Technical Report

UCAM-CL-TR-700
ISSN 1476-2986

Number 700

Context aware service composition

Maja Vuković

October 2007

This technical report is based on a dissertation submitted April 2006 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Newnham College.

ISSN 1476-2986

Summary

Context aware applications respond and adapt to changes in the computing environment. For example, they may react when the location of the user or the capabilities of the device used change. Despite the increasing importance and popularity of such applications, advances in application models to support their development have not kept up. Legacy application design models, which embed contextual dependencies in the form of *if-then* rules specifying how applications should react to context changes, are still widely used. Such models are impractical to accommodate the large variety of possibly even unanticipated context types and their values.

This dissertation proposes a new application model for building context aware applications, considering them as dynamically composed sequences of calls to *services*, software components that perform well-defined computational operations and export open interfaces through which they can be invoked. This work employs goal-oriented inferencing from planning technologies for selecting the services and assembling the sequence of their execution, allowing different compositions to result from different context parameters such as resources available, time constraints, and user location. Contextual changes during the execution of the services may trigger further re-composition causing the application to evolve dynamically.

An important challenge in providing a context aware service composition facility is dealing with failures that may occur, for instance as a result of context changes or missing service descriptions. To handle composition failures, this dissertation introduces GoalMorph, a system which transforms failed composition requests into alternative ones that can be solved.

This dissertation describes the design and implementation of the proposed framework for context aware service composition. Experimental evaluation of a realistic infotainment application demonstrates that the framework provides an efficient and scalable solution. Furthermore, it shows that GoalMorph transforms goals successfully, increasing the utility of achieved goals without imposing a prohibitive composition time overhead.

By developing the proposed framework for fault-tolerant, context aware service composition this work ultimately lowers the barrier for building extensible applications that automatically adapt to the user's context. This represents a step towards a new paradigm for developing adaptive software to accommodate the increasing dynamicity of computing environments.

Acknowledgements

This dissertation would not have been possible without the support of many great people, to whom I express my gratitude.

First of all, I would like to thank Peter Robinson, my advisor, for giving me the opportunity and freedom to pursue a PhD as a member of the Rainbow Group, Computer Laboratory, University of Cambridge. His tremendous support, guidance and understanding enabled me to complete this project.

I greatly appreciate the discussions with Jean Bacon, Carl Binding, Yigal Hoefner and Jana Koehler. I am grateful to Andreas Reuter, Anthony Jameson and Oliver Fritz for encouraging me to pursue a PhD. For their help in proofreading, support and discussions I would like to thank specifically Rana el Kaliouby, Tihana Kraš, Jennifer Rode, Tal Sobol-Shikler, Mark Stringer, Vikas Taliwal and Phil Tuddenham.

I would like to thank Giorgos Cheliotis and Christian Facciaruso for our initial discussion on symmetric matchmaking methods. My thanks go to Anthony Bussani for his kind assistance with state of the art. I would also like to extend my thanks to Gero Dittman, Diana Gulli, Chiara Marchiori and Roger Zimmerman for helping make my stay in Zurich pleasant and productive.

I was especially fortunate to have been generously supported by IBM Zurich Research Laboratory. This has also provided me with an opportunity to work with leading researchers at IBM Zurich Laboratory and IBM T.J. Watson Research Center. I am grateful to Doug Dykeman and Stefan Hild, without whose effort and flexibility this would not have been possible.

I would also like to thank Cambridge Commonwealth Trust, Newnham College, and the Royal Academy of Engineering for supporting this research.

My time at Cambridge would not have been the same without Aleksandra Gruevska and Nives Mikelić. I would also like to thank Gordana Apić for offering practical advice during critical stages of my research.

I would especially like to thank Evangelos Kotsovinos for our extensive discussions on adaptive composition, constructive feedback on dissertation drafts, patience and unending support during the most challenging times of this research project.

Last but not least, I would like to thank my parents, Vladimir and Dragana, whose hard work and devotion made all this possible. Words cannot express my gratitude to my sister Tamara and brother Ivan for their love and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

ADL	Action Description Language
AI	Artificial Intelligence
API	Application Programming Interface
BPEL4WS	Business Process Execution Language for Web Services
BPWS4J	Business Process Execution Language for Web Services Java Run Time
CORBA	Common Object Request Broker Architecture
DAML	DARPA Agent Markup Language
DAML-S	DAML for Services
DARPA	Defense Advanced Research Projects Agency
DCOM	Distributed Component Object Model
HTN	Hierarchical Task Network
HTTP	HyperText Transfer Protocol
MBP	Model Based Planning
MDP	Markov Decision Process
OWL	Web Ontology Language
OWL-S	OWL for Services
PDDL	Planning Domain Description Language
RDF	Resource Description Framework
RMI	Remote Method Invocation
SHOP2	Simple Hierarchical Ordered Planner 2
SOAP	Simple Object Access Protocol
STRIPS	Stanford Research Institute Problem Solver
TCP/IP	Transmission Control Protocol/Internet Protocol
UDDI	Universal Description, Discovery and Integration
UNSPSC	United Nations Standard Products and Services Code
URL	Uniform Resource Locator
WSDL	Web Service Description Language
WSMF	Web Service Modeling Framework
WSMO	Web Service Modeling Ontology
XML	eXtensible Markup Language

Terminology

Abstract execution plan: a composite service in the representation format of the run-time execution technology.

Abstract plan: a high-level schema that describes the control flow between atomic services, which form a composite service.

Abstract service: a high-level description of the capabilities and categorisation of an atomic service.

Abstract service composition: the process of constructing a composite service from abstract services.

Abstract service repository: the facility that stores and manages descriptions of abstract services.

Architecture specific service composition: the process of constructing an executable composite service, by means of instantiation of abstract services in the abstract execution plan.

(Atomic) service: a specific instance of a software system that carries out a computational operation on behalf of a user.

AtomicProcess: an OWL-S description of an atomic service.

Base core goal condition: the absolute minimal core goal condition that needs to be satisfied to achieve a viable solution for a composition request.

Composition request: a construct that specifies a user's computational task. It consists of core and context goal conditions.

CompositeProcess: an OWL-S description of a composite service.

Composite service: a complex service formed by combining specific, atomic, services, to achieve a desired computational task.

Context: the entire collection of entities and their properties that can enter into a meaningful relationship with users during their interaction with applications (including the user and the system themselves), and affect their behaviour.

Context goal condition: a goal condition that arises in a specific context.

Core goal condition: a goal condition that arises from a user's task intention.

Dependent context goal condition: a context goal condition that is contingent on a core goal condition. It can often take form of an attribute of the core goal condition.

Deployable service description: a data structure that defines the physical location of the service, its input and output parameters, preconditions and post-conditions, as well as failure recovery methods.

Domain description: a formalised list of available actions in the domain, representing the causal laws and relationships between actions.

Independent context goal condition: a context goal condition that is not necessarily contingent on a core goal condition.

Goal (condition): a specific requirement that must be met.

Goal transformation: an operation that changes the form of a goal to generate an alternative goal. For example, a goal can be replaced by more specific or generic values.

GoalMorph: a composition failure management system, which uses context aware goal transformations to modify failed composition requests into ones that can be solved.

Monitoring procedure: a construct that defines how the framework should react to unanticipated events resulting from changes in the service operation or context.

Planning: the process of constructing action sequences that can be executed in a given initial state to achieve a desired goal.

Planning action: an operator that transforms the world states.

Precondition: a requirement for carrying out a service execution.

Problem definition: a description of the state of the world, called initial state, and the desired world state, called goal state.

Postcondition: an outcome property that must always hold following the execution of service.

Service binding: the message format and protocol details of service operations.

Service composition: the process of constructing a composite service from atomic services in order to achieve a specific task.

Service invocation: the process of putting a service instance into operation.

Service registry: a repository that maintains records of service descriptions. It exports interfaces for service discovery and publishing.

SimpleProcess: an OWL-S description of an abstract service, providing a specialised view of some AbstractProcess or a simplified representation of some CompositeProcess.

of a composite service.

