal{L}_1$ is considered the same as another location $\mathcal{L}_2$ if the distance between two points of $\mathcal{L}_1$ and $\mathcal{L}_2$ does not exceed a certain value (e.g., less than 10 meters).

Temporal and spatial constraints can be empty, meaning that the corresponding task can be executed anytime and anywhere. It is important to make sure that the constraints specified by users are *conflict free*, meaning that the specified execution context should not be inconsistent with the specification of process schemas. For example, in our digital class assistant (Fig. 2), QB should be executed *before* QV because there is a transition leading from QB to QV. However, a student may correspondingly specify that QB and QV will be executed at `9 a.m.` and `8 a.m.`, which clearly violates the specification. She may specify different spatial constraints for two tasks which are supposed to be executed at the same time. This is also impossible because a user cannot be at two different places at a time. It is fairly easy to do the consistency check of spatial constraints: only ensuring that the same spatial constraints are specified to the tasks which are supposed to be executed in parallel (i.e., tasks in the concurrent regions of an AND state). We use the temporal interval reasoning algorithm proposed in [20] to check the consistency of the temporal constraints. If any inconsistency is detected, the user is requested either to review her configuration, or to relax certain temporal or spatial constraints. The interaction keeps going until the specification of the composite service is declared free of conflicts.

3.1.2 Data Supply and Delivery Contexts

Data supply and delivery contexts deal with user's preferences for the input and output of tasks. The configuration of data supply context consists of two steps: 1) identify which input parameter values can be derived from user profile and 2) supply the location (e.g., URI) of the profile and the corresponding attribute names. The system will automatically generate queries to extract values from the user profile for input parameters at runtime. In our approach, user profile includes information related to a user (e.g., user's name, student id, and courses) and is represented in XML document. Once a profile for a user is defined, it can be reused across different process schemas.

1. A more detailed discussion of context can be found in [19].

TABLE 2
Selected Exception Events

Exception	Description	Level
disconnected(device *d*)	the device *d* has disconnected.	user
unpresentable(result *r*, device *d*)	the service result *r* is evaluated to be unpresentable in the user's device *d*.	user
failed(service *s*, task *t*)	the service *s* is unable to execute task *t*.	component
delay(service *s*, task *t*)	the invocation of service *s* associated with task *t* takes longer than the estimated time.	component
QoSDegraded(service *s*, QoS *q*)	the QoS *q* of service *s* has deteriorated, e.g., its execution time becomes longer.	community
serviceRegistered(service *s*, community *c*)	service *s* is registered with community *c* and becomes a member of *c*.	community
serviceDeregistered(service *s*, community *c*)	service *s* is removed from community *c*.	community

Similarly, for the output parameters of a task, a user may specify which parameter values need to be delivered to her.

It is worth mentioning that our design is not limited to the above two contexts. For example, we can envision a task that must be executed within certain security and quality contexts.

3.2 Multilevel Exception Handling Policies

Due to the dynamic nature of Web services and error-prone service provisioning environments, various *service provisioning exceptions* can occur during the enactment of a composite service. By exceptions, we mean abnormal events caused by service failures, network errors, and resource or requirements changes. The lack of exception handling causes problems like poor performance, wasted resources, nonoptimal service provision, and even failures of process enactment. Services therefore should be *proactive*: they should be able to adapt themselves in response to run-time exceptions.

Policies are rules that control choices in a system's behavior. In our work, we adopt a *policy-based approach* for exception handling that expresses and controls exception handling strategies at a high level of abstraction, decoupling it from the composite service's functionality. Service designers, even end users, can dynamically add, remove, and modify policies, to reconfigure the composite services without changing their functionalities (e.g., control flow embedded in statecharts).

3.2.1 Exception Types

We distinguish three classes of service provisioning exceptions: *user*, *component service*, and *community*.

User exceptions. Many exceptions can occur at the user level. For instance, a mobile device that a user uses to interact with services can be disconnected unexpectedly due to discharged battery, alignment of antennas, or lack of coverage area. Further, a service result might not be able to be displayed on the mobile device because of the lack of appropriate facilities. Some exceptions are related to the changes of the configured composite services launched by users. For example, a user may wish to reset her preferences on a specific task (e.g., spatial constraint of `QB`, `QV`, and `QP` in the class assistant) due to situation changes (e.g., lecture room rescheduled).

Component service exceptions. During the execution of a composite service, different exceptions at the component service level can occur. In particular, the selected service that executes a task of the composite service may become unavailable because it is overloaded or its respective server is down; the execution of a service might take longer than the estimated time and even might fail.

Community exceptions. Community level exceptions are due to QoS changes of community members or membership changes of a community. For instance, a member service might increase its execution price; the execution duration of the service might become longer due to a sudden increase in the number of requests. In addition, some new Web services with better QoS might join the community, while some registered services might leave (deregister) the community. If such changes happen after the selection, the selected services may no longer be considered as *optimal*. Redoing the selection would be necessary to make sure that optimal Web services are always selected for the execution of tasks.

An *exception event* is generated in response to the occurrence of a service provisioning exception. For example, if the invocation of a service s, which was selected to execute a task t of a composite service, is failed, the exception event `failed(`s`,`t`)` will be generated. Table 2 lists part of exception events supported in our work.

3.2.2 Exception Handling Policies

An *exception handling policy* prescribes the knowledge on the appropriate response to a particular exception event, providing a means for flexible, robust, and adaptive service invocation. Exception handling policies can be specified by a service designer at process schema definition time, and modified by an end user during schema configuration phase.

We use the Ponder language [21], especially its *obligation policies*, for the specification of exception handling policies. Obligation policies are declarative *event-action-condition* rules defining the actions that policy subjects (entities having the authority to initiate a management decision) must perform on target entities, when specific events occur and specific conditions hold upon event occurrence. Several reasons motivate our choice of Ponder obligation language for specifying exception handling policies:

- Ponder obligation policies are event-triggered rules. Such event-driven model is an ideal candidate for composite services because their component services are typically distributed.
- Ponder obligation policies are declarative rules and service designers and users can express their exception handling strategies directly without having to change composite service functionalities.
- Ponder obligation policies are amenable to policy analysis and verification. Unlike exception handling policies embedded in implementation code (e.g., `catch` clauses in Java code), it is possible to validate

Policy	Syntax	Example
On	event-specification;	arrived(r,Fiona-Proxy)
Subject	subject-specification;	classAssistant
Target	target-specification;	Fiona-Proxy
Do	action-list;	silentDeliver(r)
[**When**	constraint-expression;]	Fiona.status="at meeting"
[**EffectiveInterval**	time-expression;]	Υ'(between,[9am, 5pm])

Fig. 3. Syntax of extended Ponder obligation policies and an example.

declarative exception handling policies externally for service exception handling.

Fig. 3 shows the syntax of Ponder obligation policies. The `On` clause identifies the triggering event(s), while the `Subject` clause identifies the entities having the authority to initiate a management decision. The `Do` clause identifies the exception handling action to perform on the target entity—identified by `Target` clause—when the event occurs. Combinations of various actions can be generated using sequence and concurrency operators provided by Ponder language. `When` clause is optional, defining the conditions that must hold for the policy to be applied.

We introduce a new element called `EffectiveInterval` to indicate the time period in which a policy is enabled. `EffectiveInterval` is a temporal condition, expressed as $\Upsilon'(\mathcal{P}, \mathcal{T})$ where $\mathcal{P}$ is a comparison operator and $\mathcal{T}$ is a time (similar to temporal constraint in Section 3.1.1). If `EffectiveInterval` is not specified, we assume the policy is always enabled. By means of `EffectiveInterval`, which is not provided in the Ponder language, it is possible to state that certain policies are not always enabled, rather they are enabled only during specific temporal intervals.

Fig. 3 also provides an example policy, which is specified by a user indicating that during the working hours (i.e., from 9 a.m. to 5 p.m.), whenever a service result is arrived at her proxy service, if she is at a meeting, the result should be delivered to her PDA without sound alert.

4 Event-Driven Execution Model

Existing service provisioning systems such as [22] are centralized and service orchestration is ensured by a single process which acts as a central scheduler. These centralized execution models assume that the connection between the central scheduler and the component services is continuously available, with the same characteristics (e.g., latency, bandwidth, reliability). Such assumptions are not valid in the case of composite services for users in dynamic environments, where executions are initiated and followed up by a possibly mobile client. In addition, centralized execution models suffer of the availability and scalability problems [23]. Accordingly, to achieve adaptive and scalable execution of composite services in dynamic environments, the participating services should be *self-managed*: they should be capable of coordinating their actions in an autonomous way, without having to continuously synchronize with a central entity, which could lead to idle periods and time-outs due to disconnections.

