

Application development powered by rapid, on-demand service composition

Maja Vuković
IBM T.J.Watson Research

Evangelos Kotsovinos
Deutsche Telekom Laboratories

Peter Robinson
University of Cambridge

Abstract—Context aware application design models are often based on embedding contextual dependencies — in the form of if-then rules — to specify how applications should react to context changes. In this paper we argue that such approaches are impractical and increase the complexity of context aware applications, due to the large variety of possibly even unanticipated context types and their values. We present the design and implementation of a framework for building context aware applications on-demand, as dynamically composed sequences of calls to services. Our system employs goal-oriented inferencing for assembling composite services, monitors their execution, adapts applications to deal with contextual changes, and enables failure recovery mechanisms that minimise application disruption. By means of experimental evaluation in a realistic infotainment application, we demonstrate that the framework provides an effective, efficient, and scalable solution.

I. INTRODUCTION

Context aware applications, which use information about their context to respond and adapt to changes in the computing environment, operate in an increasingly wider variety of settings, such as cars and wearable devices. However, advances in application models to support the development of context aware systems have not kept up. Applications are often built in a scenario-specific manner, statically encoding the anticipated context types and desired application behaviour using if-then rules.

This presents important disadvantages. Firstly, if-then rule-based approaches are only feasible for the set of context types anticipated at the time of application design. 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.

The rest of the paper is structured as follows. Section II examines an example scenario demonstrating how we envisage our system to operate from a user's point of view. The architecture of our framework is analysed in Section III. Section IV presents the results of our experimental evaluation. Section V positions our work in its research context, and discusses shortcomings of previous service composition frameworks. Finally, Section VI presents our conclusions and outlines areas of future work.

II. USAGE SCENARIO

The following scenario demonstrates how the proposed framework can simplify the development of context aware applications. A user, called Miles, subscribes to an infotainment portal provided by his mobile network operator. This portal offers users 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 text to speech. User requests to the portal 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. by a context middleware solution.

Use case 1: Miles uses his SmartPhone while in Cambridge, UK, to place a request to the infotainment portal for

finding a restaurant of his liking, and providing directions to it. The portal uses our system to assemble a composite service to deal with Miles’ request from the atomic services *Restaurant Finder* and *Directions Finder*.

Use case 2: Later on 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, given the different context in which the requests are submitted. In the first case, Miles is using a *SmartPhone* while *walking* in *Cambridge*. By contrast, in the second case, Miles is *driving* through *Zurich*, and using *IVIS*. Our system allows Miles to submit both requests in the same way, and automatically handles application adaptation.

III. SYSTEM ARCHITECTURE

This section describes the architecture of the proposed framework using the scenario described in Section II. It presents the functionality that each component delivers, and the interactions between them to facilitate fault-tolerant, context aware service composition.

A. Overview

Figure 1 shows the overall operation of our system. 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 context to create a composition request for the *abstract service composition* layer. This in turn assembles a composite service,

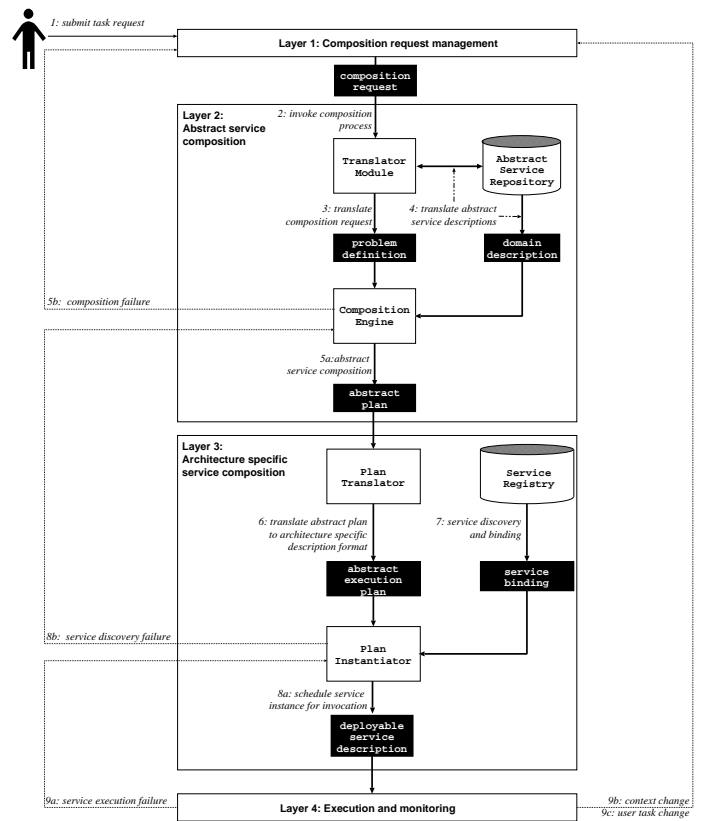


Fig. 1. System architecture overview

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.