Task: a computational operation to be performed. Tasks are often complex operations, such as making travel plans and preparing research reports.

This dissertation uses the following typefaces:

- Names of system components names are shown in title case, Courier New, such as `Context Mesh`, and are written in two separate words, except for the `GoalMorph` system, which is written as one word.
- Names of internal data structure names are shown in lower case, plain Courier New font, such as `abstract plan`.

Chapter 1

Introduction

Computing devices are becoming faster, smaller, more widespread and universally connected as the wireless networking revolution continues. At the same time advances in sensing technologies and the development of knowledge extraction and management capabilities are providing context information, such as location, physiological state, and motion. Such data about the user's setting is collectively termed *context*. This suggests a vision of systems that are intimately embedded in physical and social contexts, promising truly ubiquitous computing [Wei91].

As a result, computing 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. They are, in short, increasingly *context aware*.

However, advances in application models to support the development of context aware systems have not kept up. Researchers have been building and deploying context aware applications in a scenario-specific manner. They have often tailored them to a specific problem domain, by encoding the anticipated context and the desired application behaviour.

This dissertation envisages a novel application architecture that is *adaptive* to the changing needs of users and the dynamic computer and networking environments that users may encounter. Adaptive applications are capable of operating on a variety of computing devices that a user may own, such as a personal computer and a Personal Digital Assistant (PDA). They also adjust the set of features and adapt functions to the particular needs of the user in different environments. For example, e-mail software opts to read out e-mail while the user is driving a car.

This dissertation proposes a new application model for context aware applications, in which applications are dynamically assembled from services based on context. Moreover, the configuration of services within an application may also change in order to respond to context changes. This dissertation describes the design, prototype implementation, and evaluation of an extensible framework for context aware service composition. It investigates a novel mechanism for handling composition failures, which tackles the problem of composition requests that cannot be immediately satisfied. Furthermore, it presents a format for context aware composition requests and a method for constructing them in an automated way. Finally, the dissertation demonstrates that the proposed framework is an effective and scalable solution by means of experimental evaluation.

1.1 Motivation

Research in the development of context aware applications has gained significant attention since the early 1990s. A representative example is the Active Badge system at Olivetti Research Laboratory [WHFG92]. Active Badges transmit a signal providing information about their location to a centralised location service, through a network of sensors. An early application used the Active Badge system to route phone-calls to the nearest telephone to the user. Bennet *et al.* [BRH94] extended this application to transfer a user's computing session to the nearest workstation.

At the same time Schilit *et al.* [SAW94], researchers at Xerox PARC, developed a suite of context aware applications for PARCTAB [WSA$^+$95], a device that combines the properties of the Active Badge System and PDA. In their prior work, Schilit *et al.* [SD91] described how an operating system can use the memory of nearby idle computers for backing store, rather than swapping to a local or remote disk. One of the first PARCTAB [SW95] applications displays team members and computing devices, such as printers, according to their proximity. Another PARCTAB application is the Location Browser, an application for viewing a "location-based filesystem", in which directories are named after locations and contain files and programs. When a user moves within the building, the browser updates the displayed directory to match the location of the user.

Lamming *et al.* [LF94] developed Forget-Me-Not for PARCTAB, one of the first examples of contextual reminder applications. Forget-Me-Not is a portable memory aid which automatically collects data. It allows users to search and display the collected information based on the context, and traverse implicit links between past events. CyberMinder [DA00], comMotion [MS00] and Memo-Clip [Bei00] are further examples of reminder applications.

Context aware applications are present in many different domains, ranging from tour guides to games. Tour guide applications target, for example, museum, exhibition and shop visitors, city tourists and trade show participants. The standard functionality provided by context aware guides includes context-based information retrieval, navigation in unfamiliar environments, point-of-interest lookup, and dynamic tour generation. Examples of such applications include Personal Shopping Assistant [ACK94], CyberGuide [LKAA96], SmartSight Tourist Assistant [YYDW99], Context Sensitive Tourist GUIDE [CMD99], Context Sensitive Nomadic Exhibition Guide [OS00], and the HP Exploratorium Project [FFK+02]. These systems differ in the extent of contextual information they use as well as the services they provide to the user.

Similarly, fieldwork applications [RPM98, NSBW00] use context to assist mobile workers in field observation and data-collection activities. Another group of applications enables sharing a user's context, in order to determine her availability for communication. as well as the applicability of a specific communication medium, such as telephone, chat, and video conference. Some examples include AudioAura [MBW+98], Situational Awareness [SAT+99], In/Out Board [Dey00], Context-Call [STM00] and Awarenex [TYB+01].

More recent examples of context aware applications are context aware information appliances, such as MediaCup and Chameleon Tables. Gellersen *et al.* [GBK99] developed MediaCup, a coffee mug empowered with computing capabilities, which senses how the cup is used. For example, if someone drinks out of the cup this information is sent to a coffee machine, which may initiate a brewing process. Selker *et al.* [SAB02] devised Chameleon Tables, tables which are aware of their height and neighbouring tables. They respond to context changes by adjusting their height and the content of information displays, which are docked on the top of each table.

A number of projects explore applications of context awareness in smart environments. Ponnekanti *et al.* [PLF+01] and Roman *et al.* [RHC+02] developed adaptive systems that react to changes of resources in order to manage tasks in an intelligent office. Our prior work employed context awareness to facilitate adaptive coordination of a smart home environment [KV05].

Context information is also being exploited in games. Headon *et al.* [HC02] demonstrate how computer games can be controlled through players interacting with the physical environment. Contextual information, such as users' real movements on an ActiveFloor mat [AJLS97] replace cursor key emulation to control a game. Furthermore, the Citywide Performance [BBC+01] is an example of a mixed reality performance that takes place across a city. Users of the system move around the city and experience events that are taking place in a parallel virtual city, a 3D model that is connected to and overlaid on the physical city. Applications such as a virtual history guide and games were also developed.

In terms of how these applications are developed, they mostly employ a traditional, monolithic application model. It embeds contextual dependencies as *if-then rules*, which describe how context aware systems should react to context changes. These rules are encoded by the software engineer. Using this approach whenever the new context types and values are introduced in the system, new rules describing context behaviour need to be created by the software engineer. This makes the applications static and inflexible. Furthermore this may often limit applications to run on a specific device, while offering only predetermined functions to the user. As a result this model is not suited to accommodate pervasive computing environments, which are characterised by richness of context, by the mobility of users and devices, and by the appearance and disappearance of resources over time.

The development of context aware applications is a complex task because of the need to accommodate for a vast variety of context types and their values, including the ones that cannot be anticipated at the time when the system is designed. An example scenario is when a new device comes to the market with different system capabilities from existing ones. The commonly followed approach of hard-coding the mappings between all possible combinations of context values and the corresponding application behaviour is impractical. Furthermore, this makes context aware systems difficult to later extend when new values of existing context attributes and new context types arise. It is at the best extremely demanding to foresee all context an application may encounter during its lifetime.

Weiser [Wei91] envisages pervasive applications as means by which a user performs tasks, rather than a collection of computational features. To achieve this vision, Banavar *et al.* [BBG^{+}00] structure applications in terms of tasks and their sub-tasks, which is a service composition problem.

To address the problem of increasing complexity in facilitating context awareness, I propose the approach of building context aware applications as dynamically composed sequences of calls to Web services, using Artificial Intelligence (AI) planning technology [VR04b]. Different service compositions of such sequences result from different contexts such as available resources, time constraints, user location, and user profile. Further recomposition of the service during its execution may be triggered by changes in the context [VR04a].

Planning systems generate sequences of actions, called *plans*, which can transform an initial state of the world to a desired goal state. The domain knowledge that planners use to devise a plan includes information about the available actions, the conditions under which an action applies, termed preconditions, and the expected outcomes of applying that action, called postconditions or effects [RN95]. By explicitly declaring Web services as processes in terms of their inputs, outputs, preconditions and effects, planning technology can then be applied to solve the service composition problem. However, there are challenges to be overcome for this vision to be realised and these are discussed in the next section.