4.1 Orchestration Enabling Services

The proposed execution model is based on Linda Tuple Space [24], which is recognized as an attractive model for managing interactions among loosely coupled entities in distributed and dynamic environments. The model consists of three core services, namely the *coordination service*, the *context service*, and the *event service*. These three elements form what we call *orchestration enabling services*. When executing a composite service, orchestration enabling services automatically schedule and execute the component services, and adapt to user configured exceptions and contexts.

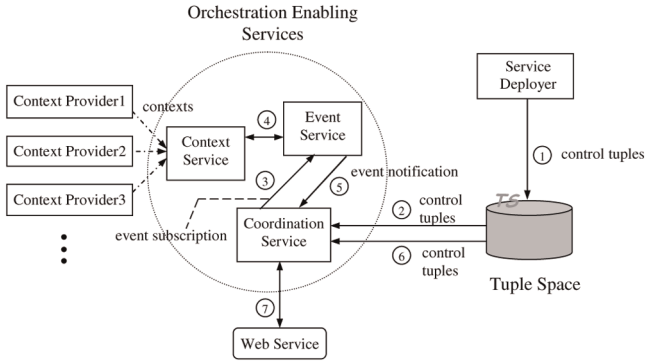

Fig. 4. Interactions of orchestration enabling services.

Each participating Web service is associated with a coordination service that monitors and controls the service execution. In essence, the coordination service is a *lightweight scheduler* that determines when should a component service be executed, and what should be done after the execution is completed. The knowledge needed by a coordination service in order to answer these questions at runtime is statically extracted from the description of the composite service (e.g., statecharts, user contexts, exception handling policies), represented in the form of *control tuples*, as detailed in Section 4.2, and placed in the corresponding tuple space. The generation of control tuples will be reported in Section 4.3.

A coordination service enforces the control tuples with the help of an event service and a context service. The event service is responsible for disseminating events registered by the coordination service, and the context service is responsible for collecting context information from *context providers*. Context providers can be components inside the system (e.g., coordination services providing execution status of Web services), or third party entities outside the system (e.g., `GlobalWeather` Web service[2]).

Fig. 4 elaborates the interactions between these three generic services. When the control tuples, which are generated from a composite service, are injected into the tuple space of a Web service (step 1), the coordination service associated with this Web service parses the control tuples (step 2), retrieves relevant information (e.g., events, conditions, and actions), and registers the events (e.g., `failed`) with the event service (step 3). The event service then subscribes relevant contexts needed by the events (e.g., `executionStatus` for event `failed`) to the context service (step 4), which collects the context values from relevant context providers. The event service fires and distributes events if the corresponding conditions are matched

2. http://www.capescience.com/webservices/globalweather/index.shtml, which provides forecasted weather information.

TABLE 3
Selected Events and Actions Supported in CCAP

Events	Descriptions
entered(location l)	the user has entered the location l.
started(timeInterval i)	the current time is equal to the lower bound of interval i.
ready(parameters p)	the values of the input parameters p are available.
completed(service s)	the completion notification of the execution is received by the controller attached to service s.
presentable(result r, device d)	the service result r is evaluated as presentable in the user's device d.
arrived(result r, userproxy u)	the service result r is received by the user's proxy service u.
Actions	**Descriptions**
notify(task t)	send a notification of completion to task t.
sendResult(output $\mathcal{O}$, receiver re)	send the output $\mathcal{O}$ to the receiver re, which could be a user, the user's agent or other tasks.
transform(serviceResult r, tranformService s, device d)	transform service result r using the transformation service s.
execute(task t)	invoke the task t.
retry(service s)	allows to invoke the service s another time after a failure.
informNewLocation(location $new\mathcal{L}$, serviceSet $\mathcal{SC}$)	inform the location $new\mathcal{L}$ of a user to the relevant Web services $\mathcal{SC}$.
collectData(inputParameter in, supplyingSource ss)	collect the value of the input parameter in using the supplying source ss.
reassign(task t, service s)	delegate the invocation of task t to a service s.

(e.g., `executionStatus`="failed") (step 5). Upon receiving the notifications (i.e., event occurrence) from the event service, the coordination service extracts the corresponding control tuples from the tuple space (step 6), evaluates the conditions, and performs the proper actions (step 7).

4.2 Control Tuples

We first introduce the concept of control tuples, and then introduce the specific control tuples supported by CCAP for the coordination of adaptive composite service executions, namely *service invocation tuples* and *exception handling tuples*.

Definition 1 (Control tuple). *A control tuple is a rule of the form $\mathcal{E}[\mathcal{C}]/\mathcal{A}$ where*

- *$\mathcal{E}$ is a conjunction of execution events. Execution events are generated in response to changes of the status of service execution environments, including execution exceptions. Table 3 gives some events supported in our system.*[3] *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 with any order, then an occurrence of $e_1 \wedge e_2$ is generated,*
- *$\mathcal{C}$ is a Boolean expression that combines conditions on execution states including event parameter values and service information (e.g., inputs and outputs of tasks), and*
- *$\mathcal{A}$ is a sequence of execution actions $a_1; a_2; \ldots; a_n$, which are executed in the order specified by the sequence. Some selected actions supported in CCAP are given in Table 3 (the signatures are omitted for clarity reasons).*

Briefly, the basic semantics of a control tuple is as follows: when the event(s) of the tuple $\mathcal{E}$ is triggered and if its condition(s) $\mathcal{C}$ evaluates to true, the corresponding action(s) $\mathcal{A}$ will be performed.

4.2.1 Service Invocation Tuples

There are two kinds of service invocation tuples: 1) *preinvocation tuples* contain the knowledge to answer, *before* the execution of this task, questions such as what are the actions that need to be performed and what are the conditions that need to be satisfied and 2) *postinvocation tuples* contain the knowledge to answer, *after* an execution of this task, questions such as which entities (e.g., other tasks) need to be notified of this completion, and which output needs to be sent to which entity. In the following, we formally define the concepts of preinvocation and postinvocation tuples of a task.

We first introduce the concept of *compound transition*, which will be used in the definitions of service invocation tuples. As discussed in Section 2, a statechart can have compound states (AND and OR states) which serve to decompose the statechart into substates and specify regions of the statechart which can be executed concurrently. Due to this feature, there can be multiple direct and indirect ways of transitioning from a given basic state to another basic state. 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 `QB` and `QP` in Fig. 2) and some are compound (e.g., the transition between `QP` and `CB` in Fig. 2). Hence, it is important to determine how to route control-flow notifications and data items between basic states. Intuitively, a compound transition is a path (i.e., a list of linked transitions) going from a basic state to another basic state without passing through any other basic state.

Definition 2 (Compound transition). *A compound transition $\mathcal{CT}$ is a sequence of transitions $tr_1, tr_2, \ldots, tr_n$ belonging to a given statechart, such that:*

- *$source(tr_1)$*[4] *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 $\mathrm{target}(tr_i)$.*

Under these conditions, $\mathcal{CT}$ is said to connect $source(tr_1)$ with $target(tr_n)$, i.e., $source(\mathcal{CT}) = source(tr_1)$ and $target(\mathcal{CT}) = target(tr_n)$. The condition part of $\mathcal{CT}$, noted $Cond(\mathcal{CT})$, is the conjunction of the conditions labeling $tr_1, tr_2, \ldots, tr_n$, expressed as $Cond(\mathcal{CT}) = \{c_1 \wedge c_2 \wedge \cdots \wedge c_n\}$, where c_i is the condition labeling transition tr_i.

3. It should be noted that the exception events (see Table 2) are not included in this table.

4. Here, $\mathrm{source}(tr)$ denotes the source state of transition tr, while $\mathrm{target}(tr)$ denotes the target state of tr.

It should be noted that although this definition of compound transition is specific to statecharts, a similar concept can be defined for virtually any other language for process-based service composition (e.g., BPEL). The techniques that we present can thus be applied to other languages, as long as a definition of compound transition for the language is provided.

