SOCA (2007) 1:197–212
DOI 10.1007/s11761-007-0016-x

ORIGINAL RESEARCH

An architecture for rapid, on-demand service composition

Maja Vuković · Evangelos Kotsovinos · Peter Robinson

Received: 1 August 2007 / Revised: 24 September 2007 / Accepted: 26 September 2007 / Published online: 1 November 2007

Abstract Legacy application design models, which are still widely used for developing context-aware applications, incur important limitations. Firstly, embedding contextual dependencies in the form of if–then rules specifying how applications should react to context changes is impractical to accommodate the large variety of possibly even unanticipated context types and their values. Additionally, application development is complicated and challenging, as programmers have to manually determine and encode the associations of all possible combinations of context parameters with application behaviour. In this paper we propose a framework for building context aware applications on-demand, as dynamically composed sequences of calls to services. We present the design and implementation of our system, which employs goal-oriented inferencing for assembling composite services, dynamically monitors their execution, and adapts applications to deal with contex- tual changes. We describe the failure recovery mechanisms we have implemented, allowing the deployment of the system in a non-perfect environment, and avoiding the delays inherent in re-discovering a suitable service instance. By means of experimental evaluation in a realistic infotainment application, we demonstrate the potential of the proposed solution an effective, efficient, and scalable approach.

Keywords Service composition · Context awareness · AI planning

This work was supported by IBM Zurich Research Laboratory.

M. Vuković (✉)
IBM T.J.Watson Research, 19 Skyline Drive,
Hawthorne, NY 10532, USA
e-mail: mvukovi@us.ibm.com

E. Kotsovinos
Deutsche Telekom Laboratories, Ernst-Reuter-Platz 7,
10587 Berlin, Germany
e-mail: evangelos.kotsovinos@telekom.de

P. Robinson
University of Cambridge, 15 JJ Thomson Avenue,
Cambridge CB3 0FD, UK

1 Introduction

Smaller and universally connected computing devices, advances in sensing technologies, and the development of knowledge extraction and management capabilities lead to an environment rich in *contextual information* such as location, physiological state, and motion. As a result, applications now operate in a variety of new settings; for example, embedded in cars or wearable devices. They use information about their context to respond and adapt to changes in the computing environment, moving towards the vision of truly ubiquitous computing [1]. They are, in short, increasingly *context aware*.

However, advances in application models to support the development of context aware systems have not kept up. Such applications are often built in a scenario-specific manner, encoding the anticipated context types and desired application behaviour. This presents important disadvantages. Firstly, embedding contextual dependencies in the form of if–then rules specifying how applications should react to context changes is only feasible for the set of context types anticipated at the time of application design. Systems built this way are not able to accommodate the large variety of possibly even unanticipated context types and their values. Additionally, application development is complicated and challenging, as programmers have to manually determine and encode the associations of all possible combinations of context parameters with application behaviour. Finally, reprogramming

scenario-specific context awareness in each system reduces code reusability and robustness.

This work presents our design and implementation of a framework that employs AI planning to *rapidly assemble applications* on-demand from individual services, based on context and user goals. Contrary to most other service composition frameworks, our system supports *dynamic on-the-fly adaptation* of applications, and comprehensive *failure recovery* mechanisms. Applications built using our system are able to deal with run-time changes in the environment in which they operate, and to continue operating even when some of their components get unpredictably disconnected. Experimental evaluation of our framework demonstrates that it provides an *efficient* and *scalable* solution, allowing for rapid composition and deployment of complex services even in realistically large and complex pervasive computing environments.

Applications built using our system enjoy the following advantages:

- *Ease of development.* Our framework allows combining individual services flexibly and conveniently, in a goal-oriented fashion, to perform complex context-aware tasks on-demand
- *Ease of maintenance.* Our system allows extending applications to cope with new types of context information (e.g. new devices coming to the market) without requiring significant reprogramming
- *On-the-fly adaptation.* Our system monitors changes in contextual information, and ensures that application behaviour is dynamically adapted accordingly, without compromising performance and scalability
- *Failure resilience.* Our framework incorporates mechanisms for automatic recovery from failures that may occur during the composition and execution process

This paper extends [2] and is structured as follows. An example scenario demonstrating how we envisage our system to operate from a user's point of view is examined in Sect. 2. The architecture of our framework for context aware service composition is analysed in Sect. 3 and its implementation is described in Sect. 4. Section 5 discusses our prototype implementation and evaluation results. Section 6 positions our work in the research context of pervasive computing, and discusses shortcomings of previous service composition frameworks. Finally, Section 7 presents our conclusions and outlines areas of future work.

In previous work we addressed failures resulting from unsuccessful composition or execution of composite services [3] and investigated the applicability of planning systems to the service composition problem [4].

2 Usage scenario

To illustrate how the proposed framework can simplify the development of context aware applications, this section introduces the following scenario in the scope of an infotainment application. A user, called Miles, subscribes to an infotainment portal provided by his mobile network operator. This portal offers users a broad array of on-line services, including a *Restaurant Finder* service, which provides a directory of available restaurants, a *Directions Finder* service, which computes the driving directions to a given restaurant, a *Translator* service, which translates the content from one language to another, and a *Speech Synthesizer* service, which converts from text format to speech.

Users such as Miles can place requests to services provided by the portal on-line and receive the corresponding information. User requests are enriched with context information regarding the *location* of the user, the current *activity* of the user, and the *type of the computing device* used to make the request. The above information is provided by a context middleware solution.

Use case 1. As shown in Fig. 1, Miles is initially using his SmartPhone while walking around Market Square in Cambridge, UK. He places a request to the infotainment portal for

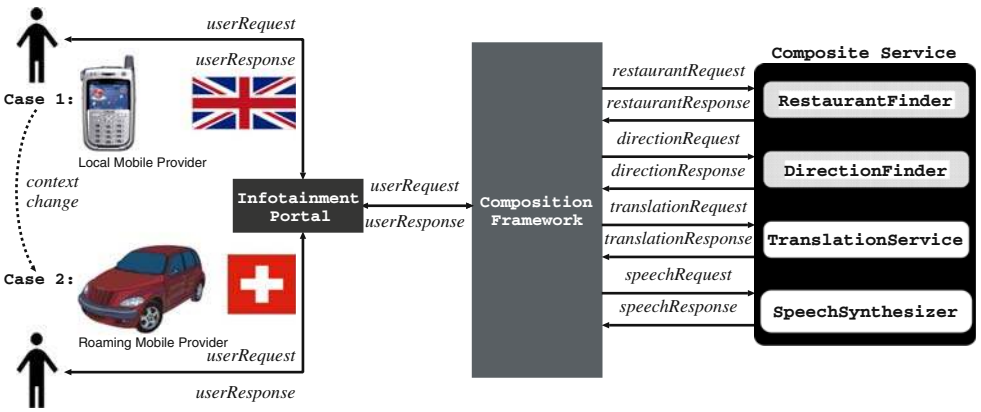

Fig. 1 Usage scenario: context aware restaurant finder

finding a restaurant of his liking, and providing directions to it. The portal uses our composition framework to assemble a composite service to deal with Miles' request. The resulting service, tailored to help Miles locate a Spanish restaurant, is composed from the atomic services *Restaurant Finder* and *Directions Finder*.

Use case 2. At a later point in time, Miles is in Zurich, Switzerland, and wishes to locate a restaurant of his liking. Miles is now registered with a Swiss infotainment portal, which has a roaming agreement with Miles' mobile network operator. Miles speaks only English, but the Swiss portal provides the local restaurant guide service only in the German, French and Italian languages. Our framework takes information about the services available, Miles' request and restrictions, and context, and assembles a service for Miles consisting of the atomic services *RestaurantFinder*, *DirectionsFinder*, and *TranslationService*. The information is delivered to Miles' mobile phone.