B. Operation

Composition request management: The composition process starts with the receipt of a user request (Step 1 in Figure 1). Based on that, this step creates a *composition request*; this includes a number of *literals* specifying the users *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 2) 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 [1] in our prototype — (Steps 3 and 4). Then the problem definition is passed to the composition engine for building an *abstract plan* (Step 5a). 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 [2], a system which transforms failed composition requests into alternative ones that can be solved (Step 5b).

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 [3] in our prototype — (Step 6). The *Plan Instantiator* — implemented using the BPWS4J engine ¹, uses the *Service Registry* — implemented using jUDDI v0.94 ² 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 8a).

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 9a). Should an unanticipated change in context occur (Step 9b), or should the user change the task specification (Step 9c), control is passed to the composition request management layer, where a new composition request is generated and recomposition triggered.

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 runs observes context changes and service execution using the monitoring model proposed by Haigh et al. [4]. 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 [5].

IV. EVALUATION

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 section presents evaluation results demonstrating the framework's *performance* and *scalability*.

A. Experimental setup

Framework deployment: Figure 2 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 1700MHz 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 ³. The two machines were in same LAN connected over a 100Mbps Ethernet. This setup is illustrative of an arrangement where the different parts of the framework, such as the Composition Engine and the Service Registry are distributed on nodes hosted on different machines connected to Internet.

Context-rich environment: All experiments were performed in a context-rich infotainment environment, corresponding to the one described in Section II. 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.

¹<http://www.alphaworks.ibm.com/tech/bpws4j>

²<http://ws.apache.org/juddi/>

³<http://tomcat.apache.org/>

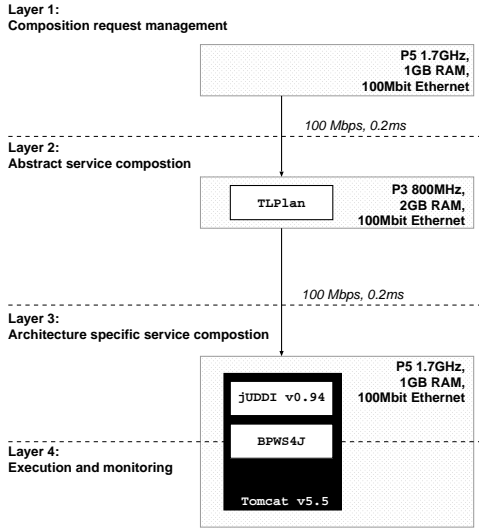


Fig. 2. Experimental setup

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

TABLE I
SAMPLE SERVICE CATEGORISATION

- 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) ⁴ categorisation. These categorisations were stored in the Service Registry. Table I 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.

⁴<http://www.unspsc.org/>

B. Performance

This experiment evaluated the performance of our system, both with and without the presence of service execution failures (e.g. services becoming 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.* [6]. 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 Section III, 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 ⁵ 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 millisecond.

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 3 shows the total framework operation time for each one of the above test cases. On average the

⁵<http://tolstoy.com/samizdat/jconfig.html>

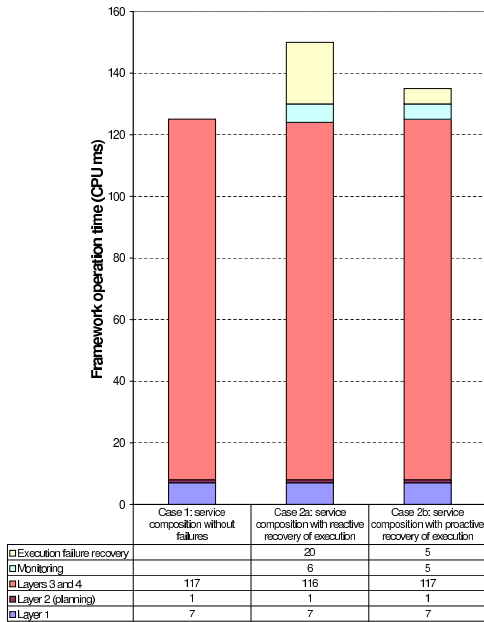


Fig. 3. Framework operation timeline for Cases 1, 2a, and 2b

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 20ms on average, while using proactive approach in Case 2b reduces the failure recovery time by 15ms — down 75% from the 20ms it took in Case 2a. This underlines the significant performance — and thus user experience — benefit that using our proactive recovery approach provides, increasing 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 ⁶ 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.