Definition 3 (Preinvocation tuple). *The preinvocation tuples of task t of a composite service $\mathcal{CS}$ include two kinds of control tuples: data collection and precondition tuples. The former is a control tuple such that:*

- *$\mathcal{E}$ is empty.*
- *$\mathcal{C}$ is a conjunction of temporal and spatial constraints of t, i.e., $\Theta_\tau(t)$ and $\Theta_\zeta(t)$. If t does not have any constraint, $\mathcal{C}$ is interpreted as* `true`.
- *$\mathcal{A}$ are actions in the form of* `collectData`(ip, ss), *where ip is an input parameter whose value should be obtained from a supplying source ss.*

While the precondition is a set of control tuples such that:

- *$\mathcal{E}$ is a conjunction of events* `ready`$(\Theta_\iota(t))$ *and a set of* `completed`(t'), *where t' is a task that there exists a compound transition $\mathcal{CT}$ such that* $\text{source}(\mathcal{CT}) = t'$ *and* $\text{target}(\mathcal{CT}) = t$. *The event* `completed`$(t')$ *is raised when a notification of completion is received from task t'.*
- *$\mathcal{C}$ is a conjunction of temporal and spatial constraints of t, i.e., $\Theta_\tau(t)$ and $\Theta_\zeta(t)$. If t does not have any constraint, $\mathcal{C}$ is interpreted as* `true`.
- *$\mathcal{A}$ is an action in the form of* `execute`(t), *which invokes the task t.*

Definition 4 (Postinvocation tuple). *The postinvocation tuples of t of $\mathcal{CS}$ contain a set of control tuples such that:*

- *$\mathcal{E}$ is an event* `completed`(t). *The event is generated when the execution of task t is completed.*
- *There exists a compound transition $\mathcal{CT}$ such that* $\text{source}(\mathcal{CT}) = t$ *and* $\text{target}(\mathcal{CT}) = t'$.
- *$\mathcal{C}$ is* $\text{Conjunction}(\text{Cond}(\mathcal{CT}))$, *where* $\text{Conjunction}(c_1 \wedge c_2 \wedge \cdots \wedge c_n) = c_1, c_1, \ldots, c_n$.
- *$\mathcal{A}$ are actions* `notify`(t') *and a set of* `sendResult`$(\mathcal{O}, r)$. *The $\mathcal{O}$ is a set of output parameters whose values need to be delivered to a receiver r, which could be the user or another task of $\mathcal{CS}$. The actions are executed in the order specified as $action_1; action_2; \ldots; action_n$.*

4.2.2 Exception Handling Tuples

Definition 5 (Exception handling tuple). *An exception handling tuple acts as an instruction to execute a particular action if a specific exception event occurs and a specific condition(s) holds. It is a control tuple such that:*

- *$\mathcal{E}$ is an exception event. The example of exception events can be found in Table 2.*
- *$\mathcal{C}$ is a conjunction of conditions on execution states including event parameter values and service information (e.g., input and output of tasks).*
- *$\mathcal{A}$ is an exception handling action. Examples of exception handling actions are: 1)* `retry`(s) *allows reinvoking a service s after a failure, 2)* `forward`(s_1, s_2) *allows a service s_1 to forward an invocation message to another service s_2, and c)* `transform`(r, s, d) *allows transforming service result r using the transformation service s according to the capabilities of the user's device d.*

Algorithm: PreInv
Input: task t
Output: the set of pre-invocation tuples $\mathcal{PRE}$

1: Let $\mathcal{TR}_\mathcal{I}$=$\{tr_1, tr_2, \ldots, tr_n\}$ be incoming transitions of t.
2: Let $\mathcal{ID}$=$\{id_1, id_2, \ldots, id_m\}$ be input data dependencies of t. $id_i = (ip_i, ss_i)$ meaning that the value of input parameter ip_i should be obtained from the supplying source ss_i.
3: $\mathcal{PRE} \leftarrow \phi$
4: Let $\mathcal{CD} \leftarrow \phi$ be the set including actions that collect values for input parameters.
5: **for all** $id_i \in \mathcal{ID}$ **do**
6: $\quad \mathcal{CD} \leftarrow \mathcal{CD} \cup$ collectData(ip_i, ss_i)
7: **end for**
8: $\mathcal{PRE} \leftarrow$ [Θ_τ and Θ_ζ]/$\mathcal{CD}$ /*data collection tuples*/
9: **for all** $tr_i \in \mathcal{TR}_\mathcal{I}$ **do**
10: $\quad \mathcal{PRE} \leftarrow \mathcal{PRE} \cup$ PreProcT(tr_i)
11: **end for**
12: **return** $\mathcal{PRE}$

13: PreProcT(tr)=
14: $\quad$ **if** source(tr) is a basic state **then**
$\quad\quad$ {ready(Θ_ι)$\wedge$completed(source(tr))[Θ_τ and Θ_ζ]/execute(t)}
15: $\quad$ **else if** source(tr) is an initial state **then**
16: $\quad\quad$ let $\mathcal{SUP}$ = superstate(source(tr))
17: $\quad\quad\quad$ **if** $\mathcal{SUP}$ is the topmost state of the statechart, **then**
18: $\quad\quad\quad\quad$ {ready(Θ_ι)[Θ_τ and Θ_ζ]/execute(t)}
19: $\quad\quad\quad$ **else** let $\{st_1, st_2, \ldots, st_n\}$ be the incoming transitions of $\mathcal{SUP}$
20: $\quad\quad\quad\quad$ PreProcT(st_1)$\cup$PreProcT(st_2)$\ldots\cup$PreProcT(st_n)
21: $\quad$ **else if** source(tr) is an OR state **then**
22: $\quad\quad$ let $\{ft_1, ft_2, \ldots, ft_n\}$ be the final transitions of source(tr)
23: $\quad\quad\quad$ PreProcT(ft_1)$\cup$PreProcT(ft_2)$\cup \ldots \cup$ PreProcT(ft_n)
24: $\quad$ **else** /* source(tr) is an AND state */
25: $\quad\quad$ let $\{\mathcal{CR}_1, \mathcal{CR}_2, \ldots, \mathcal{CR}_n\}$ be the concurrent regions of source(tr),
26: $\quad\quad$ let $\{ft_1\_\mathcal{CR}_1, ft_2\_\mathcal{CR}_1, \ldots, ft_n\_\mathcal{CR}_1\}$ be the final transitions of of $\mathcal{CR}_1$,
27: $\quad\quad$ let $\{ft_1\_\mathcal{CR}_2, ft_2\_\mathcal{CR}_2, \ldots, ft_n\_\mathcal{CR}_2\}$ be the final transitions of of $\mathcal{CR}_2$,
28: $\quad\quad \ldots$
29: $\quad\quad$ let $\{ft_1\_\mathcal{CR}_n, ft_2\_\mathcal{CR}_n, \ldots, ft_n\_\mathcal{CR}_n\}$ be the final transitions of of $\mathcal{CR}_n$
30: $\quad\quad\quad$ (PreProcT($ft_1\_\mathcal{CR}_1$)$\cup \ldots \cup$ PreProcT($ft_n\_\mathcal{CR}_1$)) $\times$
31: $\quad\quad\quad$ (PreProcT($ft_1\_\mathcal{CR}_2$)$\cup \ldots \cup$ PreProcT($ft_n\_\mathcal{CR}_2$)) $\times$
32: $\quad\quad\quad \ldots$
33: $\quad\quad\quad$ (PreProcT($ft_1\_\mathcal{CR}_n$)$\cup \ldots \cup$ PreProcT($ft_n\_\mathcal{CR}_n$))

Fig. 5. Algorithm for generation of preinvocation tuples.

4.3 Control Tuples Generation

In this section, we propose in turn the algorithms for generating the service invocation and exception handling tuples from adaptive composite services.

4.3.1 Preinvocation Tuples Generation

The generation of preinvocation tuples of a task relies on the contextual attributes of the task (e.g., temporal and spatial constraints), data dependencies of input parameters, and control flows associated with the task. The task's *incoming* transitions are analyzed and preinvocation tuples are generated for each incoming transition of the task.

The algorithm for the generation of preinvocation tuples of a task is given in Fig. 5. It takes as input a task t and produces a set of preinvocation tuples for t. The algorithm analyzes the data dependencies of the input parameters ($\mathcal{ID}$, line 2) and the incoming transitions of t ($\mathcal{TR}_\mathcal{I}$, line 1). From $\mathcal{ID}$, a set of actions ($\mathcal{CD}$) is created prescribing which

supplying source should be used in order to get the value of which input parameter (lines 4-7). The data collection tuple of t is then created by putting temporal and spatial constraints (Θ_τ and Θ_ζ) as condition and $\mathcal{CD}$ as action (line 8). The preinvocation tuples of t is the union of the data collection tuple and the precondition tuples associated with the incoming transitions of t (lines 9-11).