A few minutes later, Miles' context changes from "walking" to "driving", as he collects a rental car and starts driving to the restaurant. Our system notices the change, and adds the *SpeechSynthesizer* service to the application, reformatting the directions and routing them to Miles' in-vehicle information system (IVIS) for speech delivery.

In all above cases, Miles has the same goal: he wishes to find and get directions to a restaurant of his liking. However, the two requests result in the composite services being constructed from different atomic services, because of the different context in which the requests are submitted. In the first case, Miles is using a *SmartPhone* while *walking* around Market Square in *Cambridge*. In contrast, in the second case, Miles is *driving* through *Zurich*, and using *IVIS*. Our framework allows Miles to submit both requests in the same way, and automatically handles application adaptation.

3 System architecture

This section describes the architecture of the proposed framework using the scenario described in Sect. 2.

The overall operation of our system is shown in Fig. 2. The inputs to the system are a user request, a number of available services, and the context. When a user request is made, the *composition request management* layer combines this with contextual information to create a composition request for the *abstract service composition* layer. This in turn assembles a composite service, and passes it on to the *architecture-specific service composition* layer, which binds it to a sequence of deployable service instances. Finally, the *execution and monitoring* layer invokes the above sequence, monitors its execution, and triggers adaptation when context or execution parameters change.

Composition request management. The composition process starts with the receipt of a user request (Step 1 in Fig. 2). Based on that, this step creates a *composition request*; this includes a number of *literals* specifying the userÆs *task intention* (e.g. "find me directions to a restaurant") and contextual parameters (e.g. "in Zurich", "driving"). The request is then fed to the abstract service composition layer (Step 4) for building an abstract plan.

Abstract service composition. This layer translates the composition request and available service descriptions to a *problem definition*, which is in the representation format supported by the composition technology in use —TLPlan [5] in our prototype—(Steps 7 and 8). Then the problem definition is passed to the composition engine for building an *abstract plan* (Step 9a). This denotes a sequence of *abstract services*, high-level descriptions of service operations and cannot be directly invoked, which are bound to service instances in the following layer. Potential failures of the composition process are handled by GoalMorph [3], a system which transforms failed composition requests into alternative ones that can be solved (Step 9b).

Architecture-specific service composition. Starting from the abstract plan, this layer translates it into an *abstract execution plan*, which is represented in the language used by the execution framework employed—BPEL4WS [6] in our prototype—(Step 10). The *Plan Instantiator*—implemented using the BPWS4J engine [7], uses the *Service Registry*—implemented using jUDDI v0.94 [8] to discover *service instances*, which are realized as calls to fine granularity Web services, binding to the abstract services in the abstract execution plan (Step 7). This produces the *deployable service description*, a deployable composition of service instances, which is passed on to the execution and monitoring layer (Step 12a).

Execution and monitoring. The layer *executes* and *continuously monitors* the execution of the deployable service description. Upon disconnection or failure of individual service instances, the layer passes control back to the architecture-specific service composition layer, which can substitute the failed service instances with new ones (Step 15a). Should an unanticipated change in context occur (Step 15b), or should the user change the task specification (Step 15c), control is passed to the composition request management layer, where a new composition request is generated and recomposition triggered.

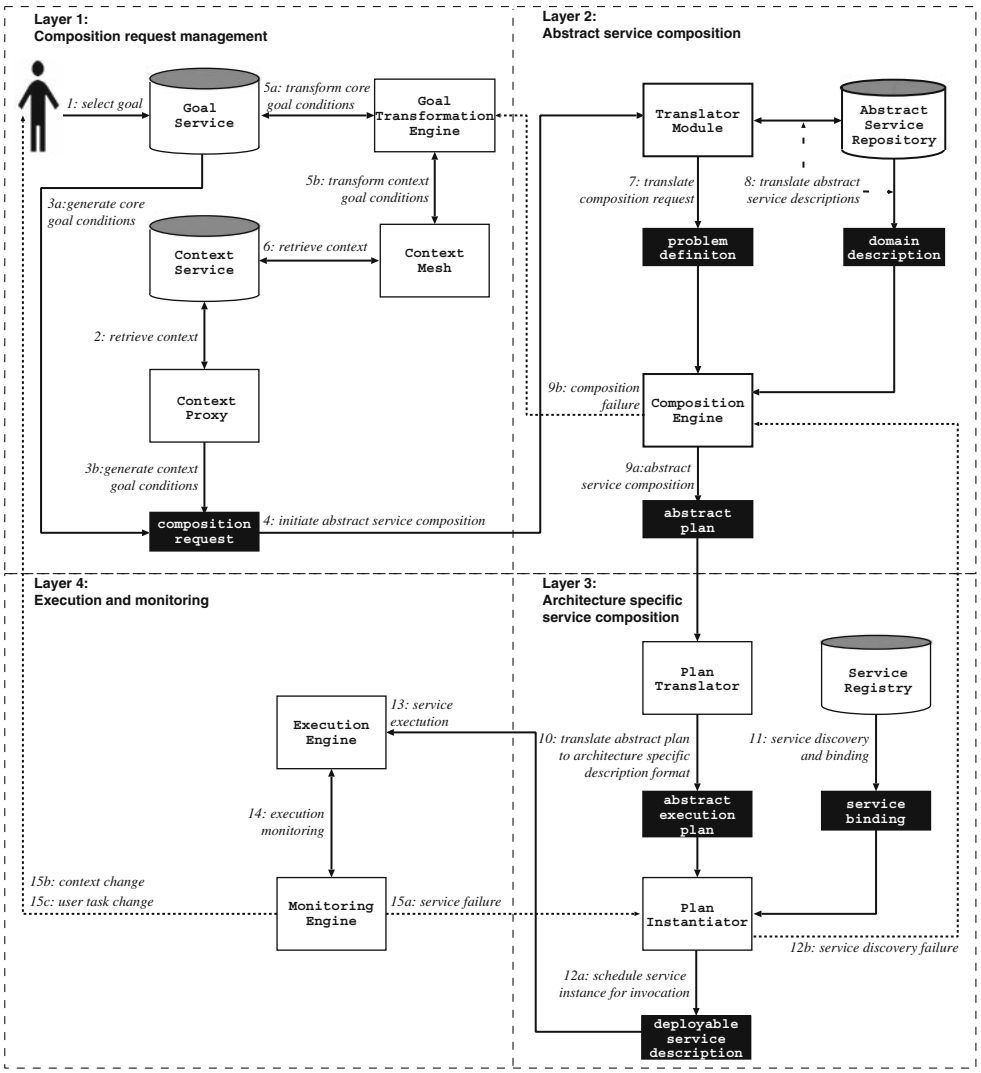

Fig. 2 System architecture overview

Context changes during the execution of composite service may (a) unexpectedly satisfy effects of scheduled services or (b) invalidate preconditions that were true at the time of abstract composition. For instance, a user may manually adjust the volume of the music in the car. As a result the effect of the scheduled service for lowering music volume is satisfied, and the service should not be executed. The monitoring process observes context changes and service execution using the monitoring model proposed by Haigh et al. [9]. For example, if the main service effect has been unexpectedly satisfied, such as the user manually lowers the music volume, the execution state is updated and the scheduled service for controlling music volume is not invoked. Observing the environment and maintaining a state description in this way improves the efficiency of the system because it will not attempt redundant service executions. Mechanisms for handling composition and execution failures are analysed in [10].

4 Implementation

This section describes how the prototype implementation applies goal-oriented inferencing from the TLPlan [5] planning algorithm to select atomic services that form a composite Web service. It also presents how the `abstract execution plan` is described in BPEL4WS [6] format. Finally it discusses the internals of the execution and monitoring layer.