Planning systems rely on a domain description, which partially encodes possible contextual constraints, however, using this approach, it is sufficient to provide the set of potential values of a certain context type, rather than defining all permutations of all potential values. An example is, when a new device is introduced to the market, which may lead to a combination of existing device capability values, such as screen size, network connections, etc. The proposed approach does not require encoding of all the combinations in the system, but rather the possible values, and deals with combinations automatically.

1.2 Research challenges

The starting point for this dissertation is the need for a new method for building context aware applications. Context aware service composition incorporates research from several different areas, such as context aware computing, service composition, and AI planning. This section provides an overview of these fields and a list of challenges that need to be overcome in each of them to enable context aware service composition.

Context awareness. Building context aware applications is a complex task. Schilit [SAW94] highlighted the challenge of balancing the requirement for timely execution with the need for predictable behaviour when developing context aware applications. Furthermore, Schilit also outlined the following specific problems to be addressed: the expressiveness of the predicate language and the accuracy and timeliness of the underlying context information.

Dey [Dey00] identified three main requirements. Firstly, there is a need for a suitable context model, which describes the relationships between different types and facilitates inference and abstraction of context. Secondly, a Quality of Information model is necessary to allow reasoning about the quality parameters of each context type and value, such as accuracy of location information. Finally, to enable easy development, a suitable infrastructure for context acquisition and management is required, which separates the acquisition from the use of context.

Service composition. Syzperski [Szy00] defined service composition as the process of constructing a complex service from atomic ones to achieve a specific task. The process of service composition inherently requires the specification of composition requests, a formal specification of static and behavioural properties of the service components, a matchmaking algorithm, and a modelling language expressing the logic of a composite service.

Web service technology is a popular way of developing distributed applications. There are two main directions in providing models for composite Web services, in order to formalise the specification of Web services, their composition and execution: industry standards and research ontology-based approaches.

Industry solutions are eXtensible Markup Language (XML) based standards. Examples include Web Service Choreography Interfaces (WSCI) [Wsc02] and Business Process Execution Language for Web Services (BPEL4WS) [CAD$^+$05]. van der Aalst *et al.* [vdADtH03] observed that these languages provide support for *"communication oriented process definition"*, but they lack well-defined semantics.

The Semantic Web community focuses on reasoning about Web resources by describing their preconditions and effects with terms precisely defined in ontologies. Examples include efforts such as Web Ontology Language for Services (OWL-S) [MBH$^+$04] and Web Service Modelling Ontology (WSMO) [FB02]. OWL-S explicitly defines a set of ontologies that support reasoning about Web services. By contrast, WSMO proposes a conceptual framework within which such ontologies can be created.

Planning based service composition. Koehler *et al.* [KS03] identified several open issues in planning-based Web service composition. Firstly, conventional plans are sequences of actions. Modelling Web service interaction requires control structures involving loops, choice and parallelism. This will enable complex behaviour of Web services, such as concurrent execution, or iteration while a certain condition holds. Consequently, an automated means of assembling complex actions from atomic ones is essential. Furthermore, Web service composition requires modelling a number of sophisticated features compared to the actions in existing planning technologies, such as varying action durations and resource constraints.

One important challenge stems from the fact that classical planning assumes knowledge of all available world states and actions. However, this is an untenable assumption for context aware service composition, as Web services may generate new objects or messages at run-time, which can be further processed by other services. Therefore some means for expressing nondeterminism and unanticipated behaviour of services is necessary.

Context aware service composition. Service composition is a dynamic and flexible process, which allows for reconfiguration as the context changes and therefore removing the need for embedded contextual dependencies. Context commands specify application behaviour in certain contexts. Using context requires actions that are the result of contextual changes to be represented in the user task specification. Traditionally composition requests are pre-compiled and stored in repositories; however, this approach is impractical when dealing with a potentially large number of contextual commands. Some automated means of constructing context aware composition requests is essential.

The frequently changing context and availability of services in a computing environment points to the fact that the process of service composition will be exposed to failures. For example, service composition process may terminate because of missing service descriptions. Furthermore, failures may occur during execution of a composite service, because atomic services may stop functioning because of network disconnection. Suitable fault tolerance mechanisms are therefore necessary to make applications resilient to composition and execution failures.

Additionally, supporting context awareness requires an extensible approach to accommodate an increasing number of context types, their values and the corresponding application behaviour.

1.3 Dissertation aims

This dissertation proposes a framework for context aware service composition, to address the need for a new application model for context awareness. To build a general-purpose, extensible framework that allows composition requests to be assembled based on context, a number of important requirements need to be addressed at the same time. This dissertation investigates the applicability of planning to Web service composition, proposes a new way of handling composition failures and presents an extensible system design.

It specifically addresses the following challenges:

1. *Composition failure recovery.* Service composition may fail due to missing service descriptions or changing context. However, rather than completely failing to satisfy a composition request, sometimes it may be possible and desirable to generate and present a user with a partial, but viable, solution.

2. *Automated context aware request construction.* A user's current context may be used to customise the composition request. Consider a user who

wants driving directions. This request must be considered in context. For example, when a user is driving the navigation application must read out the driving directions. In contrast to constraints arising from user's intention, the context-implied constraints cannot always be predicted and encoded ahead of time. The effect of context parameters on the composition request should therefore be determined in an automated way.

3. *Execution failure recovery.* During execution of a composite service, atomic services may become unavailable, for example because of a network disconnection. The framework should accommodate such failures and employ suitable recovery mechanisms such as service replacement and caching.

4. *Scalability.* The domain size may increase as service providers advertise the services they offer, and as service-oriented architectures gain popularity. Consequently, 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 domain 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, such as possibly using server farms to cope with scale. The linear growth of computer requirements is a common scalability criterion, an observation found in systems research [FGC+97].

5. *Independence of application domain.* The framework should be designed as a general-purpose solution and its implementation should not include any scenario-specific dependencies.

6. *Independence of component technology.* The framework should support a variety of types of component technologies. Web services are only one possible type of component-based technology, along with Distributed Component Object Model (DCOM) [HK97], Common Object Request Broker Architecture (CORBA) [Obj91], and other technologies. Furthermore, the framework should support, at the same time, different types of component in the composite service, as well as services that may emerge in the future.

7. *Independence of composition methodology.* This work aims to employ existing methods for composition, where AI planning is only one possible solution. Therefore it is important that the framework is open to alternative composition methodologies. Different types of planners, as well as

non-planning approaches, such as data view integration [TKA03] can also be used to control service composition. Additionally, different types of composition methods are suited to different types of composition requests and application domains. For example, nondeterministic and partially observable domains may benefit from model-based planning.

8. *Independence of context middleware.* Context middleware is a software component that provides access to different context values that are of relevance to a composition request. Independence of the context middleware is essential for two reasons. Firstly, different context types may be provided by context providers offering services through different middleware solutions. Secondly, context middleware may fail and the framework may need to switch to another context acquisition and management system.

1.4 Dissertation outline

This chapter identified the need for a new methodology for the development of context aware applications. It introduced the idea of context aware service composition, in which applications are assembled dynamically from atomic services. It also presented a framework of general research challenges and the specific aims of this dissertation. This chapter concludes by outlining the contents of this dissertation and by presenting the author's publication record.

Chapter 2 analyses related work in context aware computing, and highlights the necessity for and importance of the new application model for context awareness, by identifying the shortcomings of traditional monolithic solutions. It also sets out specific requirements for the proposed framework, based on a review of existing research in planning-based service composition.

Chapter 3 describes the design of the framework for context aware service composition through a sample usage scenario. It presents the system architecture in terms of its main components and the operations they provide. It also discusses how the framework achieves independence of the component technology and composition methodology by employing internal representations.

Chapter 4 introduces `GoalMorph`, a novel composition failure management system. `GoalMorph` applies context aware goal transformations when composition requests cannot be satisfied, to generate ones that can be partially fulfilled. This chapter also experimentally demonstrates that `GoalMorph` is a practical approach and does not impose a prohibitive composition time overhead.