The function named `PreProcT` computes the preconditions of a transition, which takes as input a transition tr, and returns a set of precondition tuples associated with this transition. `PreProcT` distinguishes the cases where the source of the transition is a basic state, an initial state, and compound state (i.e., `AND` or `OR` state). In the first case, the precondition `ready(`Θ_ι`)`$\wedge$`completed(source(tr))[`Θ_τ `and` Θ_ζ`]/execute(t)` is created, meaning that when all the values of the input parameters of t are available, and the execution of task `source(`tr`)` is finished, if the temporal and spatial constraints are satisfied, task t will be executed (line 14). In the second case (the transition tr stems from an initial state), the incoming transitions of its superstate are considered and one or several precondition tuples are generated for each of them (lines 15-20). Notice that if the superstate of tr is the topmost state of the statechart, tr is an *initial transition* of the composite service and the precondition tuple is therefore {`ready(`Θ_ι`)[`Θ_τ `and` Θ_ζ`]/execute(`t`)`}. In the third case (the transition tr stems from a compound state), the function `PreProcT` is applied recursively to the final transitions of the compound state, and the results are merged (lines 21-33). In the case of an OR state, the merging is a simple set union (line 23). In the case of an AND state, each concurrent region is treated as an OR state, and the precondition tuples obtained for each concurrent region are merged through a *Cartesian product* (lines 25-33), meaning that the AND state is exited (i.e., task t can be executed) if one of the final transitions in each of the concurrent regions is taken.

Algorithm: PostInv
Input: task t
Output: the set of post-invocation tuples $\mathcal{POST}$

1: Let $\mathcal{TR}_\mathcal{O}$=$\{tr_1,tr_2,\ldots,tr_n\}$ be outgoing transitions of t.
2: Let $\mathcal{OD}$=$\{od_1,od_2,\ldots,od_m\}$ be output data dependencies of t. $od_i = (\mathcal{O}_i, r_i)$ meaning that output parameters $\mathcal{O}_i$ should be delivered to r_i.
3: $\mathcal{POST} \leftarrow \phi$
4: Let $\mathcal{RD} \leftarrow \phi$ be the set including actions that deliver outputs to the receivers.
5: **for all** $od_i \in \mathcal{OD}$ **do**
6: $\quad \mathcal{RD} \leftarrow \mathcal{RD} \cup$ sendResult($\mathcal{O}_i$, r_i)
7: **end for**
8: **for all** $tr_i \in \mathcal{TR}_\mathcal{O}$ **do**
9: $\quad \mathcal{POST} \leftarrow \mathcal{POST} \cup$ AddAction($\mathcal{RD}$, PostProcT(tr_i))
10: **end for**
11: **return** $\mathcal{POST}$

12: PostProcT(tr)=
13: $\quad$ **if** target(tr) is a basic state **then** {completed(source(tr))[cond(tr)]/notify(target(tr))}
14: $\quad$ **else if** target(tr) is a compound state **then**
15: $\quad\quad$ let $\{it_1, it_2, \ldots, it_k\}$ be the initial transitions of target(tr)
16: $\quad\quad$ AddCond(cond(tr), PostProcT(it_1)$\cup$PostProcT(it_2)$\cup\ldots\cup$PostProcT(it_k))
17: $\quad$ **else if** target(tr) is a final state **then**
18: $\quad\quad$ let $\mathcal{SUP}$=superstate(target(tr))
19: $\quad\quad\quad$ **if** $\mathcal{SUP}$ is the topmost state of the statecharts **then**
20: $\quad\quad\quad\quad$ {completed(source(tr))[cond(tr)]/notify($\mathcal{SUP}$)}
21: $\quad\quad\quad$ **else if** $\mathcal{SUP}$ is an OR state **then**
$\quad\quad\quad\quad$ AddCond(cond(tr), PostInv($\mathcal{SUP}$))
22: $\quad\quad\quad$ **else**
$\quad\quad\quad\quad \bigcup_{s\in \mathrm{CTargets}(\mathcal{SUP})}$ {completed($source(tr)$)[cond(tr_i)]/notify(s)}
23: AddCond(c, $\{e_1[c_1]a_1, e_2[c_2]a_2, \ldots, e_n[c_n]a_n\}$) = $\{e_1[c$ and $c_1]a_1, e_2[c$ and $c_2]a_2, \ldots, e_n[c$ and $c_n]a_n\}$
24: AddAction($\mathcal{A}$, $\{e_1[c_1]a_1, e_2[c_2]a_2, \ldots, e_n[c_n]a_n\}$) = $\{e_1[c_1]a_1, e_2[c_2]a_2, \ldots, e_n[c_n]a_n\} \cup \bigcup_{a\in\mathcal{A}} \{e_1[c_1]a, e_2[c_2]a, \ldots, e_n[c_n]a\}$
25: CTargets(t)=$\{$target($\mathcal{CT}$)$|$ $\mathcal{CT}$ is a compound transition $\wedge$ source($\mathcal{CT}$)=$t\}$

Fig. 6. Algorithm for generation of postinvocation tuples.

4.3.2 Postinvocation Tuples Generation

Similarly, Fig. 6 describes the algorithm `PostInv` for generating postinvocation tuples for a task. The algorithm takes as input a task t, and produces a set of postinvocation tuples. The algorithm analyzes the data dependencies of the output parameters ($\mathcal{OD}$, line 2) and the outgoing transitions of t ($\mathcal{TR}_\mathcal{O}$, line 1). From $\mathcal{OD}$, a set of actions (i.e., $\mathcal{RD}$) is created indicating which outputs should be delivered to which receivers (lines 4-7). The postinvocation set of t is the union of the postinvocation tuples associated with the outgoing transitions of t (lines 8-10).

The postinvocation 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 postinvocation tuples including the postprocessing actions associated with this transition. There exist various cases. 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 the task controller of t' (line 13). If an outgoing transition points to a compound state, one postinvocation tuple is generated for each of the initial transitions of this compound state (lines 14-16). Finally, if the outgoing transition points to a final state of a compound state, the outgoing transitions of this compound state are considered in turn, and one or several postinvocation tuples are produced for each of them (lines 17-22). Three auxiliary functions `AddCond`, `AddAction`, and `CTargets` are described in lines 23-25 using a functional programming style. Note that `PostProcT` does not handle the output data dependencies of the task (i.e., $\mathcal{RD}$), which need to be added to the tuples generated by `PostProcT` (line 9).

4.3.3 Exception Handling Tuples Generation

Exception handling tuples are generated from predefined exception handling policies (Section 3.2). Since exception handling policies are expressed as extended Ponder obligation polices that are also event-condition-action rules, the generation of exception handling tuples from exception handling policies is straightforward.

The generated tuples are injected into the tuple spaces of relevant coordination services. The information on where an exception handling tuple should be uploaded is specified in the *target* entity of the corresponding exception handling policy. As mentioned in Section 3.2, there are three different kinds of targets: *component service*, *service community*, and *user*. The generated exception handling tuples should be injected into:

- the tuple space of the coordination service of the component service if the target of a policy is a *component service*,

- the tuple space of the coordination service of the community if the target of a policy is a *service community*, or
- the tuple space of the user's proxy service if the target of a policy is a *user*.

4.4 Discussion

The ECA model was originally developed in active database systems [25] and has been widely used in workflow and business processes management [26], [27]. Although the ECA model has a sound theoretical basis, it is not easy to visualize the meaning of rules, and thus very difficult for users to understand and manage them. In addition, the previous approaches require a considerable amount of manual efforts in generating rules, causing many difficulties in dealing with complex processes.

Our approach, in fact, combines the techniques of graphical process representation and ECA rules. The statechart-based service composition model provides convenience for a human user to grasp the actual processes, while ECA rules (i.e., control tuples in CCAP) are used to transform the graphical model of composite services into a machine-readable form so that the execution of the composite services can be performed automatically. In particular, we developed an algorithm to systematically analyze composite service specifications, which, in turn, automatically generates service orchestration rules, based on a set of abstracted events and actions.

The most significant benefit of realizing service orchestration by means of control tuples (in the form of ECA rules) is what we called *knowledge independence*, in the sense that control tuples are stored separately from a composite Web service specification. This provides the possibility of distributing control tuples to participating Web services, thereby realizing a fully distributed execution of composite services. In addition, it is possible to add or remove control tuples to overlay behavior on top of *deployed* composite services in response to unforeseen situations or particular requirements.

5 Implementation and Experiments

This section is devoted to the implementation and performance study of our proposed service composition approach.

5.1 Implementation

CCAP is designed as a layered architecture shown in Fig. 7. The service builder, the service discovery engine, the proxy service, and the service deployer compose the *service development and deployment environment*, which provide a service composition environment where service designers and users can compose and invoke Web services. In particular, service designers can specify process schemas while end users can select a specific process schema, configure, and then execute it. The *runtime environment* consists of a set of generic services (coordination, context, and event) that provide mechanisms for enacting the execution of composite Web services.