4.1 Composition Engine: TLPlan for Web service composition

TLPlan is used to synthesise plans in the domain of our usage scenario, which has been described in Sect. 2. The fundamental steps in planning include describing the planning domain, specifying the initial and goal worlds, and invoking the planning process.

```
(declare-described-symbols
    ;; Restaurant ontology
    (predicate catering_facility 1)
    (predicate catering_facility_take_away 1)
    (predicate catering_facility_home_delivery 1)
    ...
    (predicate catering_facility_restaurant 1)
    (predicate catering_facility_bistro 1)
    (predicate catering_facility_cafeteria 1)

    ;; Restaurant operation effects
    (predicate restaurant_found 1)
    (predicate restaurant_booking_made 2)

    ;; Restaurant properties
    (predicate restaurant_name 1)
    (predicate restaurant_address 2)
    (predicate restaurant_email 2)
    (predicate restaurant_website 2)
    (predicate restaurant_smoking 2)
    (predicate restaurant_cuisine 2)
    (predicate persons 1)
    (predicate time 1)

    ;; Device properties
    (predicate volume_level 1)
    (predicate brightness_level 1)
    ...
    )

;; Comment: declared symbols (declare_defined_symbols
    (predicate restaurant_has_space 2)
    ...
)

;; Comment: symbol definition ;;; Restaurant-Has-Space: True iff the
table has space.

(def-defined-predicate (restaurant_has_space)
    (exists (?r ?n) (restaurant_name ?r)
        (restaurant_space ?r ?n) (> ?n 0)))
```

Fig. 3 Sample domain description in TLPlan

Domain description. A `domain description` contains details about the literals, predicates and function symbols to be used in the domain. Figure 3 shows literals describing concepts in the usage scenario. For example, the literal `(predicate restaurant_booking_made 2)` is used to describe the effect of a booking being made. It has arity 2, where arguments represent the number of persons and the time for which the booking was made. The literal `(predicate restaurant_has_space 2)` is used to determine if the booking can be made (Fig. 4).

Problem definition. The problem definition specifies the initial world and the goal world, using lists of domain predicates and function definitions. Figure 5 shows a sample problem definition for the usage scenario, e.g. goal conditions `(directions_found current_address restaurant_address)` and `(direction speech_out)` are states to be reached. The core part of this request is finding the directions, and the context goal is that directions should be read out in audio form, as the user is currently driving.

Plan. The planner is invoked by loading the `domain description` file and the `problem definition` file.

```
(def-adl-operator
(make_restaurant_booking ?r ?ppl ?t)
      (pre
          (restaurant ?r)
          (restaurant_found ?r)
          (restaurant_booking_online ?r ?e)
          (restaurant_has_space ?r ?ppl)
          (persons ?ppl)
          (time ?t)
          (and
              (not (restaurant_booking_made ?ppl ?t))
              (not (restaurant_booked ?r))
              (restaurant ?r)
              (persons ?ppl)
              (time ?t)))
      (add
          (restaurant_booking_made ?ppl ?t)
          (restaurant_booked ?r)
      )
)
```

Fig. 4 Sample TLPlan operator

```
; Initial world (define (initial_state_Case_3)
        (cuisine lebanese)
        (location zurich)
        ...
        (persons 2)
        (time 2000)
        (activity driving)
)


; Goal world (define (goal_state_Case_3)
        (restaurant_found_location lebanese zurich)
        (restaurant_booking_made 2 2000)
        (restaurant_booked restaurant_name)
        (directions_found current_address restaurant_address)
        (directions speech_out)
)
```

Fig. 5 Sample TLPlan problem

The resulting plan is a list of operators and a sequence in which they should be applied. Figure 6 shows the sample output given the problem in Fig. 5.

4.2 Representation of abstract execution plans

This layer converts the `abstract plan` into the `abstract execution plan`, which is represented in the architecture specific language. To express the logic of a composite Web service the framework uses BPEL4WS, an XML-based flow composition language.

```
(get_restaurant_by_cuisine_location lebanese zurich)
(get_restaurant_address restaurant_name)
(make_restaurant_booking restaurant_name 2 2000)
(get_directions_door_to_door current_address restaurant_address)
(translate de en directions)
(txt2speech directions)
```

Fig. 6 Sample TLPlan plan

```
<process
    xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    name="RestaurantProcess"
    targetNamespace="urn:restaurant:main"
    xmlns:tns="urn:restaurant:main">

  <partners>
    <partner name="partner_user"
        xmlns:user_ns="urn:prototype:user"
        serviceLinkType=
            "user_ns:{urn:prototype:user}
                RestaurantDirectionsServiceComposition_SLT"/>

    <partner name="partner_plan_instantiator_proxy"
        xmlns:domain_ns="urn:prototype:restaurant"
        serviceLinkType=
            "domain_ns:{urn:prototype:restaurant}
                PlanInstantiatorProxy_SLT"/>
  </partners>
  ...
</process>
```

Fig. 7 Sample partner definition in Business Process Execution Langauge for Web Services

BPEL4WS models the interaction among participating Web services, termed *partners* in BPEL4WS, to describe composite Web services. It specifies the role of the partners providing each Web service and the flow of the messages they exchange. Figure 7 shows how partners are defined. In the current implementation partners are: `partner_user` and `partner_plan_instantiator_proxy`. Partner definition includes partner name, role, and the link to the service definition in its Web Service Description Language (WSDL) [11] file.

Figure 8 shows the `abstract execution plan` in the BPEL4WS format. The partner `partner_plan_instantiator_proxy` represents the **`Plan Instantiator`** component. Its `instantiate` operation is called for each service description in the `abstract execution plan`. As an input it takes the service description, Quality of Service parameters and the location of the **`Service Registries`**. It uses this information to perform service discovery and binding. For example, the `<invoke>` construct `invoke_restaurant-lookup-ch` takes as an input the variable `input_restaurant-lookup-ch`, which contains search and input parameters for a restaurant finder service instance.

The `abstract execution plan`, described in BPEL4WS is deployed on IBM Business Process Execution Language for Web Services Java Run Time (BPWS4J) v2.1 [7], a platform that executes BPEL4WS processes. The BPWS4J is a Web component that runs on an application server. This implementation of the framework uses the Tomcat v5.1 [12] application server.

The main limitation of BPWS4J is that it does not allow for dynamic binding and discovery of services. As an input it takes three parameters: (1) a BPEL4WS document that describes a composite service to be executed, (2) a WSDL document without binding information, which describes the interface that the composite service will present to clients or partners in BPEL4WS terms and (3) WSDL documents that describe the services that the composite service may invoke during its execution. An `abstract execution plan` contains an abstract service description. It does not have information about service instances and their WSDL documents, which are a necessary parameter to BPWS4J as described above.

To address this limitation, the framework introduces the **`Plan Instantiator`** component. The BPWS4J uses the **`Plan Instantiator`** as a proxy to communicate with **`Service Registries`** to obtain WSDL files and instantiate services. This is achieved by encapsulating service search parameters as an input to the `instantiate` operation of the partner `partner_plan_instantiator_proxy`, shown in Fig. 8.

4.3 Execution and monitoring layer

This section describes how the framework mediates the interaction between the composition layers and the execution environment. It presents how the framework adapts and applies the monitoring model proposed by Haigh et al. [9], which includes *service monitors* that observe service execution, and *event monitors* that track changes in the environment.

The **`Execution Engine`** schedules and invokes service instances, which are defined by `deployable service descriptions`. During service execution the environment is changing and can therefore invalidate the facts, which are used by the **`Composition Engine`** to assemble a composite service. The purpose of the **`Monitoring Engine`** is to provide the **`Composition Engine`** and the **`Execution Engine`** with an up-to-date view of the state of the execution environment.