Chapter 5 presents an implementation of the system architecture, providing details of each of the platform components and showing how they realise the desired functionality. It discusses the applicability of planning technology to the Web service composition problem and describes how the framework employs the TLPlan [BK95] planning system. Finally, it presents the mechanisms used for monitoring and handling failure tolerance during execution.

Chapter 6 describes the performance and scalability experiments undertaken, which demonstrates that the proposed context aware service composition facility is a viable approach. Furthermore it presents the results of qualitative evaluation used to determine the framework's effectiveness in reducing the development effort required for building context aware applications. Finally, it considers how the dissertation aims outlined in Section 1.3 have been met by the design of the proposed framework for context aware service composition.

Chapter 7 highlights the main contributions of this work and the conclusions reached. It suggests areas with a potential for future work, in the context of a commercial deployment of complex composite services. These include privacy and security issues, and a method for scheduling composition requests.

1.5 Publication record

Parts of the work done towards this dissertation have been published[1] in international journals, conferences and workshops as follows.

1. Maja Vuković and Peter Robinson. GoalMorph: Partial Goal Satisfaction for Flexible Service Composition. *International Journal of Web Services Practices*, 1(1–2):40–56, December 2005.

2. Evangelos Kotsovinos and Maja Vuković. su-chef: Adaptive Coordination of Intelligent Home Environments. In *Proceedings of the Joint International Conference on Autonomic and Autonomous Systems 2005 / International Conference on Networking and Services 2005 (ICAS/ICNS 2005)*, Papeete, Tahiti, October 2005. IEEE Computer Society.

3. Maja Vuković and Peter Robinson. GoalMorph: Partial Goal Satisfaction for Flexible Service Composition. In *Proceedings of the International Conference on Next Generation Web Services Practices (NWeSP)*, Seoul, Korea, August 2005.

[1]I was the lead author on all the publications, except for the paper I co-authored with Joachim Peer.

4. Maja Vuković and Peter Robinson. Context Aware Service Composition. In *Proceedings of the Third UK UbiNet Workshop*, Bath, UK, February 2005.

5. Maja Vuković and Peter Robinson. SHOP2 and TLPlan for Proactive Service Composition. In *Proceedings of the UK-Russia Workshop on Proactive Computing*, Nizhniy Novgorod, Russia, February 2005.

6. Evangelos Kotsovinos and Maja Vuković. su-chef: Dynamic Service Composition For Next-Generation Cooking. In *Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems (WMCSA 2004), Poster Session*, Lake District, UK, December 2004.

7. Maja Vuković. Plan Based Application Modeling for Context Awareness. In *Proceedings of the Doctoral Colloquium. The Sixth International Conference on Ubiquitous Computing (UbiComp)*, Nottingham, UK, September 2004.

8. Joachim Peer and Maja Vuković. A Proposal for a Semantic Web Service Description Format. In Liang-Jie Zhang, editor, *Proceedings of the European Conference On Web Services (ECOWS)*, volume 3250 of *Lecture Notes in Computer Science*, pages 285–299, Erfurt, Germany, 2004. Springer.

9. Maja Vuković and Peter Robinson. Application Modeling for Context Awareness. *IEEE Pervasive Computing Magazine, Building and Evaluating Ubiquitous System Software. Work in Progress Section*, 3(3):Page 59, July-October 2004.

10. Maja Vuković and Peter Robinson. Adaptive, Planning Based, Web Service Composition for Context Awareness. In *Proceedings of the Second International Conference on Pervasive Computing (Pervasive 2004), Advances in Pervasive Computing*, volume 176, pages 247–252, April 2004.

Chapter 2

Research context

Research on context aware service composition tackles the problem of developing extensible and scalable applications that adapt to context. It incorporates work from several disciplines, such as context data acquisition, analysis and inference, modelling, management and distribution, as well as service composition.

This chapter describes related work in two main research categories: *context aware computing* and *service composition.* The first section discusses the concepts of context and context awareness, and analyses the middleware for context acquisition and management. It surveys existing approaches for developing context aware applications, such as *task-driven computing*, which is grounded in the idea of service composition. The second section presents the foundations of service composition, the process of constructing flexible software systems from service components. It describes Web services, a technology that facilitates platform independence, interoperability and modularity for Web applications. It also shows how goal-oriented inferencing, an increasingly popular approach from planning technologies, can be applied to the Web service composition problem. The chapter concludes by highlighting the shortcomings of the existing planning based service composition frameworks.

2.1 Context aware computing

With the move from traditional desktop computing to mobile and pervasive environments there is a greater demand for *context awareness*, a need to exploit implicit information in order to adapt application behaviour. Context awareness has gained attention partly as a result of technical advances allowing for low-cost sensing of context.

2.1.1 Context and context awareness

Context definition. Many researchers have attempted to formalise the meaning of context in the computing environment; however, a universally accepted definition is yet to be agreed. According to the Oxford English dictionary the word context refers to *"the circumstances that form the setting for an event, statement, or idea"*. Past attempts to define context in the computing environment originally took the approach of the definition by enumeration and later of using synonyms for context.

Initially, researchers enumerated certain context types, which they considered important and relevant. Schilit and Theimer [ST94] defined context to be: location, identities of nearby people, objects and changes to these objects. The three context classes that Schilit *et al.* [SAW94] later identified are computing, including network connectivity, communication costs, bandwidth, resources, user parameters such as user profile, location, social situation and physical properties, for instance lightning, noise level, temperature. Chen and Kotz [CK00] expanded the taxonomy of Schilit and Theimer, by introducing the time class, which represents parameters, such as time of day, week, month and season of the year. Dey *et al.* [DAW98] refer to context as a user's emotional state, focus of attention, location and orientation, date and time, objects and people in the environment. Defining context by enumeration is, however, an application-specific approach. Furthermore it is not complete, as the list of contextual types is not exhaustive.

More formal and more generic definitions used either the user's environment or the application environment as the basis for establishing the meaning of context. Brown [Bro96] considers context as elements of a user's environment that the user's computing device is aware of. Ward *et al.* [WJH97] view context as the state of the setting in which the application is operating. Similarly, Schmitd *et al.* [SAT$^+$99] described context as:

> "... knowledge about the user's and IT device's state, including surroundings, situation, and to a less extent, location."

In their later work, Dey *et al.* [DA99] discuss that the important aspects of context cannot be enumerated, as they differ from situation to situation and depend on the purpose of the application. Furthermore they formally defined context as:

> "... any information that can be used to characterise the situation of an entity. An entity is a person, place or object that is considered relevant or the interaction between a user and an application, including the user and applications themselves."

Derived from the definition provided by Dey *et al.* in the most general sense, the work presented in this dissertation considers context as the entire collection of entities and their properties that can enter into a meaningful relationship with users during their interaction with applications (including the user and the system themselves), and affect their behaviour.

Context modelling. To make contextual data usable and sharable by applications, it is necessary to model sensor data values. Most current systems use their own method when modelling context, thus making exchange of context and interoperability between existing context aware systems more difficult. To facilitate the development of extensible and interoperable context aware applications it is essential to have a set of principles for specifying any given context from any domain. A set of well-defined, uniform context models and protocols is required.

Context modelling has been the subject of recent research, although primarily embedded in the study of overall software support for building context aware applications, such as toolkits [Dey00] and infrastructures [HL01, Jon02]. In the mobile computing most of the related most work is focused on modelling location information, although location is just one of many context types. Human Computer Interaction (HCI) and Artificial Intelligence (AI) communities are addressing user and task models.

Most context models use standard methodologies for describing context, such as key-value pairs and ontologies. These approaches vary in their level of formalism, abstraction capabilities, support for Quality of Information, ease of retrieval and domain independence. Strang *et al.* [SLP04] classified context applications according to the data structures employed for context modelling, extending the initial categorisation of Chen and Kotz [CK00].