CCAP has been implemented in Java and is based on state-of-the-art technologies like XML, SOAP, WSDL, and UDDI [28]. Java2WSDL, a tool provided by *Apache Axis*,[5] is

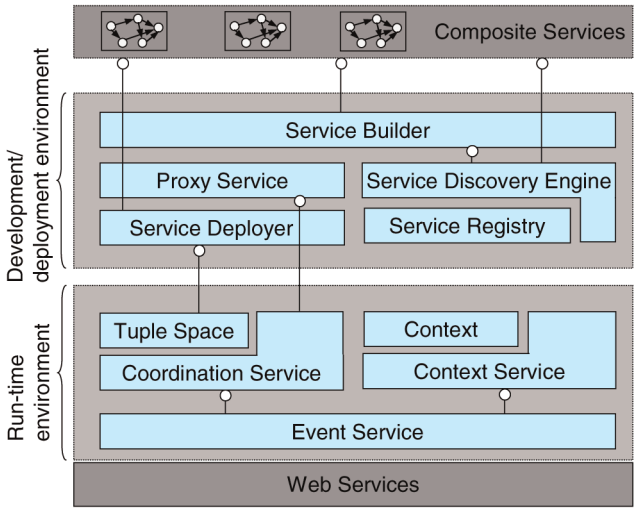

Fig. 7. Multilayer architecture of CCAP.

used to generate WSDL descriptions from the Java class files so that all the components of CCAP are invoked as Web services. Services are deployed using Apache Axis. In our implementation, we use *Apache Tomcat*[6] as a Web server where Apache Axis is deployed. Apache Axis provides not only a server-side infrastructure for deploying and managing services, but a client-side API for invoking these services. Each service has a *deployment descriptor* that includes the unique identifier of the Java class to be invoked, session scope of the class, and operations in the class available for the clients. Each service is deployed using the service management client by providing its descriptor and the URL of the Axis servlet `rpcrouter`.

5.1.1 Service Development/Deployment Environment

The service discovery engine facilitates the advertisement and location of services. Service registration, discovery, and invocation are implemented by SOAP calls. When a service registers with a discovery engine, a UDDI SOAP request containing the service description in WSDL is sent to the UDDI registry. After a service is registered in the UDDI registry, service designers and end users can locate the service by sending the UDDI SOAP request (e.g., business name, service type) to the UDDI registry.

The discovery engine is implemented using the IBM Web Services Toolkit 2.4 (WSTK) [29]. WSTK provides several components and tools for Web service development. In particular, we use the UDDI Java API (UDDI4J) to access a private UDDI registry (i.e., hosted by the CCAP platform), as well as the WSDL generation tool for creating the WSDL documents and SOAP service descriptors required by the discovery engine.

The service builder assists service designers in the creation and maintenance of composite services. It provides an editor for describing the statechart diagram of a composite service operation and for importing operations from existing services into composite services and communities. It should be noted that the service builder also supports the specification of process schemas. We designed an XML schema for process

5. http://ws.apache.org/axis/index.html.

6. http://jakarta.apache.org/tomcat/.

description. Each process is represented in an XML document and has a business entry in the UDDI registry, with a tModel[7] of type `processSpec`. Service developers can design their new composite services (e.g., via specifying particular preferences) on top of these schemas.

The service deployer is responsible for generating control tuples of every task of a composite service statechart, using the algorithms presented in Section 4.3. The input of the programs implementing these algorithms are statecharts represented as XML documents (which are generated by the service builder), while the outputs are control tuples formatted in XML as well. Once the control tuples are generated, the service deployer assists the service designer in the process of uploading these tuples into the tuple spaces of the corresponding component services and the composite service. Tuple spaces are implemented using IBM TSpaces [30], which is a network communication buffer with database capabilities. Coordination services communicate asynchronously through the shared spaces by writing, reading, and taking control tuples. Each tuple is implemented as a vector of Java objects.

5.1.2 Runtime Environment

This layer contains the three core generic services that provides the execution semantics for the adaptive composite services. The *coordination* service provides an operation called `coordinate` for receiving messages, managing service instances (i.e., creating and deleting instances), registering events to the event service, triggering actions, tracing service invocations, and communicating with other coordination services. The coordination service relies on the tuple space of the associated service to manage service activities.

The *context* service detects, collects, and disseminates context information while the event service fires and distributes events. The context service is built on top of the Context Toolkit [31], a package that supports the development of context-aware applications. In particular, we implemented context providers as a set of context widgets that encapsulate context information and provide methods to access them. Each context widget has a set of attributes that can be queried by the context service. The communications between context widgets and applications are implemented based on `BaseObject` class, provided by the toolkit. Finally, the *event* service provides operations for receiving messages, including subscribing messages from the coordination service of a service and context information from the context service, and notifying the fired events to the coordination services. Readers are referred to [8] for a detailed description of the system implementation.

5.2 Usability and Performance Evaluation

The aim of our performance study is twofold. First, we investigate the potential usage of the proposed service composition platform via a usability study. Second, we compare the performance of our proposed service composition techniques from various aspects including scalability, execution cost, and adaptation effectiveness.

7. In UDDI, a tModel provides a semantic classification of the functionality of a service, together with a formal description of its interfaces.

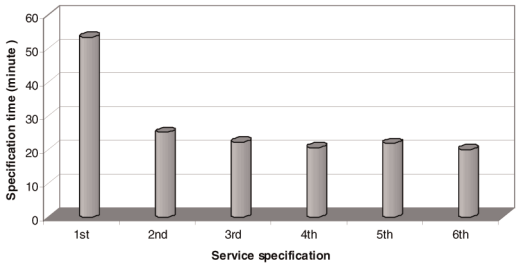

Fig. 8. The time used for specifying the services.

5.2.1 Usability Evaluation

We conducted a usability study to evaluate users' willingness to use the system, and their perception of the system's utility and ease of use.

We presented our system to 41 people, all from different educational backgrounds (21 undergraduate students, 12 masters students, and 8 PhD students) and computer literate. The presentation included a PowerPoint show of the system overview, a demonstration of the CCAP system, and the digital class assistant service that was built on top of CCAP. The participants were then asked to use the system and to report their experience by answering a questionnaire.

The feedback from the participants was quite encouraging. Thirty four people reported that they understood the design principles of CCAP very well after their usage of the system, while seven indicated a partial comprehension. However, lacking the technical knowledge about the system does not seem to prevent students from using it. Most people (37) expressed their willingness to use the system in designing and accessing composite Web services.

We evaluated the learnability and efficiency of CCAP system by asking the participants to create the digital class assistant service and five other composite services using our service composition tools. These five services have the same complexity as the digital class assistant service. The participants were asked to create these services, including the specification of process templates on a desktop and the configuration of the templates using a Pocket PC, and record the time used. We found that users spent more time on their first attempt. The average time they spent is 53 minutes. However, compared to the first attempt, the average time used for specifying the rest five services is only 18 minutes. We can see that it took about 35 minutes for the participants to learn how to use CCAP system. Fig. 8 shows the time spent by a participant in the specification of the six services.

We also collected the feedback of the participants on our CCAP system design. The results are summarized in Table 4. From the table, we can see that most participants can use the system with little effort. In fact, nine people even enjoyed their experience in using CCAP. The ease-of-use component that most people agreed, with no surprise, is the service editor that provide a visual interface where composite services can be specified by drawing statechart diagrams. However, most participants (29) rated the PDA service configuration tool as the most difficult one to use. From Table 4, we can see that although most people had a *not so bad* experience in using the configuration tool, very

TABLE 4
Evaluation Results of the System Design

Questions	Responses		
	A	B	C
What is your opinion on the system interface design: (A) easy to follow; (B) some confusions but generally can be followed; (C) needs redesign	15	21	5
Describe your experience in using the tools: (A) easy to use; (B) spent some time to get familiar with; (C) spent too much effort and frustrated	9	28	4
Which component of CCAP is the most interesting one and is easy to follow: (A) discovery engine; (B) service editor; (C) distributed execution	8	25	8
Which tool of CCAP is the most difficult one to use: (A) discovery engine; (B) service editor; (C) PDA-based service configuration	5	7	29
For the service configuration, which one you prefer to use: (A) desktop version; (B) PDA version; (C) does not matter*	31	4	5

* One person did not respond to this question.

few people (four) would like to use the PDA version if they are given both versions of configuration tool (i.e., Desktop PC and PDA). Reasons that prevent people from using the PDA based configuration tool include:

- difficulties in operating the interface,
- slow interaction speed, and
- more importantly, endless anxiety about the battery power and network connections of PDAs.

We also found out that the people who gave the low ease-of-use ratings are generally not familiar with handheld computing devices. In any case, user training on how to use such kinds of devices would help alleviate some of this anxiety.

5.2.2 Performance Evaluation

This section presents two experimental results. The first one compares the performance between our distributed services orchestration approach and the centralized orchestration approach. The second experiment studies the adaptation effectiveness of the system.