The execution of each service is embedded in a `monitoring procedure`, which verifies service preconditions and postconditions. The `monitoring procedure` is run sequentially, before and after service execution. For example, before a `RestaurantFinder` service is executed, the **`Monitoring Engine`** determines whether the cuisine type parameter has been supplied. If this is not the case, control is passed back to the user and the composition request management layer to acquire the missing parameters. Similarly, before executing the `DirectionFinder` service, if the current location is no longer available, control is passed to the composition request management layer to reformulate the request.

Once all necessary preconditions are satisfied, the service is invoked. The **`Monitoring Engine`** then examines the outcome of service execution and passes control back to the abstract service composition and the architecture specific service composition layers, if the actual outcome of service operation is not as expected. For example, if details

Fig. 8 Abstract execution plan in Business Process Execution Language for Web Services

```
<sequence name="RestaurantProcess_sequence">
    <receive name="receive_RestaurantProcess" partner="partner_user"
        xmlns:user_ns="urn:prototype:user"
        portType="user_ns:RestaurantDirectionsPT"
        operation="user_ns:getRestaurantDirections"
        variable="var_user">
    </receive>
    ...
    <invoke name="invoke_restaurant-lookup-ch" partner="plan_instantiator_proxy"
        xmlns:domain_ns="urn:prototype:restaurant"
        portType="domain_ns:Proxy_PT"
        operation="domain_ns:instantiate"
        inputVariable="input_restaurant-lookup-ch"
        outputVariable="output_restaurant-lookup-ch">
    </invoke>
    <assign >
        <copy>
        <from variable="output_restaurant-lookup-ch"
                part="restaurantName"/>
        <to variable="input_address-lookup-ch"
            part="restaurantName"/>
        </copy>
    </assign>
    ...
    <invoke name="invoke_direction-lookup-ch" partner="plan_instantiator_proxy"
        portType="domain_ns:Proxy_PT"
        operation="instantiate"
        inputVariable="input_direction-lookup-ch"
        outputVariable="output_direction-lookup-ch">
    </invoke>
    <assign >
        <copy>
        <from variable="output_direction-lookup-ch" part="directions"/>
        <to variable="var_user" part="directions"/>
        </copy>
    </assign>
    <reply name="reply_RestaurantProcess"  partner="partner_user"
        xmlns:user_ns="urn:prototype:user"
        portType="user_ns:RestaurantDirectionsPT"
        operation="user_ns:getRestaurantDirections"
        variable="var_user">
    </reply>
</sequence>
```

of restaurants are not produced as a result of executing the `RestaurantFinder` service, control is passed back to the abstract service composition and the architecture specific service composition layers. The **`Monitoring Engine`** also passes the information on the current state of the environment to the **`Composition Engine`**, which may trigger a recomposition of the request.

5 Evaluation

As service-oriented architectures gain popularity, more services will be made available by service providers, therefore increasing the size of the problem definition—the construct fed to the Composition Engine for assembling the composite service. Additionally, the number of composition requests is anticipated to increase over time as a wider user community takes advantage of the service composition framework. To be able to handle both the increase in the size of the problem definition and increase in the number of users and their requests, the framework must be able to *scale gracefully* and maintain its *performance* and *responsiveness*.

Ensuring that the system's computational requirements scale linearly in the above conditions enables the addition of computational resources on demand, as by doing so a linear performance improvement will be observed. This is a common scalability criterion in systems research [13]. The first part of this section presents evaluation results of quantitative

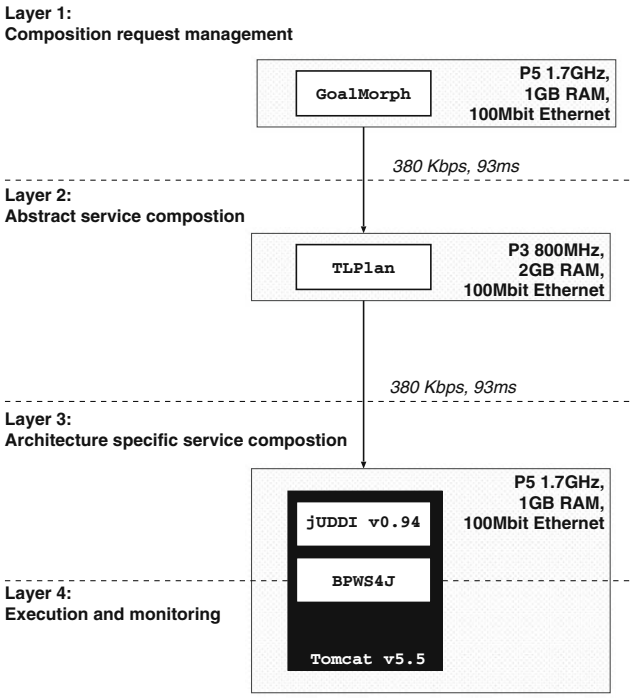

Fig. 9 Experimental setup

methods demonstrating the framework's *performance* and *scalability*.

At the same time, it is important to show that the design of the service composition framework does indeed make context aware applications *easier to build* and *maintain* by reducing their complexity and increase their extensibility. The second part of this section conducts a qualitative evaluation of the system, analysing how it helps reducing the *development effort* required for building context aware applications.

5.1 Quantitative evaluation

In this section we present our experimental results, which demonstrate that the framework is able to respond quickly and scale gracefully.

5.1.1 Experimental setup

Framework deployment. Figure 9 shows the configuration of the environment in which all experiments were conducted. Two machines were used hosting different layers of the framework. The machine providing access to the abstract service composition layer, including TLPlan, was a dual processor Pentium III 800 MHz with 2 GB RAM. An IBM Thinkpad T41 with an Intel Pentium M 1700 MHz processor and 1 GB RAM hosted the composition request management, architecture specific service composition, and execution and monitoring layers. Layers 3 and 4 were deployed on Tomcat v5.5 application server [12]. The two machines were in same LAN connected over a 100Mbps Ethernet.

Table 1 Sample service categorisation

Service type	Number of geographical categories	Number of semantic annotations	Number of UNSPCS codes
`RestaurantFinder`	1	3	2
`AddressFinder`	1	2	1
`DirectionsFinder`	1	3	4
`Translation-Service`	1	2	1
`SpeechSynthesizer`	1	2	1

Context-rich environment. All experiments were performed in a context-rich infotainment environment, corresponding to the one described in Sect. 2. This environment had the following characteristics:

- There were 20 sample abstract services in the Abstract Service Repository, such as Restaurant Finder and Directions Finder.
- The Service Registry contained a number of instances of each abstract service available in the Abstract Service Repository.
- The infotainment environment was represented in a problem definition containing 100 elements describing the scenario concepts, such as restaurant and its properties.
- Service instances were associated with two categories: geographical, describing the area in which each service is applicable, and the United Nations Standard Products and Services Code System (UNSPSC) [14] categorisation. These categorisations were stored in the Service Registry. Table 1 shows the number of categories that applied to each service instance. Additional structures were used to describe preconditions and postconditions of each service (i.e. semantic annotations).
- The following sample user requests were generated to test the system: "find a dining or entertainment venue (location-based)", "find an entertainment venue (event-based)", "find a dining venue (cuisine-based)", "find directions to the venue", "book dining or entertainment venue", and "make booking and find directions for dining and entertainment venues". Each request was enriched with context, such as location, device used, activity, social context, time and weather.

5.1.2 Performance

This experiment evaluated the performance of our system, both with and without the presence of service execution failures. Such failures may occur, for instance, when services

become unavailable as a result of network disconnection. Our system handles such failures either reactively or proactively, similar to the approach proposed by Gu et al. [15]. In particular, we evaluated the performance of our system in the following three cases:

- **Case 1.** The process of service composition occurred under ideal conditions, without any execution failures.
- **Case 2a.** Service execution failures caused control to be passed from the execution and monitoring layer back to the architecture specific composition layer, in which a replacement service was discovered, bound and scheduled for invocation. This is termed a *reactive* recovery method.
- **Case 2b.** Several service instances of each type were deployed, allowing the rapid switchover from the unavailable instance to another upon disconnection. This reduced the overhead of the discovery process once the execution failure occurred, as will be shown in the following section. This is termed a *proactive* recovery method.