C. 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*

⁶<http://sourceforge.net/projects/wsd14j>

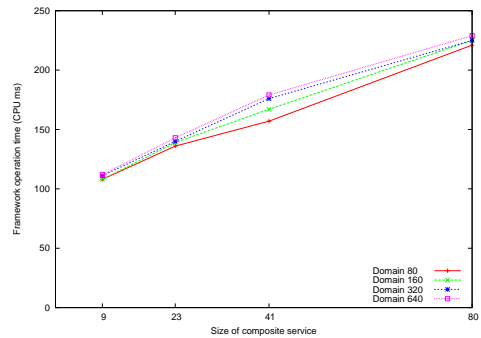


Fig. 4. Scalability when increasing problem definition size

requests submitted. The experimental configuration described in Section IV-A 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 Section IV-A, 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 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 4 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 II 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 5 shows the average composite framework operation time and provides a breakdown of time taken by each layer of

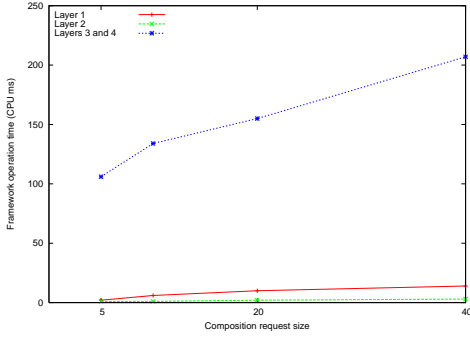


Fig. 5. Scalability when increasing the composition request size, for domain size 160.

Test cases				
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

TABLE II
COMPOSITION REQUEST SIZE AND COMPOSITE SERVICE SIZE

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 Section IV-A. The BPWS4J engine v2.1, which was used in our prototype implementation, does not support programmatical deployment of more than 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.

Figure 6 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.

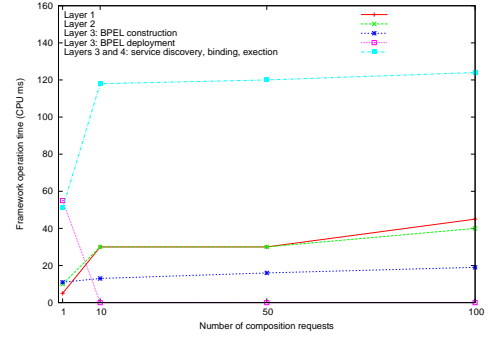


Fig. 6. Scalability when increasing number of composition requests

Step / Requests	1	10	50	100
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 III
SCALABILITY WHEN INCREASING THE COMPOSITION REQUEST SIZE:
COMPOSITION TIME DISTRIBUTION (CPU MS)

Table III 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 [7]. Furthermore, the relatively small increase in the time taken for layers 3 and 4, shown in Table III, 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.

Finally, it is important to show that the design of the service composition framework does indeed make context aware applications *easier to build and maintain*. In [5] we conduct a qualitative evaluation of the system, analysing how it helps reducing the *development effort* required for building context aware applications.

V. RELATED WORK

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 (design and runtime) features we define is not exhaustive, and extends the one proposed by Koehler *et al.* [8]. An important challenge in providing a context aware service composition facility is dealing with failures that may occur during *composition*, for instance as a result of context changes or missing service descriptions; as well as during *execution* of a composite service when atomic services may become unavailable, for example because of a network disconnection.

Wu *et al.* [9] used the SHOP2 [10] 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.* [11] used and extended Golog [12] to compose services encoded in DAML-S [13]. User requests are expressed as generic ConGolog [14] 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 [15], a toolkit for Web service composition. It employs a rule-based expert system based on the Rete algorithm [16], 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.* [17] 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, however lack of support for resource constraints and composition failure recovery.

Akkiraju *et al.*, [18] devised a two layered workflow composition architecture. The higher layer focuses on ab-

stract 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.* [19] 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 [20] toolset, supporting automated service composition, monitoring and execution. ASTRO lacks support for on-the-fly recomposition and recovery from composition failures.