For the experiments, we used the implemented class assistant service (see Fig. 2). We conducted experiments using a PDA and a cluster of PCs running the prototype system. A Mitac Mio 168 GPS integrated Pocket PC is used as a mobile device that connects the 11 Mbit Lucent 802.11 bit access points installed in the building of the School of Computer Science and Engineering at UNSW. A cluster of PCs was used to run the remaining part of the system. One of them is dedicated to the class assistant composite service while others are servers for component services. All PCs have the same configuration of Pentium III 933 MHz and 512 Mbit RAM. Each PC runs Debian Linux and the Java 2 Standard Edition V1.4.2, and is connected to a LAN through 100 Mbps Ethernet cards.

Distributed versus centralized orchestration. The purpose of this experiment is to study and compare the performance of our distributed orchestration model with that of the centralized one. In the implementation of the centralized approach, a *central scheduler* is responsible for sending and receiving messages to and from the component services. The central scheduler is located on the same machine as the class assistant service, while the component services (e.g., `consultation booking` service) are located on other machines. The physical message exchanges in the centralized model correspond to the messages exchanged between the central scheduler and the component services.

On the other hand, in the implementation of the distributed approach, the orchestration enabling services (i.e., coordination service, context service, and event service) of a task and the component service invoked in this task are located in the same machine. The physical message exchanges in this approach correspond to the messages exchanged between the coordination service of the composite service and the coordination services of its component services, as well as those exchanged between the component coordination services. It should be noted that we do not count the messages exchanged between orchestration enabling services (e.g., messages exchanged between the event service and the coordination service) because they are located in the same machines.

We conducted experiment to investigate the execution performance of distributed and centralized execution models, with different size of exchanged messages. In the experiment, we assume that the size of all exchanged messages remains the same during the service execution. The size of messages ranges over the values from 1,000 to 1,024,000. For each message size, we executed the class assistant service 10 times and computed the average service response time. The results for case 1 (i.e., the question has been asked by other students and all the student's questions have been answered by the lecturer, see Fig. 2) is shown in Fig. 9. Similar results were obtained for the other cases.

From Fig. 9a, we can see that, in both the distributed and the centralized approach, the service response time does not change significantly when the size of messages is small. For example, for the distributed approach, the service response time changes slowly from 11.02 to 15.3 seconds when the message size grows from 1,000 to 8,000. In addition, the response time of the distributed approach is slightly bigger than the response time of the centralized approach. To make it easy to compare, we depict the service response time of both approaches with the message sizes from 1,000 to 32,000, in Fig. 9b. It is shown that, in the distributed approach, the response time is 11.02 seconds when the message size is 1,000 and it is 10.52 seconds in the centralized approach. The main reason of more time required for distributed approach is due to the number of exchanged messages in the distributed approach is more than the one in the centralized approach.

We also note that when the size of messages increases, the service response time of the centralized approach increases more sharply than that of the distributed approach (see Fig. 9a). This is due to the reason that the messages in the centralized approach need to systematically transit through a central scheduler which easily constitutes a bottleneck.

Adaptation effectiveness. The purpose of this experiment is to study the effectiveness of the system's adaptation to dynamic environments. The experiment was done by

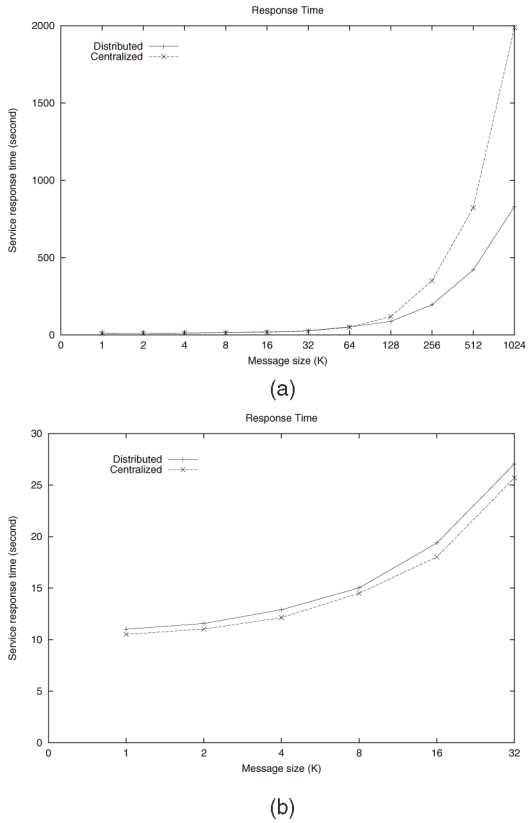

Fig. 9. The response time of the class assistant service. (b) is a closer look at the partial part (message size ranges from 1,000 to 32,000) of (a).

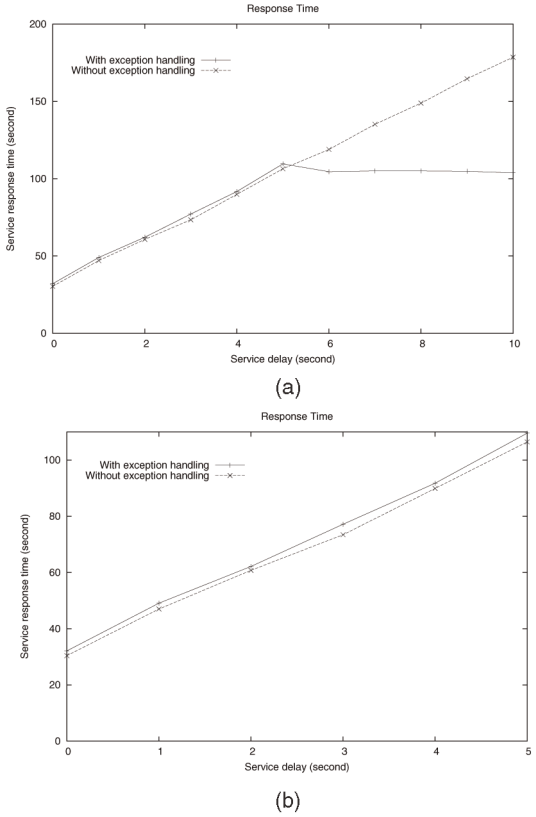

Fig. 10. System performance with and without `Timeout` policy. (b) is a closer look at the partial part (service delay ranges from 0 to 5 seconds) of (a).

comparing the performance of the system with and without the exception handling mechanism.

A `Timeout` policy, "*if the waiting time is longer than five seconds during the invocation of a service, cancel the invocation and execute the substitute service*," was used in the experiment to see how effective our system is when the network becomes overloaded. The exception handling tuples were generated from this policy and injected into the tuple spaces of the services. To simulate the network congestion, we defined a *service delay* that makes the services sleep for a specific amount of time (e.g., 1,000 ms) before their acceptance of the invocation requests. We then executed the two versions of the class assistant service with various network congestion situations and measured the time used. The service delays we simulated range from 1,000 to 10,000 ms. Under each network congestion condition, we executed our service 10 times and computed the average service response time. The result is depicted in Fig. 10.

From Fig. 10a, we can see that exception handling mechanism significantly improved the system performance. When service delays are less than 5 seconds, for exception handling approach and nonexception handling approach, there is no major impact on the service performance because the exception handling policy is not applied yet due to unsatisfied condition of the exception handling tuples. However, the performance difference of both approaches becomes significant when the service delay is increased. For example, when the service delay is 10 seconds, it spends 178.568 seconds to execute the class assistant without exception handling mechanism, while only 103.780 seconds for the one with exception handling. In particular, with the exception handling, the response time of the application tends to level off because the coordination services select new services with less workload for the invocation instead of waiting for the overloaded ones.

It is also interesting to note that when the delay time is less than 5 seconds, the response time of the application with exception handling is slightly higher than the one without exception handling (Fig. 10b). The reason of the higher service response time is due to the overhead introduced by the processing of exception handling tuples. However, we do not consider this to be a significant performance overhead, comparing with the performance achieved by the exception handling tuples.

6 Related Work

Over the last few years, the prosperous research on Web services has led to a multitude of results in composition techniques for Web services. In this section, we overview major techniques that are most closely related to our approach.

6.1 Techniques for Web Service Composition

Current efforts in Web services composition can be generally grouped into three categories: *manual*, *automatic*, and *semiautomatic* composition [4]. By manual composition, we mean that the composite service is designed by a human designer (i.e., service provider) and the whole service

composition takes place during the design time. This approach works fine as long as the service environment, business partners, and component services do not or rarely change. On the other hand, automatic service composition approaches typically exploit the Semantic Web and artificial intelligence (AI) planning techniques. By giving a set of component services and a specified requirement (e.g., user's request), a composite service specification can be generated automatically [7]. However, realizing a fully automatic service composition is still very difficult and presents several open issues [7], [4], [5]. The basic weakness of most research efforts proposed so far is that Web services do not share a full understanding of their semantics, which largely affects the automatic selection of services. Currently, the first results on automatic composition of Web services are those presented in [1], [3], [32], [33]. In a latest effort reported in [33], the classical semantic matching of services using Description Logic is augmented with *Concept Covering* and *Concept Abduction*. This allows users to obtain an explanation of what concepts are missing when no-exact matches are found for composition. The explanation provides users opportunities to refine composite services.