We measured the *total framework operation* CPU time taken by the subsequent steps of the system's operation—as described in Sect. 3, including the time needed for recovery from any execution failures. The measurements were performed by obtaining snapshots of the total CPU time consumed by the system using JConfig [16] after each of the steps in the composition process as previously described. All measurements were repeated 20 times for a composition request containing 10 literals, with the resulting composite service containing 23 atomic services. The discovery process was performed with 160 service instances in total in the Service Registry. All measurements were rounded to the nearest milisecond.

The translation of abstract service descriptions was performed only once at the beginning of the overall evaluation, as each test case uses the same problem definition. Therefore the measurements do not include the time for this step. On average, this test takes approximately 4.3 seconds in our prototype.

Results. Figure 10 shows the total framework operation time for each one of the above test cases. On average the normal composition without any failures takes approximately 125 ms for a composite service consisting of 23 atomic services. Handling a composition failure in the reactive mode costs 20 ms on average, while using proactive approach in Case 2b reduces the failure recovery time by 15 ms—down 75% from the 20 ms it took in Case 2a. This underlines the significant performance—and thus user experience — benefit that using our proactive recovery approach provides, increasing

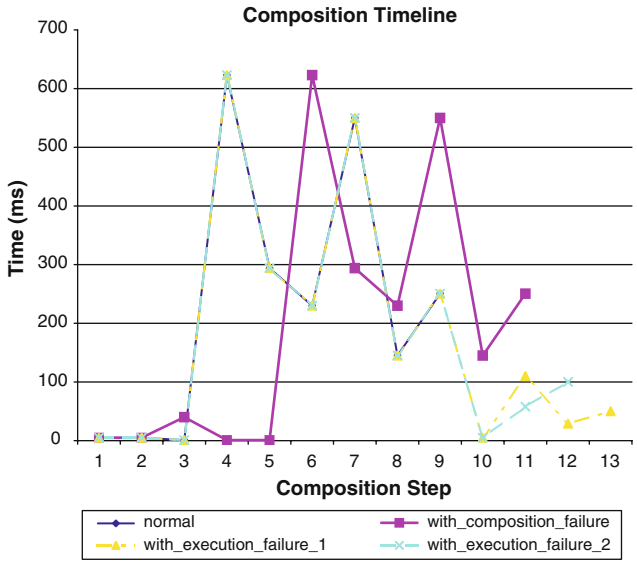

Fig. 10 Framework operation timeline for Cases 1, 2a, and 2b

the framework's suitability for interactive, latency-sensitive applications.

It is important to note that the reason why the architecture specific composition layer takes a disproportionate amount of time compared to the other steps is the low performance of the WSDL parser [17] used for generating the BPEL4WS file.

This experiment demonstrates that the framework *performs more than adequately well*, and *handles execution failures rapidly*, especially when proactive recovery is used. The above properties underline the suitability of the system for dynamic, wide-area pervasive computing environments.

5.1.3 Scalability

This set of experiments demonstrates the framework's ability to operate under increasing: (1) *problem definition size*, (2) *size of composition requests* (in terms of the literals they contain), and (3) *number of concurrent composition requests* submitted. The experimental configuration described in Sect. 5.1.1 was used to run the scalability tests. The complexity of the problem definition size was extended to contain 150 facts and 100 service types, to accommodate composite services consisting of up to 100 atomic services.

Scalability when increasing problem definition size. This experiment varied the number of service instances available in the Service Registry from 80 to 640. The rest of the parameters of the planning domain remained as described in Sect. 5.1.1, consisting of 100 facts and 20 abstract service types. For each stage of the problem definition growth a full composition process was executed to reconstruct the framework operation. At the same time the size of the resulting

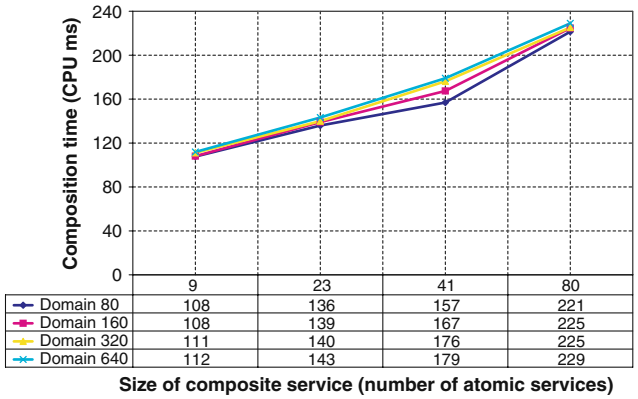

Fig. 11 Scalability when increasing problem definition size

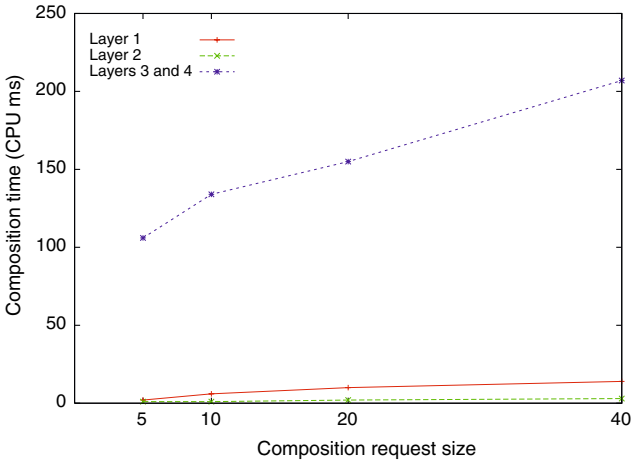

Fig. 12 Scalability when increasing the composition request size, for domain size 160.

Table 2 Composition request size and composite service size

Test cases				
Composition request size (number of goal conditions)	5	10	20	40
Number of context goal conditions	1	5	7	10
Average size of the resulting composite service	9	23	41	80

composite service was varied from 9 to 80. The composition process was invoked 20 times to measure the average CPU time needed by the system to assemble, deploy, and monitor the composite service.

Figure 11 shows the results of this experiment, demonstrating that the framework *scales gracefully* to a realistic problem definition (640 instances), while still requiring *less than 229 ms* of CPU time. Additionally, the experiment shows that our framework *scales linearly* as the problem definition size increases, allowing the on demand addition of computing resources to cope with larger pervasive environments.

Scalability when increasing composition request size. This experiment measured the impact of composition request size — in terms of the number of literals it contained — on the framework operation time. Table 2 shows the number of literals in the composition request in each test case — varied from 6 to 50, as well as the average size of the resulting composite service.

Figure 12 shows the average framework operation time and provides a breakdown of time taken by each layer of the framework. The framework processes an unusually large composition request of size 40, assembles, and invokes a complex composite service of size 80 in *less than 225 ms* of CPU time. Additionally, as in the previous experiment, the system is shown to *scale linearly*, allowing handling larger composition requests if needed to accommodate for realistic deployment environments.

Scalability when increasing number of composition requests. This experiment was conducted to measure the framework operation time when an increasing number of concurrent composition requests—varied from 1 to 100 — were submitted, in the environment described in Sect. 5.1.1. The BPWS4J engine v2.1, which was used in our prototype implementation, does not support programmatical deployment of more then one BPEL4WS file simultaneously, at the time of writing. Therefore the measurements focus on the scalability of the framework without the deployment of the BPEL4WS file.