Building on the existing surveys, the most commonly employed data structures are:

1. *Key-value pairs* store a set of data items that contain a key, a context type, and a value, the actual context data. Schilit *et al.* [SAW94] model location information in this way. Similarly, Maass *et al.* [Maa97] store location information pairs in an X.500 conforming directory information tree. This approach is simple and allows efficient pattern-matching queries and retrieval; however it lacks capabilities for modelling complex data.

2. The *Logic-based* approach applies a formal system to describe context in terms of a concluding expression or a fact that may be derived from a set of other expressions. Bates *et al.* [BHB97] and Harter *et al.* [HHS$^+$99] represent a context model as an entity relationship. Chen *et al.* [CFJ03]

devised COBRA-ONT, an ontology for supporting pervasive context aware systems, expressed in the Web Ontology Language (OWL) [CvHH+01]. Using COBRA-ONT they describe places, agents, events and their associated properties in an intelligent meeting-room domain using logical predicates.

The main limitation of this approach is that the scope of the context model defines the limits of the possible domain of application. The design of the context model introduces an overhead in application development, because of lack of automated means in constructing descriptions.

3. *Object oriented* method encapsulates context data as states of the object, which can be accessed through specified methods. Project TEA [SAT+99] introduced the concept *cue* to abstract raw and logical sensor data. Context data is modelled in a layered structure, where context is then described as an abstraction on top of the available cues. The Active Object Model employed in the GUIDE [CMD99] project is specifically designed for a location context. While this approach does provide greater flexibility and modularity of context data, it may result in complex navigational data access.

4. *Markup schemas* are based on the concept of a hierarchical data structure, where each context type is annotated with a description of what role its values play. Pascoe [Pas97] presented the *Stick-e note*, which describes context types as tags and values as their fields in Standard Generalized Markup Language (SGML). Later, Ryan [Rya99] developed ConteXtml, an XML based protocol for exchanging the contextual information, based on the Stick-e note model. Composite Capabilities/Preferences Profile (CC/PP) [Ccp99] is an effort to standardise a language for specifying how computing client devices express their capabilities and preferences. Comprehensive Structure Context Profile (CSCP) [HBS02] and CC/PP Context Extension [IRRH03], based on CC/PP, include component attribute trees for specific context types. However, their extensibility is limited due to constraints of the underlying CC/PP vocabulary.

Context quality. One of the specific characteristics of context is its imperfection, as it often relies on properties of real world entities. Context quality may vary greatly, depending on the data source. Sensors are prone to failures and as a result context data can often be incorrect, inconsistent and incomplete. Consequently context aware applications need to allow for these inaccuracies and uncertainties. Quality of Information metrics aim to enable applications to specify their requirements in terms of data quality.

Dey [Dey00] proposed a Quality of Information model, which includes the following metrics: accuracy, reliability, coverage, resolution, frequency, and timeliness. Reliability defines how tolerant the application is with regard to sensor failures. Coverage and resolution define the set of all possible values for a context attribute, and the change that is required for the context attribute to change respectively. Frequency defines how often the information needs to be updated and timeliness defines the time the application allows between the actual context change and the related notification to the application. Ebling *et al.* [EHL01] defined two Quality of Information metrics. One is *freshness*, which denotes when the context value was last updated and the other is *confidence*, which describes to which extent the value is accurate.

Context awareness. In a general sense context awareness refers to the ability of an application to discover and take advantage of contextual information, such as user location and nearby devices. To determine whether an application is context aware or not, researchers have devised taxonomies of features characteristic of context aware applications.

Schilit *et al.* [SAW94] produced one of the first classifications of context aware applications. It contains two orthogonal dimensions. The first one identifies whether the task is to get information or to execute a command. The second one determines whether the task is executed manually or automatically. Schilit *et al.* identified the following types of context aware features in applications:

1. *Proximate selection:* emphasising or making easier to choose items relevant to the user's context.

2. *Automatic contextual reconfiguration:* addition of new or removal of existing components based on the context.

3. *Contextual commands:* parameterisation of the presentation and behaviour of commands of the user based on the context.

4. *Context-triggered actions:* automatic execution of application commands for the user when the right context exists.

Pascoe [Pas97] described the following four features of context aware applications:

1. *Contextual sensing:* the ability to detect the context and present it.

2. *Contextual adaptation:* automated execution or modification of a service based on the context.

3. *Contextual resource discovery:* exploitation of services and resources relevant to the context.

4. *Contextual augmentation:* association of digital data with the user's context.

Dey [Dey00] considered an application to be context aware if it uses contextual information to provide relevant information and services to the user, where relevance depends on the user's task. Dey proposed three categories of context aware applications, by combining the ideas from Schilit *et al.*'s and Pascoe's taxonomies. The first group of context aware applications presents information and services to a user, based on the context. The second group automatically executes a service when the user enters a specified context. The third group of applications tags information to contexts for later retrieval.

2.1.2 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.*

Required features. Dey [Dey00] analysed a typical development cycle of a context aware application and identifies the following essential features of context middleware for supporting context aware applications:

1. *Context specification.* This provides means of identifying the problem domain, specifying the context, defining the functionality and how the application is adapted given the different context data.

2. *Resource discovery.* Once the contextual information is specified the next step is to discover the relevant and available data sources, such as sensors, which address these context needs and provide Application Programming Interfaces (APIs) to extract the information.

3. *Context acquisition.* Applications then have to be able to query or be notified of changes in the context.

4. *Interpretation.* Low level sensor data has to be interpreted and transformed into high-level context data for use by the application. Ideally, there would be several layers of context abstraction depending on the application needs.

5. *Context storage.* The raw low-level sensor and inferred high-level data need to be stored, in order to allow for tracking of the context history and changes.

6. *Transparent distributed communications.* Acquiring the context from a number of distributed sources should be transparent to applications. The distributed sources need to be synchronised to allow for accurate comparison of the context.

7. *Constant availability.* The behaviour of context aware applications relies on context acquired from context providers, which are often independent distributed systems. Context information must always be available.

Comparison of context middleware. This section analyses the features of architectures for context acquisition and management. It provides a view of the evolution of context middleware, ranging from early architectures focused on making context aware computing applications *possible* to build, such as Schilit's system [Sch95], to the more recent work that aims at *easing* the application development, such as Context Toolkit [Dey00] and Context Weaver [CBC+04]. This analysis extends the set of essential architectural features proposed by Dey, to include the support for the following:

1. *Distributed context repository.* Having a central repository for storing contextual data represents a single point of failure and potential bottleneck for the context middleware. The ability to have multiple context repositories is necessary to facilitate constant availability of context middleware.

2. *Security.* The heterogeneity of wireless network protocols used by the large variety of network connected hardware and software sensors providing context data increases the risk of security compromises.

3. *Privacy.* Context middleware may gather, collate and distribute personal information about individuals. It is essential that users have means for retaining control over the distribution and dissemination of their private information.

4. *Quality of Information.* Failures of sensors and network disconnection may result in imperfect, incomplete, and often unavailable context data. Applications need a way to specify their required level of context quality.

Schilit [Sch95] presented a system architecture that allows for acquiring device and user context. It supports the delivery of context through efficient querying and notification mechanisms. This system consists of three main components: (1) device agents that maintain the status and capabilities of devices, (2) user agents that maintain the user preferences and (3) active maps that maintain the location information of devices and users. Device and user agents are built on an individual basis, tailored to the set of sensors that each uses.

Pascoe [Pas97] prototyped Stick-e, an architecture for context aware computing to allow for developers to design context aware applications and user interfaces. Context awareness is facilitated through a Stick-e note, which is defined in terms of the context it is related to and content that it represents. Applications are developed according to a Model-View-Controller (MVC) pattern, which separates an application's data *model*, user interface called *view*, and *control* logic into three distinct components so that modifications to one component can be made with minimal impact to the others. Using this analogy, Stick-e note represents a model. A controller performs trigger checking, to determine if the specified context is entered. Finally, the view is an application interface composed depending on the Stick-e note description. The Stick-e note framework, however, does not provide support for retrieving, storing or interpreting context. It focuses on allowing application designers to use the context.