There exist some research efforts that leverage manual and automatic compositions. Instead of coupling component services tightly in the service model, such approaches feature a high-level abstraction of the process models at the design time, while the concrete composite services are either generated automatically using tools or decided dynamically at run time. For example, in [34], authors propose some model-driven approaches for Web services composition. A completed executable service specification (e.g., BPEL [17]) can be generated from the composite service specification (e.g., UML activity model, protocol specifications, and interface). In the Modeling Web Service Composition and Execution (MoSCoE) project,[8] composition is achieved by providing high-level abstraction, mashing of component services, and iterative refinement. Once a satisfactory composite service is realized, it can be translated to BPEL execution codes. Our approach is similar to MoSCoE in the sense that we also adopt a semiautomatic approach to service composition. Our composite services are specified as *process schemas* not high-level goals and the component services are selected, at run time, based on nonfunctional properties (e.g., QoS parameters, contextual) constraints specified by users.

6.2 Techniques for Service Orchestration

Composite services orchestration is a very active area of research and development. The underlying execution models of the projects (e.g., [22]) are based on a centralized process execution engine that is responsible for scheduling, dispatching, and controlling the execution of all the instances of a composite service. This contrasts with our work's distributed execution approach where the control-flow and data-flow notifications are directly exchanged between participating component services without going through the composite service.

Our work's distributed orchestration model has some similarities with the work presented in [23], which proposes a decentralized orchestration approach where a composite Web service specification is partitioned and executed at distributed locations. The approach, however, differs from our work, in that it is only applicable when the assignment of activities to their executing entities is known during the deployment of the workflow, which is a restrictive assumption in the context of service composition where providers can leave and join a community or alter the characteristics of their offers (e.g., the QoS or the price) after the composite service has been defined and deployed. In addition, it depends on full-fledged execution engines (e.g., BPWS4J[9]) for the execution of partitions, which contrasts with our lightweight coordination services. SelfServ [15] and OSIRIS [35] also feature a distributed, peer-to-peer service orchestration model. OSIRIS proposes a distributed process engine that routes process instances directly from one node to the next ones. The metainformation of a composite service is maintained in global repositories and distributed to participating nodes. The generation of such meta information from composite service specifications, however, is not given. In SelfServ, the responsibility of coordinating the execution of a composite service is distributed across several software components called *coordinators*. Compared to SelfServ, our work is one step further. Our orchestration model takes advantage of powerful coordination model of tuple spaces, which provides direct support of asynchronous interactions among participating services and users. In addition, our design of service orchestration control tuples considers both control flow and data dependencies of composite services. As a result, the communication requirements can be minimized.

6.3 Techniques Dealing with Service Adaptability

Several techniques have been proposed to deal with adaptability of composite Web services. The work of Miller et al. [36] is one of the first works that addresses the exception problems for Web-based workflow applications, but it provides only a classification of exceptions. Both BPEL4WS [17] and WSMF [37] offer the ability for exception handling by using *fault/compensation handlers* and *exception handling section*, respectively. However, their exception handling specifications are incorporated tightly with the composite service specifications that is difficult to maintain and evolve. In contrast, our Web service composition model provides a policy-based, multilevel exception handling approach that expresses and controls exception handling strategies at a high level of abstraction, separated from the composite services functionalities. Brambilla et al. [27] presents a high-level approach for the management of exceptions that occur during the execution of workflow-based Web applications. Authors provide a Web-based exception classification and a set of policies for capturing and recovering exceptions, which is quite similar to our exception handling model. However, the work is based on a modeling language called WebML,[10] which was initially designed for the development of Web page (i.e., hypertext)

8. http://www.moscoe.org/.

9. OASIS Business Transaction Protocol, http://www.oasis-open.org/business-transaction/.

10. Web modeling language, http://www.webml.org/.

based applications, not Web services. In a recent effort reported in [10], a platform has been developed where BPEL processes can be extended with policies and constraints for runtime configuration. The other two recent efforts for adaptive Web services composition, reported in [9], [11], focus on QoS-based, dynamic service selection for context-aware business processes.

Finally, Aspect Oriented Programming (AOP) [38] is emerging as a promising technique for realizing adaptive and flexible Web services composition. Nonfunctional composition properties (e.g., service selection policies) and new business rules are specified as aspects, which can be activated or deactivated appropriately at execution time. However, applying AOP to Web services composition is still in its early stage. Some first efforts are reported in [39], [40], [41], [42]. In particular, [40] is a recent effort to enhance BPEL orchestration engine with AOP technique to enable adaptive service provisioning. The emphasis is on dynamic modification of the BPEL specification to allow on-demand debugging, execution monitoring, or an application specific GUI. The hooks for plugging the appropriate aspects at runtime are limited to what is expressible in XPath. It is still very tedious and complex for users to specify aspects using XML or Java classes.

7 Conclusion

In this paper, we have presented CCAP, a system that supports configurable and adaptive composition of Web services. Firstly, we introduced a configurable service composition model. The innovative aspect of our model is to provide distinct abstractions for service context and exceptions, which can be embedded or plugged into the process schemas through simple interaction with end users. This departs from existing approaches in the area of service composition, which rely on scripting languages or process modeling notations (e.g., state diagrams, Petri nets, process algebra) without taking into account context awareness and runtime exception handling. Secondly, we proposed an execution model where statechart-based composite service specifications are transformed into a set of control tuples, represented as ECA rules. Since control tuples are stored separately from a composite Web service specification, it provides the possibility of distributing control tuples to participating Web services, thereby realizing a fully distributed execution of composite services. Thirdly, CCAP has been implemented using a number of state-of-the-art technologies and is fully functional. We also conducted an extensive performance study to validate the feasibility and benefits of our approach. In particular, we introduced a usability study to evaluate the learnability, efficiency, and user adoption of our approach, which, to the best of our knowledge, is one of the few works that provide evaluations of Web services composition approaches using real users.

Experience with CCAP has shown that our system can simplify the specification and deployment of composite Web services, can provide asynchronous service orchestration for distributed, loosely coupled participating services, and can favor robust and adaptive service execution in highly dynamic environments. These encouraging results are stimulating a number of further researches to extend the current prototype. First, a possible extension to CCAP is a mechanism for seamlessly accessing services among multiple computing devices. Indeed, during the invocation of a Web service, especially one having long running business activities or with complex tasks (e.g., composite services), users are more likely to be switching from device to device (e.g., from office PC to PDA). Applications cannot be allowed to terminate and start again simply because users change devices and users should not experience a break during the service invocation while they are moving. This is extremely important for people in time critical working environments (e.g., doctors in hospitals). Second, it is important to extend CCAP to support team work so that multiple (mobile) users can virtually collaborate with each other in same business processes. Finally, it is also interesting to add more flexibility to CCAP (beyond its exception handling capability) by supporting runtime modifications to the schema of a composite service (e.g., removing a task), and build more applications on top of CCAP, to further study the performance of the platform for large-scale service compositions.

Acknowledgment

Quan Z. Sheng's work has been partially supported by Australian Research Council (ARC) Discovery Grant DP0878367. The authors would like to thank Manoj Chandra, Eileen Oi-Yan Mak, Nathan Wong, and Daniel Pak for their participation in the implementation of CCAP.

References

[1] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava, "A Service Creation Environment Based on End to End Composition of Web Services," *Proc. 14th Int'l Conf. World Wide Web (WWW '05),* May 2005.

[2] A. Charfi and M. Mezini, "Middleware Services for Web Service Compositions," *Proc. 14th Int'l World Wide Web Conf. (WWW '05),* May 2005.

[3] B. Medjahed, "Semantic Web Enabled Composition of Web Services," PhD dissertation, Virginia Polytechnic Inst. and State Univ., 2004.

[4] N. Milanovic and M. Malek, "Current Solutions for Web Service Composition," *IEEE Internet Computing,* vol. 8, no. 6, pp. 51-59, Nov./Dec. 2004.

[5] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed, "Deploying and Managing Web Services: Issues, Solutions, and Directions," *The VLDB J.,* vol. 17, no. 3, pp. 537-572, 2008.

[6] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," *Computer,* vol. 40, no. 11, pp. 38-45, Nov. 2007.

[7] D. Berardi, G.D. Giacomo, and D. Calvanese, "Automatic Composition of Process-Based Web Services: A Challenge," *Proc. 14th Int'l World Wide Web Conf. (WWW '05),* May 2005.