The above findings are summarised in Table IV. 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.

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 [5]. Furthermore, our system employs *proactive recovery* from execution failures to mitigate the delays inherent in discovering a new suitable service instance.

VI. 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 based on user goals, available services, and context. Contrary to related systems, our framework supports recomposition of composite services on-the-fly, and provides a comprehensive failure management solution, facilitating proactive recovery from failures occurring during both the composition and execution stages. Our framework has been shown to be *efficient* and *scalable* through experimental evaluation. It supports efficient composition and deployment of realistically complex composite services, scales gracefully,

Service framework								
#	Feature	Wu's	McIlraith's	SWORD	ESC	Akkiraju'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

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

TABLE IV
COMPARISON OF FEATURES SUPPORTED BY PLANNING-BASED SERVICE COMPOSITION FRAMEWORKS.

and enables the rapid recovery from execution failures.

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] F. Bacchus and F. Kabanza, "Using Temporal Logic to Control Search in a Forward Chaining Planner," in *Proc. of TIME 1995*.
- [2] M. Vuković and P. Robinson, "GoalMorph: Partial Goal Satisfaction for Flexible Service Composition," *International Journal of Web Services Practices*, vol. 1, no. 1-2, pp. 40-56, December 2005.
- [3] F. Curbera, T. Andrews, H. Dholakia, Y. Golland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, "Business Process Execution Language For Web Services, v1.1." White Paper, 2005.
- [4] K. Z. Haigh and M. Veloso, "Interleaving Planning and Robot Execution for Asynchronous User Requests," in *Proc. AAAI Spring Symp. 1996*.
- [5] M. Vukovic, "Context aware service composition," PhD Thesis, University of Cambridge, 2006.
- [6] X. Gu, K. Nahrstedt, and B. Yu, "SpiderNet: an Integrated Peer-to-Peer Service Composition Framework," in *Proc. of HPDC 2004*.
- [7] "Web Services Description Language for Java," Software available at <http://sourceforge.net/projects/wsdl4j>, 2005.
- [8] J. Koehler and B. Srivastava, "Web Service Composition: Current Solutions and Open Problems," in *Proc. of ICAPS 2003*.
- [9] D. Wu, E. Sirin, J. Hendler, D. Nau, and B. Parsia, "Automatic Web Services Composition Using SHOP2," in *Proc. of the 13th Intl Conf on Automated Planning and Scheduling*, 2003.
- [10] D. S. Nau, H. Muñoz-Avila, Y. Cao, A. Lotem, and S. Mitchell, "Total-Order Planning with Partially Ordered Subtasks," in *Proc. of IJCAI 2001*.
- [11] S. McIlraith and T. C. Son, "Adapting Golog for Composition of Semantic Web Services," in *Proc of KR2002*.
- [12] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl, "GOLOG: A Logic Programming Language for Dynamic Domains," *Journal of Logic Programming*, vol. 31, no. 1-3, pp. 59-83, 1997.
- [13] A. Ankolenkar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. Sycara, "DAML-S: Web Service Description for the Semantic Web," in *Proc. of ISWC 2002*.
- [14] G. D. Giacomo, Y. Lesperance, and H. J. Levesque, "Congolog, a Concurrent Programming Language Based on the Situation Calculus," *Artificial Intelligence*, vol. 121, no. 1-2, pp. 109-169, 2000.
- [15] S. R. Ponnekanti and A. Fox, "SWORD: A Developer Toolkit for Web Service Composition," in *Proc. of WWW11*, 2002.
- [16] C. Forgy, "Rete: A fast algorithm for the many patterns/many objects match problem," *Artificial Intelligence*, vol. 19, no. 1, pp. 17-37, 1982.
- [17] D. Berardi, "Automatic service composition. models, techniques, tools." Ph.D. dissertation, University of Rome "La Sapienza", 2005.
- [18] R. Akkiraju, K. Verma, R. Goodwin, P. Doshi, and J. Lee, "Executing Abstract Web Process Flows," in *Proc. of ICAPS 2004*.
- [19] M. Pistore, F. Barbon, P. Bertoli, D. Shapara, and P. Traverso, "Planning and Monitoring Web Service Composition," in *Proc of AIMSA 2004*.
- [20] M. Trainotti, M. Pistore, G. Calabrese, G. Zacco, G. Lucchese, F. Barbon, P. Bertoli, and P. Traverso, "ASTRO: Supporting Composition and Execution of Web Services," in *Proc. of ICAPS 2005*.