The Technology for Enabling Awareness (TEA) [SAT$^+$99] project utilises a four-layered architecture for context recognition. The first layer contains logical and physical sensors. The sensor data is encapsulated in *cues* in the second layer. The third layer derives context from the cues. Finally, scripting primitives in layer four allow for an application to perform basic actions when the user enters, leaves or is in a specified context. The notion of cues provides a separation of context acquisition and context use. Cues write the data to a central repository, based on the blackboard model. This architecture was used to determine the context of a cell phone in order to automatically set its profile, for example switches the ringer off when user is in the meeting. Overall the system provides limited support for context specification and does not allow context storage, and as a result it does not facilitate retrieval of historical context. Furthermore, there is no support for multiple application accessibility.

Dey *et al.* [DAS01] devised Context Toolkit, a conceptual framework and a toolkit for supporting the rapid prototyping of context aware applications. The

Context Toolkit uses a notion of context widgets as a programming methodology. Context widgets are software components, hosted in a distributed infrastructure, responsible for providing applications with access to context information while hiding the details of context sensing. The Context Toolkit provides several services for context acquisition and management. It encapsulates access to context data from sensors through a network API. An interpreting service allows for abstraction of context data. The distributed infrastructure of the Context Toolkit enables sharing of context data. The Context Toolkit provides storage for context data and keeps track of historical context. Finally, it embodies a mechanism for basic access control to give privacy protection.

Hong *et al.* [HL01] proposed Context Fabric, a network-accessible middleware infrastructure. By contrast to the Context Toolkit, this service-based approach makes the components independent, as they do not need to rely on a central manager. However, this increases the complexity of the system, as each component must contain connections, message processing and failure management capabilities. The infrastructure approach promises independence of for the hardware platform, operating system, and programming language used. Hong *et al.* envisaged the following services within the Context Fabric framework:

- Context Event Service: a universal event system, which takes subscriptions, stores them and asynchronously notifies interested subscribers.
- Context Query Service: a service that provides a general mechanism for querying the context state.
- Automatic Path Creation: a service that collects all the relevant sensor data and instantiates data flows required to meet the context needs.
- Sensor Management Service: a sensor discovery and registration service.

Hong *et al.* also defined a Context Specification Language (XML-based) for expressing the context needs at higher levels.

Winograd [Win01] described a data-centric communication and application programming architecture that supports context aware applications, which are part of the Interactive Workspaces project (iRoom) at the Stanford University. This architecture uses a blackboard metaphor with two data layers. The first layer is an Event Heap, which provides distribution of simple event tuples, for instance objects that hold context data, implemented by TSpaces [WMLF98]. A process posts messages to a common shared message board and can subscribe to receive messages matching a specified pattern. A centralised server manages all communication with clients and providers. The second layer is Context Memory, an XML structured database of context data. This approach is simple and robust, as it provides one standard communication link to the blackboard. Winograd outlined necessary extensions, including mechanism for scalability and facility for providing multiple linked distributed blackboards.

Cohen *et al.* [CBC+04] implemented the Context Weaver context middleware, based on the Context Service project [LSD+02]. Context Weaver keeps track of context providers and allows applications to access context by describing the kind of data they require using a uniform interface. It uses descriptive provider queries to ensure transparency in accessing context data and allows for heterogenous data sources. For example, location data can be provided by a sensor in one environment and by a Web service in another. This facilitates portability of Context Weaver across different computing environments. Also, if a provider fails, Context Weaver automatically tries to rebind the application to another provider of the same kind of data. The Context Privacy Engine, embedded in Context Weaver, allows specification of access controls for each item of context data. Both administrators and individuals who are subjects of context information can specify privacy and access control policies.

Summary. Table 2.1 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.

2.1.3 Models for context awareness

Legacy application models are typically vertically integrated monolithic services, which provide the entire solution to computational problems handling user's tasks. They carry a high development cost with inherent inflexibility and as such are not suitable to support context awareness. This section reviews the existing approaches for developing context aware applications.

Task driven computing

Users employ a number of computer applications and services to perform their tasks, such as writing trip reports, running research experiments, accessing in-

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

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

Table 2.1: Comparison of context middleware

formation portal and booking opera tickets. Research in task based computing deals with the problem of task modelling and management. Task models capture what users need from the computing environment for each of their tasks and task management enables the automatic configuration of computing environments.

Task based computing relates to context aware service composition in two ways. Firstly, the task management process itself is context-aware. For instance, tasks can be suspended automatically when a user moves away from the device; or a desired task, such as navigation assistance, can be resumed automatically when a user enters his car. Secondly, tasks are represented as collection of services involved, therefore task management becomes essentially a service composition problem. This section provides a review of task based approaches for building adaptive applications.

Background. Reaching for Weiser's vision [Wei91], Banavar *et al.* [BBG+00] analysed the nature of pervasive applications. They suggested that the traditional view of computing devices and applications in the context of desktop computing is not compatible with the way mobile computing devices are used and mobile applications built. As a result, Banavar *et al.* established models of computing devices, applications and environments in pervasive computing settings, as part of their Platform Independent Model for Applications (PIMA) project. They treat the device as a portal into an application space, rather than the repository of custom software managed by the user. An application becomes the means by which a user performs a task, whilst the environment is viewed as the user's information enhanced surroundings, and not a virtual space that stores and runs software.

Based on these propositions they identify the following challenges for an application model for pervasive computing:

1. *Device neutral application:* An application should be developed independently of the device on which it may be used, to accommodate for the variet of different devices available for use in pervasive environments.

2. *High-level user interaction:* For an application to be device independent its description should capture the purpose of user interaction at a high level, rather then including the rigid decomposition of the interaction, which may be device specific.

3. *Abstract service descriptions:* In dynamic pervasive environments resources appear and disappear over time. Therefore an application model should not make assumptions about the availability of services, rather they should be specified in an abstract manner.

Banavar *et al.* presented a new application model, in which the structure of a program is described in terms of tasks and their sub-tasks, which is, at its core, a service composition problem. This requires specification of an abstract service description language, identification and description of abstract interaction elements and services, and development of a navigation model for managing the task-based model for the program structure.

The roots of this model are present in a number of existing, mature technologies. For example, work on User-Interface Management Systems (UIMS) [Ols91], separates User Interface (UI) from the rest of the application logic. Protocols, such as Remote Method Invocation (RMI) [WRW96] enable the communication between distributed components. Java [Fla04] makes it possible to develop and deploy device independent code. Finally, component frameworks, such as CORBA [Obj91] allow devices to discover services and adapt application functionality to changes in the user environment.

Platforms. The Portolano [EHAB99], Oxygen [Der99], Aura [GSSS02], and Gaia [RHC$^+$02] projects investigated the idea of task-driven computing in ubiquitous environments. The Portolano project at the University of Washington is motivated by task-oriented applications, and focuses on the infrastructure and interface aspects in its implementation. The authors envisioned a computing environment with multiple user interfaces, which rely upon user intent, inferred from the user's interaction with the environment, rather than explicit user direction as in PIMA. The Portolano project, like PIMA, also considers applications as collections of network-based services organised into extensible horizontal layers, which interact with applications and users.

Oxygen [Der99], a project at the Massachusetts Institute of Technology, focuses on a number of environment-enabling technologies to improve the user experience. Its authors believe that monolithic software will be replaced by dynamic mechanisms for application delivery. As part of their work on the automation of everyday tasks and adaptation of machines to user needs, they develop *Pebbles*, platform-independent software components. Each pebble is described in terms of formal interface specifications and informal descriptions. Saif *et al.* [SPP$^+$03] dynamically assemble pebbles using a planning mechanism in response to evolving system requirements. This architecture is based on the notion of generic plan customisation, rather than on-line planning. Tasks are, however, explicitly defined, as in PIMA.