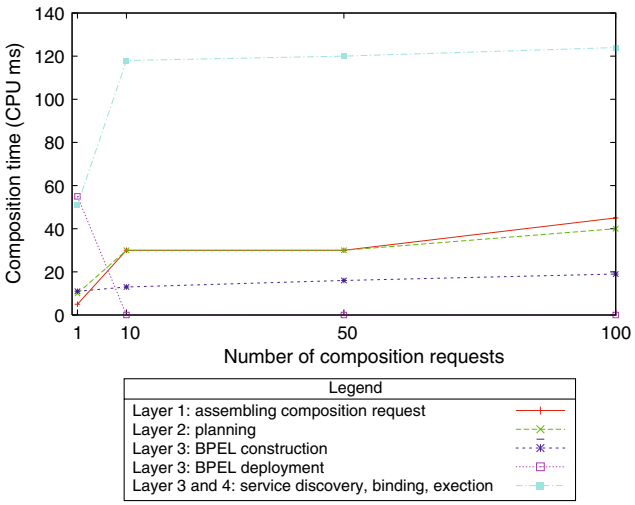

Fig. 13 Scalability when increasing number of composition requests

Figure 13 shows that a large difference in the number of requests submitted (100 to 1) makes only little difference in the total time taken to serve the request (2 to 1); the total time for 100 simultaneous composition requests is *less than twice the time for a single request*. This demonstrates the ability of the system to *scale gracefully* to accommodate large numbers of concurrent composition requests, thus making it suitable to large-scale, multi-user deployments.

Table 3 Scalability when increasing the composition request size: composition time distribution (CPU ms)

Step	1 request	10 requests	50 requests	100 requests
Layer 1: Composition request assembly	5	30	30	45
Layer 2: Abstract service composition	10	30	30	40
Layer 3: Generate BPEL4WS	11	13	16	19
Layer 3: Deploy BPEL4WS	55	n/a	n/a	n/a
Layers 3 and 4: Service discovery, binding, and execution scheduling	51	118	120	124
Total time (ms)	*132*	*191*	*196*	*228*

Table 3 analyses the result in a bit more depth, showing the distribution of time across the subsequent steps of our system's operation. The reported time for service discovery, binding and invocation is a total time for processing all the participating atomic services. The results have been rounded to one decimal place.

As the table shows, the bottleneck of the composition process is the generation of the BPEL4WS file, due to the low performance of the WSDL parser used [17]. Furthermore, the relatively small increase in the time taken for layers 3 and 4, shown in Table 3, results from the caching mechanism employed by the Service Registry, as the test data set contained composition requests consisting of randomised, but possibly overlapping, service instance queries.

5.2 Qualitative evaluation

At present, devising a set of standard and commonly accepted metrics for measuring development complexity remains an open research topic. Recent research has focused mainly on performance analysis and system complexity [18,19]. One of the challenges is that the complexity of a system can be viewed from the perspective of a number of different stakeholders. An application developer is concerned with development time, flexibility and extensibility of the application. A system administrator is concerned with the amount of management and configuration required to keep the application running. Most importantly, the end user is concerned with how the complexity of the system affects its usability and the ability of the user to specify and select computational tasks; in other words, to specify the composition request in the proposed framework.

This section analyses the effort involved in developing context aware applications using legacy application frameworks, which embed the contextual dependencies, use the traditional application development methodology, and employ available application design toolkits and context middleware solutions. It then compares it to the effort it takes to develop context aware applications using our proposed methodology, grounded in the context aware service composition framework.

Table 4 summarises the design and development steps involved in each approach. For each phase in the development process, the table shows its development mode, its frequency when adaptation is required and its difficulty. Development modes are categorised as *manual coding*, *semi-automated* — with the assistance of toolkits and scripts, or *fully automated* — with the assistance of frameworks or middleware.

The most critical steps are 3, 4, 5, 7, 16, 17, and 18, which deal with the specification of context behaviour and the systems's ability to handle unpredictability and react to failures. These tasks are *manual or semi-automatic*, have to be done *repetitively*, and are *moderate* to *difficult* in terms of their complexity, as they require specifying and manually encoding context behaviour in the application.

These steps are *automated* and *simplified* using the proposed framework, significantly reducing the amount of manual effort needed for application development. The trade-off is that service composition approach requires the semantic annotation of services (Step 8) and construction of the domain description (Step 9). However, despite the fact that these tasks require (possibly manual) design and development, both stages occur *only once* for each application domain. We believe that this is a limited amount of effort compared to the continued effort of individual programmers to keep up with new contextual parameters.

To build a software solution in a given scenario using our application, the developer needs to perform two key steps. Firstly, she needs to describe the problem definition, which corresponds to representing the scenario concepts and abstract services. If abstract services from other scenarios or previous deployments — e.g. libraries — are available, they can be reused. Secondly, the developer needs to define the user task intentions that will be supported, and the context types that are relevant in the scenario. We are investigating the automation of both of these operations based on inference in our ongoing work.

Unlike the legacy approach, when the new context is introduced into the application, the proposed service composition approach *removes the need for manual reprogramming*. The composition framework automatically discovers new context using context middleware, generates new context

Table 4 Comparison of design process when building context aware applications using a legacy approach and using our proposed service composition approach

Step	Step description	Development mode		Frequency		Difficulty	
		Legacy	Our system	Legacy	Our system	Legacy	Our system
Task specification							
1	Specify context	semi	semi	repeated	repeated	easy	easy
2	Register with context providers	auto	auto	repeated	repeated	easy	easy
3	Specify desired core goal	*manual*	*semi*	*repeated*	*repeated*	*moderate*	*easy*
4	Specify desired context behaviour	*semi*	*auto*	*repeated*	*repeated*	*difficult*	*easy*
5	Select desired task	*semi*	*semi*	*repeated*	*repeated*	*moderate*	*easy*
Application behaviour specification and configuration							
6	Develop core functionality	manual	n/a	one-off	n/a	moderate	n/a
7	Encode context aware behaviour	*manual*	*n/a*	*repeated*	*one-off*	*difficult*	*effortless*
8	Semantic description of services	n/a	semi	n/a	one-off	n/a	difficult
9	Generate domain description	n/a	semi	n/a	one-off	n/a	moderate
10	Generate problem definition	n/a	auto	n/a	repeated	n/a	easy
11	Generate abstract plan	n/a	auto	n/a	repeated	n/a	easy
12	React to plan failures	n/a	auto	n/a	repeated	n/a	easy
13	Generate architecture specific plan	n/a	auto	n/a	repeated	n/a	easy
Application execution							
14	(Platform-specific) deployment	semi	n/a	repeated	n/a	moderate	n/a
15	Service discovery and invocation	n/a	auto	n/a	repeated	n/a	easy
Unpredictability and failure recovery							
16	React to context changes	*semi*	*auto*	*repeated*	*repeated*	*difficult*	*easy*
17	React to task changes	*auto*	*auto*	*repeated*	*repeated*	*moderate*	*easy*
18	React to execution failures	*semi*	*auto*	*repeated*	*repeated*	*moderate*	*easy*

behaviour (Step 7 implemented by [3]) and adapts applications. Furthermore, an important advantage of our approach is its *generality* and independence of specific application scenarios.

6 Related work

This section describes related work in two main research categories: context aware computing and service composition.

6.1 Middleware for context awareness

Building context aware applications from scratch is not practical, as the facility for specifying, acquiring and processing context must be developed each time. As a result researchers are building infrastructures to decrease the development overhead by decoupling of context from application. Such context architectures are commonly called *context middleware*.

Dey [20] analysed a typical development cycle of a context aware application and identifies the following essential features of context middleware for supporting context aware applications: context specification, resource discovery, context acquisition, interpretation, context storage, transparent distributed computing and constant availability. This analysis extends the set of essential architectural features proposed by Dey, to include the support for the following: distributed context repository, security, privacy and quality of information.