[8] Q.Z. Sheng, "Composite Web Services Provisioning in Dynamic Environments," PhD dissertation, The Univ. of New South Wales, 2006.

[9] D. Ardagna and B. Pernici, "Adaptive Service Composition in Flexible Processes," *IEEE Trans. Software Eng.,* vol. 33, no. 6, pp. 369-384, June 2007.

[10] L. Baresi, E. Di Nitto, C. Ghezzi, and S. Guinea, "A Framework for the Deployment of Adaptable Web Service Compositions," *Service Oriented Computing and Applications,* vol. 1, no. 1, pp. 75-91, 2007.

[11] G. Canfora, M. Di Penta, R. Esposito, and M. Villani, "A Framework for QoS-Aware Binding and Re-Binding of Composite Web Services," *J. Systems and Software,* vol. 81, no. 10, pp. 1754-1769, 2008.

[12] Q.Z. Sheng, B. Benatallah, Z. Maamar, M. Dumas, and A.H. Ngu, "Enabling Personalized Composition and Adaptive Provisioning of Web Services," *Proc. 16th Int'l Conf. Advanced Information Systems Eng. (CAiSE '04),* June 2004.
[13] T. Fjellheim, S. Milliner, M. Dumas, and J. Vayssière, "A Process-Based Methodology for Designing Event-Based Mobile Composite Applications," *Data & Knowledge Eng.,* vol. 61, no. 1, pp. 6-22, 2007.
[14] The Unified Modeling Language (UML) Version 1.5, http://www.omg.org/technology/documents/formal/uml.htm, 2009.
[15] B. Benatallah, Q.Z. Sheng, and M. Dumas, "The Self-Serv Environment for Web Services Composition," *IEEE Internet Computing,* vol. 7, no. 1, pp. 40-48, Jan./Feb. 2003.
[16] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 4, pp. 293-333, Oct. 1996.
[17] T. Andrews et al., "Business Process Execution Language for Web Services 1.1," http://www-106.ibm.com/developerworks/library/ws-bpel, 2009.
[18] J. Clark and S. DeRose, "XML Path Language (XPATH) Version 1.0," http://www.w3.org/TR/xpath, Nov. 1999.
[19] Q.Z. Sheng and B. Benatallah, "ContextUML: A UML-Based Modeling Language for Model-Driven Context-Aware Web Service Development," *Proc. Fourth Int'l Conf. Mobile Business (ICMB '05),* July 2005.
[20] J.F. Allen, "Maintaining Knowledge about Temporal Intervals," *Comm. ACM,* vol. 26, no. 11, pp. 832-843, 1983.
[21] R. Montanari, E. Lupu, and C. Stefanelli, "Policy-Based Dynamic Reconfiguration of Mobile-Code Applications," *Computer,* vol. 37, no. 7, pp. 73-80, July 2004.
[22] S.R. Ponnekanti and A. Fox, "SWORD: A Developer Toolkit for Web Service Composition," *Proc. 11th Int'l World Wide Web Conf.,* May 2002.
[23] G.B. Chafle, S. Chandra, V. Mann, and M.G. Nanda, "Decentralized Orchestration of Composite Web Services," *Proc. 13th Int'l World Wide Web Conf. (WWW '04),* May 2004.
[24] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *Computer,* vol. 19, no. 8, pp. 26-34, Aug. 1986.
[25] N.W. Paton and O. Díaz, "Active Database Systems," *ACM Computing Surveys,* vol. 31, no. 1, pp. 63-103, 1999.
[26] U. Dayal, M. Hsu, and R. Ladin, "Organizing Long-Running Activities with Triggers and Transactions," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '90),* May 1990.
[27] M. Brambilla, S. Ceri, S. Comai, and C. Tziviskou, "Exception Handling in Workflow-Driven Web Applications," *Proc. 14th Int'l Conf. World Wide Web (WWW '05),* May 2005.
[28] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing,* vol. 6, no. 2, pp. 86-93, Mar./Apr. 2002.
[29] IBM WSTK Toolkit, http://alphaworks.ibm.com/tech/webservicestoolkit, 2009.
[30] P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford, "T Spaces," *IBM Systems J.,* vol. 37, no. 3, pp. 454-474, 1998.
[31] D. Salber, A.K. Dey, and G.D. Abowd, "The Context Toolkit: Aiding the Development of Context-Enabled Applications," *Proc. Conf. Human Factors in Computing Systems (CHI '99),* May 1999.
[32] P. Traverso and M. Pistore, "Automated Composition of Semantic Web Services into Executable Processes," *Proc. Third Int'l Semantic Web Conf. (ISWS '04),* Nov. 2004.
[33] A. Ragone, T.D. Noia, E.D. Sciascio, F.M. Donini, S. Colucci, and F. Colasuonno, "Fully Automated Web Services Discovery and Composition through Concept Covering and Concept Abduction," *Int'l J. Web Services Research,* vol. 4, no. 3, pp. 85-112, 2007.
[34] D. Skogan, R. Gronmo, and I. Solheim, "Web Service Composition in UML," *Proc. Eighth Int'l IEEE Enterprise Distributed Object Computing Conf. (EDOC '04),* Sept. 2004.
[35] C. Schuler, R. Weber, H. Schuldt, and H.-J. Schek, "Peer-to-Peer Process Execution with Osiris," *Proc. First Int'l Conf. Service-Oriented Computing (ICSOC '03),* Dec. 2003.
[36] J. Miller, A. Sheth, K. Kochut, and Z. Luo, "Recovery Issues in Web-Based Workflow," *Proc. 12th Int'l Conf. Computer Applications in Industry and Eng. (CAINE '99),* Nov. 1999.
[37] D. Fensel and C. Bussler, "The Web Service Modeling Framework WSMF," *Electronic Commerce Research and Applications,* vol. 1, no. 2, pp. 113-137, 2002.
[38] G. Murphy and C. Schwanninger, "Aspect-Oriented Programming," *IEEE Software,* vol. 23, no. 1, pp. 20-23, Jan./Feb. 2006.
[39] A. Charfi and S. Kloppenburg, "Aspect-Oriented Web Service Composition in AO4BPEL," *Proc. Fifth Int'l Conf. Aspect-Oriented Software Development (AOSD '06),* Mar. 2006.
[40] C. Courbis and A. Finkelstein, "Weaving Aspects into Web Service Orchestrations," *Proc. 2005 IEEE Int'l Conf. Web Services (ICWS '05),* July 2005.
[41] W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati, "An Aspect-Oriented Framework for Service Adaptation," *Proc. Fourth Int'l Conf. Service-Oriented Computing (ICSOC '06),* Dec. 2006.
[42] B. Verheecke, W. Vanderperren, and V. Jonckers, "Unraveling Crosscutting Concerns in Web Services Middleware," *IEEE Software,* vol. 23, no. 1, pp. 42-50, Jan./Feb. 2006.

Quan Z. Sheng received the PhD degree in computer science from the University of New South Wales in 2006. He is a lecturer in the School of Computer Science at the University of Adelaide. His research interests include service-oriented architectures, distributed computing, and pervasive computing. He was the recipient of the Microsoft Research Fellowship in 2003. He is the author of more than 40 publications. He served on program committees for dozens of conferences and was the program cochair of the IEEE SITIS 2008, the publication chair of the WISE 2005, and the publicity co-chair of the ICSOC 2005 and the WISE 2007. He is a member of the IEEE and the ACM.

Boualem Benatallah received the PhD degree in computer science from the University of Grenoble, France. He is a professor in the School of Computer Science and Engineering at the University of New South Wales, where he is also the founder of the Service-Oriented Computing Research Group. His research interests include Web service protocols analysis and management, enterprise services integration, process modeling, and service-oriented architectures for pervasive computing. He is a member of the IEEE.

Zakaria Maamar received the MSc and PhD degrees in computer science from Laval University, Quebec, Canada, in 1995 and 1998, respectively. He is an associate professor at the College of Information Technology of Zayed University in Dubai, U.A.E. Prior to joining ZU, he held a defense scientist position with the Defense Research Establishment Valcartier in Quebec, Canada. His research interests are related to software agents, context-aware computing, and Web services.

Anne H.H. Ngu is currently an associate professor with the Department of Computer Science at Texas State University-San Marcos. Her main research interests are in information integration over the Web, service-oriented computing, databases, scientific workflows, and agent technologies. The unifying theme of her research has been to provide transparent global access to heterogeneous, distributed autonomous information sources. From 1992 to 2000, she worked as a senior lecturer in the School of Computer Science and Engineering, the University of New South Wales. She had held research scientist positions with Telecordia Technologies and MCC. She had also been a summer faculty scholar in Lawrence Livermore National Laboratory from 2003 to 2006.