Project Aura [GSSS02] at Carnegie Mellon University, aims to support user mobility and resource variability while minimising distraction of the user. This is being addressed at all system levels ranging from hardware, operating system and application up to the end-users. In order to maintain a user's computational task in a mobile environment, Aura introduces a new layer in system abstraction, called Prism. It lies above the application and service layer, but below the user layer, and enables task reconfiguration as the context changes. Prism consists of three components: Task Manager, Context Observer and Environment Manager. Task Manager is the component responsible for explicit representation of users tasks. Context Observer allows configuration of tasks according to the environmental characteristics. Finally, Environment Manager facilitates resource monitoring and adaptation. Wang *et al.* [WG00] used the approach of task-driven computing, based on the Aura architecture. They also consider the task as a coalition of abstract services. Aura continuously monitors the environment to detect when task requirements are not fulfilled and initiates any necessary reconfiguration.

Gaia [RHC$^+$02] is middleware for managing resources in physical spaces, developed at the University of Illinois at Urbana-Champaign. It provides user-oriented interfaces for physical spaces populated with network enabled computing resources. Gaia enables the development of applications that are adaptable and customisable based on the space context. Within the Gaia framework, Hess *et al.* [HRC02] present a method for building applications in ubiquitous environments. They introduce the concept of User Virtual Space, which associates data, tasks and devices with users, enabling application portability across computing environments. The main focus of this work is on the application construction, to allow for adaptation given the contextual changes. Hess *et al.* develop a new design pattern, called Model-Presentation-Controller-Coordinator (MPCC), which decouples application components and exposes the internal structure of the application. Applications are described in generic terms and may be customised for the resources available in a particular space.

2.2 Service composition

This section introduces the foundations of service composition and reviews a number of existing component technologies. It presents the Web service component development technology and discusses how AI planning can be applied to the problem of Web service composition.

2.2.1 Foundations

Service component

The concept of component-based system development emerged from the increasing demand for software to dynamically grow and address changing requirements [CKJH02]. To achieve software extensibility and adaptivity this approach is based on supporting reusability of software components.

One of the first approaches that recognised the value of software *modularisation* to facilitate flexible and comprehensible system design and to reduce the development time, was that of Parnas [Par72]. In order to write large systems consisting of smaller modules, DeRemeer *et al.* [DK75] identified the need for a *module interconnection language* to assemble modules into larger systems.

Smalltalk [Ing78] is a programming system based on the metaphor of communicating objects. It represents an early step towards modularised system development. Smalltalk formed the basis for object-oriented programming, which later generalised to *component-oriented* computing.

Different component definitions emerged in different fields of software engineering research. In the most general sense, a component is a well-conceived, prefabricated, reusable unit of deployment and composition. Syzperski [SP97] defines the software component as follows:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Derived from this definition, the work presented in this dissertation considers a service as a specific instance of a software system that carries out a computational operation on behalf of a user. It exports its interfaces and is capable of being accessed via standard network protocols.

Service composition

Service composition is the process of constructing a complex composite service from atomic services to achieve a specific task. The process of composition, as Shaw *et al.* outlined [SG96], involves: formal specification of static and behavioural properties of the component, matchmaking algorithms and decomposition techniques.

Service composition models can be categorised into *centralised* and *distributed* ones. In centralised models, there is a single component responsible for assembling a composite service. By contrast in distributed or cooperative models, service providers interact to devise composite services. This approach introduces trust issues as services depend on each other to guarantee the overall successful completion of the composition and execution. In a centralised system a central broker selects and composes services, thus is more straightforward.

Chakraborty *et al.* [CJ01] classified composition into *off-line* and *on-line* based on the level of automation and dynamics in the composition process. An off-line, static approach involves pre-compilation of the composite service prior to the user's request, at system design time. This is utilised primarily in stable environments where context and available resources can be determined in advance. In contrast, an on-line, dynamic approach refers to service composition being performed on user demand, at run-time. This is suitable for exploiting the current state of available services and making adaptations based on run time parameters, such as the available bandwidth and the cost of executing the various sub-components.

Service composition has traditionally been viewed as a static process performed manually by the developer at design time. With the emergence of the Internet, service composition shifted towards operating at run-time, orchestrated by a centralised or distributed composition manager component. As mobile environments become increasingly common, composition is performed on an ad-hoc basis, in a distributed, peer-to-peer way.

Aside from description languages and matchmaking algorithms, service composition requires suitable failure recovery mechanisms to handle service discovery, composition and execution failures. Research into service composition can be grouped in three categories: service description languages, composition methodologies, and service composition frameworks. The next sections review service description and discovery formats, composition frameworks and architectures for planning-based composition.

2.2.2 Component technologies

Several middleware architectures, which aim to simplify the development of interoperable object-based distributed applications, have been developed. To enable services to interact and achieve the desired goal a mechanism for self-advertisement and discovery is required. A number of existing distributed models for computation, which incorporate means of describing (service) components and discovery technologies that enable automatic service configuration are described in this section.

CORBA [Obj91] is middleware for architecture-independent development of object-based distributed applications transparent to the programmer. Object Request Brokers (ORBs) facilitate communication between nodes in heterogeneous environments at the object level. They provide mechanisms for object discovery and instantiation on remote machines, and marshalling and unmarshalling of object parameters. Furthermore ORBs handle security, object retrieval, and method invocations. CORBA also provides a significant number of support services. The event and notification services provide a substrate for easier asynchronous interaction between objects. The naming service handles associations between names and objects, including name binding and resolution. The collection service allows the manipulation of several objects as a group. The concurrency service mediates simultaneous accesses to an object so that consistency is not compromised. The object trading service facilitates the offering and discovery of instances of services of particular types.

Wollarth *et al.* [WRW96] presented the Remote Method Invocation (RMI) mechanism for creation of distributed object-based applications in Java. RMI, like CORBA, uses serialisation techniques to marshal and unmarshal object parameters. In contrast to CORBA, RMI requires that code is written in the Java programming language. Using RMI, entire objects can be passed and returned as parameters in remote method invocations whereas in CORBA the parameters need to be primitive data types, references, or structures composed of the two. RMI allows for any new Java code to be sent across the network and dynamically executed at run-time by foreign Java Virtual Machines (JVMs). This way developers do not need to define a fixed codebase at development time although they need to ensure that the necessary class definitions are available.

The Distributed Component Object Model (DCOM) [HK97] extends the Component Object Model (COM) [Cor95] to support communication among objects on different, network connected, computing devices. It supports remote objects by running a protocol called Object Remote Procedure Call (ORPC), which is built on top of Remote Procedure Call (RPC) in Distributed Computing Environment(DCE) and interacts with COM's run-time services. A DCOM server provides objects of a particular type at run-time, and supports multiple interfaces, each representing a different behaviour of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. The client object then starts calling the server object's exposed methods through the acquired interface pointer as if the server object resided in the client's address space. The DCOM server components are language independent.

2.2.3 Service composition frameworks

This section surveys existing service composition frameworks. The review outlines the type of service components each architecture supports and the methodology it employs for service composition, and analyses each system's ability to recover from both composition request failures and run-time service failures.

eFlow [CIJ+00] is a system for on-line, adaptive composition of e-services, developed at HP Laboratories in Palo Alto. It models composite services as a graph defining the order of their execution. Graphs contain three types of nodes: service nodes, decision nodes and event nodes. Decision nodes carry flow control rules. Event nodes enable service processes to send and receive information about suspension, completion and failure of service. eFlow performs composition in a centralised way. It uses a service broker to discover a service, which can fulfill the requests specified in the service node definition. eFlow is based on Java and is compliant with workflow and Internet standards, such as XML and the Workflow Management Coalition Interface [Hol05], targeting fixed infrastructure type services.

Mao *et al.* [MKB01] developed Ninja [Nin97], a system for automated composition of existing, XML-described services through heterogeneous devices and networks, given Quality of Service metrics. It is based on a cluster computing platform, which utilises redundant control paths to enable fast fault-recovery. The central element of Ninja is Automatic Path Creation (APC), a component that identifies a set of services and the corresponding network connectors for devising and executing a composite service. To devise a composite service APC creates a *logical path* by searching over a graph of the service space using a shortest path strategy. It then creates a *physical path* to locate service instances, which is used to instantiate, execute and monitor a composite service. Ninja achieves good resource utilisation by strategically placing and locating services and dynamically adapting to resource availability.