Table 5 shows that not all of the identified properties are present in a single architecture. The review shows that there has been an advance in addressing technical challenges in developing context middleware. However, most conventional architectures for context awareness do not address social and legal issues with respect to privacy and security concerns. It is especially evident that support for privacy and Quality of Information is in its early stages. Only Context Weaver integrates a Quality of Information into its model of context. Context Toolkit, Context Fabric and Context Weaver provide limited support for expressing access control policies for context data.

Table 5 Comparison of context middleware

Feature	Context Middleware						
	Schilit's [21]	Stick-e [20]	TEA [22]	Context Toolkit [23]	Context Fabric [24]	iRoom [25]	Context Weaver [26]
Specification	*	*	*	✓	✓	*	✓
Acquisition	✓	✓	✓	✓	✓	✓	✓
Interpretation	×	×	✓	✓	✓	*	✓
Storage	×	×	✓	✓	✓	✓	✓
Resource discovery	*	*	*	✓	✓	×	✓
Transparent distributed communications	*	×	×	✓	✓	✓	✓
Constant availability	✓	×	×	✓	✓	✓	
Distributed context repository	×	×	×	✓	×	*	✓
Security	×	×	×	*	*	×	*
Privacy	×	×	×	*	*	×	✓
Quality of Information model	×	×	×	*	*	×	✓

× = no support, * = partial or proposed support, ✓ = full support

6.2 Planning-based service composition

We conducted a structured comparison of a number of existing planning-based service composition frameworks with our proposed solution, based on a defined *set of features*. This refers to the functionality that planning-based service composition frameworks need to provide, in order to enable their wide applicability to large-scale, realistically complex pervasive computing environments. The set of features we define is not exhaustive, and extends the one proposed by Koehler et al. [27]. The features we have identified as necessary are the following: extended goals, complex actions, dynamic composition, on-the fly recomposition, user interaction, automatic service discovery, monitoring for nondeterminism, implicit task specification, resource constraints, composition and execution failure recovery.

Comparison of planning-based service composition frameworks. Wu *et al.* [28] used the SHOP2 [29] planner for automating Web service composition in a scheduling scenario. Their system does not support on-the-fly recomposition, does not automatically discover services, and is not able to recover from composition failures.

McIlraith et al. [30] used and extended Golog [31], a high level logic programming language built on top of Situation Calculus [32] able to compose services encoded in DAML-S [33]. User requests are expressed as generic ConGolog [34] templates, constructed off-line and then modified based on user preferences and constraints. This work does not deal with on-the-fly recomposition, nor does it facilitate composition failure recovery.

Ponnekatni et al. proposed SWORD [35], a toolkit for Web service composition. SWORD employs a rule-based expert system based on the Rete algorithm [36], which automatically determines if a desired service can be realised as a composition of existing, predefined, services. SWORD exhibits several limitations, as it does not support extended goals, complex actions, on-the-fly recomposition, automatic service discovery, nondeterminism and monitoring, resource constraints, and failure recovery.

Berardi et al. [37] considered a Web service as a tree of all possible interactions with clients and developed the E-Service Composer (ESC). They used Situation Calculus to provide automated composition, supporting direct interaction with the user in the composition process. The main weaknesses of ESC relate to its lack of support for resource constraints and composition failure recovery.

Akkiraju et al. [38] devised a two layered workflow composition architecture. The higher layer focuses on abstract business process flow specification, where processes are described at a high-level using BPEL4WS semantically annotated with DAML-S. The lower layer deals with service discovery, composition, binding, and execution. This system does not facilitate on-the-fly recomposition and composition failure recovery.

Pistore et al. [39] proposed a service composition framework grounded in the concept of planning as model checking, also known as Model-Based Planning (MBP). They developed the ASTRO [40] toolset, supporting automated service composition, monitoring and execution.

The above findings are summarised in Table 6. Common characteristics of all systems include their centralised architecture, their support for dynamism in the composition process, and the presence of mechanisms for handling execution failures — exclusively reactively, by replacing a service instance with another one upon failure.

Table 6 Comparison of features supported by planning-based service composition frameworks

#	Service framework							
	Feature	Wu's	mcIlIarth's	SWORD	ESC	Akkrijau's	ASTRO	Our
–	Composition method	SHOP2	ConGolog	Rete	Situation calculus	State planner	MBP	TLPlan
–	Service markup	OWL-S	OWL-S	XML	WSTL	OWL-S	BPEL4WS	OWL-S
–	Composition model	central	central	central	central	central	central	central
1	Extended goals	*	✓	×	*	✓	✓	*
2	Complex actions	✓	✓	×	✓	✓	✓	*
3	Dynamic composition	*	*	*	✓	*	✓	✓
4	On-the-fly recomposition	×	×	×	✓	×	×	✓
5	User interaction	*	*	*	✓	*	*	*
6	Automatic service discovery	×	*	×	✓	✓	*	✓
7	Monitoring for nondeterminism	*	*	×	*	*	✓	*
8	Implicit task specification	×	×	×	×	×	×	×
9	Resource constraints	*	*	×	×	*	*	✓
10	Composition failure recovery	×	×	×	×	×	×	✓
11	Execution failure recovery reactive	reactive	×	reactive	reactive	reactive	proactive and reactive	

× = no support, * = partial or proposed support, ✓ = full support

However, all service composition frameworks we investigated lacks features that are crucial for applicability to realistically large and complex pervasive computing environments, and which our framework provides integrated support for: *on-the-fly recomposition*, allowing the adaptation of the composite service to deal with the dynamicity of the environment, and *composition failure recovery*, to allow dealing with incorrectly defined services and user request literals or incomplete knowledge of the world — the mechanisms our system provides to facilitate the latter are analysed in detail in our previous work [10]. Furthermore, our system employs *proactive recovery* from execution failures to mitigate the delays inherent in discovering a new suitable service instance.

7 Conclusion and future work

This paper has proposed a framework implementing a new approach for developing context-aware applications in a structured and extensible way, by rapidly composing them from individual services *on demand*, at a user's request. The composition process takes into account the available services and the context, in order to assemble a composite service that meets the user's task intention. A distinguishing feature of our framework is its support for *recomposition of composite services on-the-fly*, when the context, service availability, or user's task intention changes drastically. Additionally, our framework provides a comprehensive failure management solution, by facilitating recovery from failures occurring during both the *composition* and *execution* stages. The *proactive failure recovery* approach we have developed for the latter case has been shown to provide significant performance benefits over the standard, reactive methodology.

Our framework has been shown to be *efficient* and *scalable* through the experimental evaluation of our prototype implementation. The system supports the composition and deployment of realistically complex composite services, consisting of 80 atomic services, in *less than 225 ms*. The system is able to scale gracefully also when the composition request size and number of concurrent requests are increased. Furthermore, the occurrence of execution failures adds only *5 ms* to the total framework operation time, due to the efficient proactive recovery methodology we have implemented. Finally, by means of qualitative analysis, the framework has been shown to *automise* the most difficult, manual and frequently occurring steps in developing context aware applications, such as encoding the large and increasing number of combinations of context types that applications have to adapt to.

In the future, we plan to investigate assembling and executing composite services based on real-time Quality of Service measurements, acquired through interaction with the services. We also plan to work on optimising performance by interleaving the processes of service discovery and execution, allowing each discovered service to be immediately invoked while the subsequent service is being instantiated.

References

1. Weiser M (1991) The computer for the 21st century. Sci Am 265(3):66–75
2. Vukovic M, Kotsovinos E, Robinson P (2007) Application development powered by rapid, on-demand service composition. In:

Proceedings of the 2007 IEEE international conference on service-oriented computing and applications (IEEE SOCA 2007), Newport Beach, California
3. Vuković M, Robinson P (2005) GoalMorph: Partial goal satisfaction for flexible service composition. Int J Web Services Pract 1(1–2):40–56
4. Vuković M, Robinson P (2005) SHOP2 and TLPlan for Proactive Service Composition. In: Proceedings of the UK–Russia workshop on proactive computing. Nizhniy Novgorod, Russia
5. Bacchus F, Kabanza F (1995) Using temporal logic to control search in a forward chaining planner. In: Proceedings of the second international workshop on temporal representation and reasoning (TIME). Melbourne Beach, FL, USA
6. Curbera F, Andrews T, Dholakia H, Goland Y, Klein J, Leymann F, Liu K, Roller D, Smith D, Thatte S, Trickovic I, Weerawarana S (2005) Business Process Execution Language for Web Services, version 1.1. White Paper available at ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf (Last accessed 1st March 2006)
7. IBM (2004) The IBM Business Process Execution Language for Web Services Java^{TM} Run Time (BPWS4J). http://www.alphaworks.ibm.com/tech/bpws4j (Last accessed 1st March 2006)
8. jUDDI v.0.94rc (2003) Java Implementation of the Universal Description Discovery, and Integration (UDDI) Specification for Web Services. Apache Web Services Project. http://ws.apache.org/juddi/ (Last accessed 1st March 2006).
9. Haigh KZ, Veloso M (1996) Interleaving Planning and Robot Execution for Asynchronous User Requests. In: Planning with Incomplete Information for Robot Problems: Papers from the 1996 American Association for Artificial Intelligence (AAAI) Spring Symposium, Stanford University in Palo Alto, California, USA, AAAI Press, Menlo Park, California 35–44
10. Vukovic M (2006) Context Aware Service Composition. PhD thesis, University of Cambridge
11. Christensen E, Curbera F, Meredith G, Weerawarana S (2001) Web Services Description Language (WSDL) 1.1. Specification available at http://www.w3.org/TR/wsdl (Last accessed 1st March 2006)
12. Apache Tomcat v5.5 Application Server (2006) Software available at http://tomcat.apache.org/ (Last accessed 1st March 2006)
13. Fox A, Gribble SD, Chawathe Y, Brewer EA, Gauthier P (1997) Cluster-based scalable network services. In: Symposium on operating systems principles 78–91
14. The United Nations Standard Products and Services Code UNSPSC. Website available at http://www.unspsc.org/ (1998) (Last accessed 1st March 2006)
15. Gu X, Nahrstedt K, Yu B (2004) SpiderNet: an Integrated Peer-to-Peer Service Composition Framework. In: Proceedings of the IEEE international symposium on high performance distributed computing (HPDC), Honolulu, Hawaii, USA 110–119
16. JConfig (2002) Software available from http://tolstoy.com/samizdat/jconfig.html (Last accessed 19th July 2006)
17. Web Services Description Language for Java (2005) Software available at http://sourceforge.net/projects/wsdl4j (Last accessed 1st March 2006)
18. Ranganathan A, Campbell RH (2003) What is the complexity of a distributed System? Technical Report UIUCDCS-R-2005-2568, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA
19. McCann JA, Huebscher MC (2004) Evaluation issues in autonomic computing. In: Jin H, Pan Y, Xiao N. (eds) Proceedings of the third international conference on grid and cooperative computing workshops (GCC) of Lecture Notes in Computer Science, vol 3252. Springer, Wuhan 597–608
20. Dey AK (2000) Providing architectural support for building context-aware applications. PhD thesis,. Georgia Institute of Technology, Atlanta, GA, USA
21. Hong JI, Landay JA (2001) An infrastructure approach to context-aware computing. Human–Comput Interact J 16:287–303
22. Winograd T (2001) Architectures for Context. Human-Comput Interact J 16:401–419
23. Schilit BN (1995) System architecture for context-aware mobile computing. PhD thesis. Columbia University, New York, USA
24. Pascoe J (1997) The Stick-e Note Architecture: extending the interface beyond the user. In: Moore J, Edmonds E, Puerta A (eds) Proceedings of the international conference on intelligent user interfaces, Orlando, FL, USA. ACM, 261–264
25. Schmidt A, Aidoo KA, Takaluoma A, Tuomela U, Laerhoven KV, de Velde WV (1999) Advanced interaction in context. In: Proceedings of the first international symposium on handheld and ubiquitous computing (HUC). Springer, Karlsruhe, pp 89–101
26. Lei H, Sow DM, Davis II JS, Banavar G, Ebling MR (2002) The design applications of a context service. ACM SIGMOBILE Mobile Comput Commun Rev 6(4):45–55
27. Koehler J, Srivastava B (2003) Web service composition: current solutions and open problems. In: The Proceedings of the international conference on automated planning and scheduling (ICAPS). Workshop on planning for Web Services, Trento, Italy pp 28–35
28. Wu D, Sirin E, Hendler J, Nau D, Parsia B (2003) Automatic Web Services composition using SHOP2. In: Proceedings of the 13th international conference on automated planning and scheduling. Workshop on planning for Web Services, Trento, Italy
29. Nau DS, Muñoz-Avila H, Cao Y, Lotem A, Mitchell S (2001) Total-order planning with partially ordered subtasks. In: Nebel B (eds) Proceedings of the seventeenth international joint conference on artificial intelligence (IJCAI), Seattle, Washington, USA, pp 425–430
30. McIlraith S, Son TC (2002) Adapting golog for composition of semantic Web Services. In: Proceedings of the eighth international conference on knowledge representation and reasoning (KR2002), Toulouse, France
31. Levesque HJ, Reiter R, Lesperance Y, Lin F, Scherl RB (1997) GOLOG: a logic programming language for dynamic domains. J Logic Programm 31(1-3):59–83
32. McCarthy J, Hayes PJ (1969) Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer B, Michie D (eds) Machine intelligence 4. Edinburgh University Press, Edinburgh, pp 463–502
33. Ankolenkar A, Burstein M, Hobbs JR, Lassila O, Martin DL, McDermott D, McIlraith SA, Narayanan S, Paolucci M, Payne TR, Sycara K (2002) DAML-S: Web service description for the semantic Web. In: Proceedings of the first international semantic Web conference (ISWC), Sardinia, Italy
34. Giacomo GD, Lesperance Y, Levesque HJ (2000) Congolog, a concurrent programming language based on the situation calculus. Artif Intell 121(1-2):109–169
35. Ponnekanti SR, Fox A (2002) SWORD: A developer toolkit for Web service composition. In: Proceedings of the 11th World Wide Web conference (Web Engineering Track), Honolulu, Hawaii, USA
36. Forgy C (1982) Rete: A fast algorithm for the many patterns/many objects match problem. Artif Intell 19(1):17–37
37. Berardi D (2005) Automatic service composition. Models, techniques, tools. PhD thesis, University of Rome "La Sapienza", Rome, Italy
38. Akkiraju R, Verma K, Goodwin R, Doshi P, Lee J (2004) Executing abstract Web process flows. In: Proceedings of the international conference on automated planning and scheduling (ICAPS). Workshop on planning and scheduling for Web and grid services, Whistler, BC, Canada
39. Pistore M, Barbon F, Bertoli P, Shaparau D, Traverso P (2004) Planning and monitoring Web service composition. In: Proceedings of the artificial intelligence: methodology, systems, and

applications, 11th international conference (AIMSA), Varna, Bulgaria pp 106–115
40. Trainotti M, Pistore M, Calabrese G, Zacco G, Lucchese G, Barbon F, Bertoli P, Traverso P (2005) ASTRO: supporting composition and execution of Web services. In: Proceedings of the international conference on automated and planning sheduling (ICAPS). Demo, Monterey, CA, USA