ICrafter [PLF+01] is a service framework for interactive workspaces, a class of ubiquitous computing environments. ICrafter services are devices, such as a scanner or applications, for example a Web browser, with which a user interacts through *appliances*, which include input devices. ICrafter provides an infrastructure for UI selection, generation, and adaptation to offload services and user input devices. It is designed with the aim of automatically creating UIs for composite services. The ICrafter architecture utilises a central composition model, which supports both off-line composition using service templates, and on-line composition. Services are described using an XML-based Service Description Language (SDL).

SAHARA [RAC+02] is an architecture for the creation, deployment and management of services, enabling composition across independent service providers. It employs a layered reference model, where services range from providing basic network reachability and creating overlay networks, to instances of application building blocks, requiring processing and storage. Services are made available over the Internet and are composed by service level paths. SAHARA supports both centralised and distributed composition models. It embodies the following mechanisms measurement-based adaptation, utility-based resource allocation, trust management, service verification, and policy management. Finally, it allows for heterogeneous service composition across different service providers.

Chakraborty *et al.*[CPJ+02] presented *Anamika*, a distributed, decentralised architecture for dynamic service composition in pervasive environments, based on a peer-to-peer model. Services are described using DAML-S [ABH+02] and incorporate information about inputs, outputs, functionality classification, as well as platform specific information, such as processor type. This is used by the composition broker to reason about possible service compositions. The Anamika architecture is tolerant to faults arising form service and network unavailability. Their peer-to-peer model allows any device to act as a broker facilitating service composition, making the design immune to a single point of failure.

	Service Framework					
Feature	**eFlow**	**Ninja**	**ICrafter**	**SAHARA**	**Anamika**	**TCF**
Distributed model	×	×	×	✓	✓	×
Dynamic composition	✓	✓	✓	✓	✓	✓
Component technology	XML-based	XML-based	XML SDL	XML-based	DAML-S	DAML-S, UPnP
Composition method	Graph	Graph	Graph	Graph	DAML-S	DAML-S
Composition failure recovery	×	×	×	*	*	×
Execution failure recovery	✓	✓	✓	✓	✓	✓

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

Table 2.2: Comparison of service composition frameworks

Masuoka *et al.* [MPL03] developed the *Task Computing Framework* (TCF) to allow users to compose and execute complex tasks in ubiquitous environments. Their architecture provides a Task Computing Client to discover, create, manage and manipulate services. TCF allows for both on-line and off-line composition. It embodies a centralised composition model to assemble semantically enriched Web services, described in DAML-S, as well as plain UPnP services.

Summary. Table 2.2 summarises the features supported by each system. This review has shown that existing composition architectures predominantly accommodate components described in XML-based languages. SAHARA and Anamika are examples of distributed composition architectures, all others follow a centralised approach. Early works, such as eFlow and Ninja, employ graph structures to model composite services, while the more recent work uses DAML-S structures to store composite service templates. All frameworks are resilient against run-time failures, such as those caused by system overload or network level disconnections. None, however, offer full support for composition-level failures, for example arising as a result of missing service descriptions.

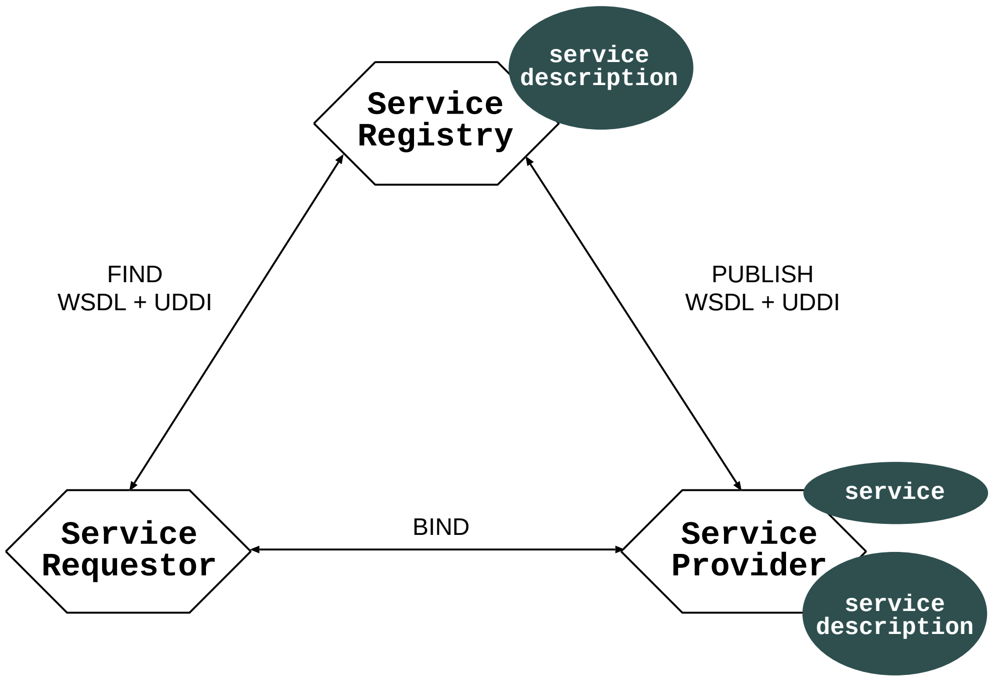


Figure 2.1: Web service architecture. Chart taken from Gottschalk *et al.* [GGKS02]

2.2.4 Web service composition

Web service technology [GGKS02] aims to provide methodologies for constructing distributed, component-based applications automatically. Just like its predecessors CORBA, DCOM, and RMI, it is built on the idea of separation of the component's interface from its internal mechanism, thus allowing for transparency in interoperation. This section presents the Web services architecture and Web Service Description Language. It also discusses a number of semantic languages for describing Web service capabilities.

Web service architecture

A Web service is a network-accessible software component, which performs a specific computational task. Figure 2.1 shows the three main operations in the Web service architecture: service description, discovery and invocation.

Service providers describe Web services using an XML-based formal notation called Web Service Description Language (WSDL) [CCMW01]. This includes message formats describing the operations supported, transport protocols, and the Web service's network location. Services are made available by having their WSDL and related information published to the Universal Description, Discovery and Integration (UDDI) [Udd00] registry. A service consumer queries UDDI to discover available services matching its request. It fetches the WSDL file of the wanted service in order to bind to it. To invoke the service, a requestor uses the specified protocol, most commonly Simple Object Access Protocol (SOAP) [Soa00].

Web service description. A WSDL document defines services as collections of network endpoints, called ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This approach facilitates the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitute a reusable binding. Finally, a port is defined by associating a network address with a reusable binding. A collection of ports define a service.

Web service registry. UDDI is a central repository of services, where service providers advertise themselves in terms of physical contact. To publish a service a provider submits the corresponding WSDL document to a UDDI registry, making it available for any developer or executing service to discover. Service requestors describe the constraints on service requirements and submit their request to UDDI, which in turn produces a list of any matching services and returns their WSDL descriptions.

At present UDDI uses a simple approach to capturing business and service semantics and search mechanisms. Services are described by following four data types: `businessEntity`, `businessService`, `bindingTemplate`, and `tModel`. The `businessEntity` provides information about a service provider, and can contain one or more `businessServices`. The technical and business descriptions for a Web service are defined in a `businessService` and its `bindingTemplates`. Each `bindingTemplate` contains a reference to one or more `tModels`.

Web services are located based on their name, ports and description of features and categories using a `tModel` metadata construct. `tModels` enable the categorisation and identification of entities registered in UDDI. The information that makes up a `tModel` includes a key, a name, an optional description, and a URL that points to a location for additional information.

UDDI supports only a keyword based search of `businesses`, `services` and `tModels` in its repository, and no form of inference or flexible match between keywords can be performed. Furthermore WSDL specifications are focused only

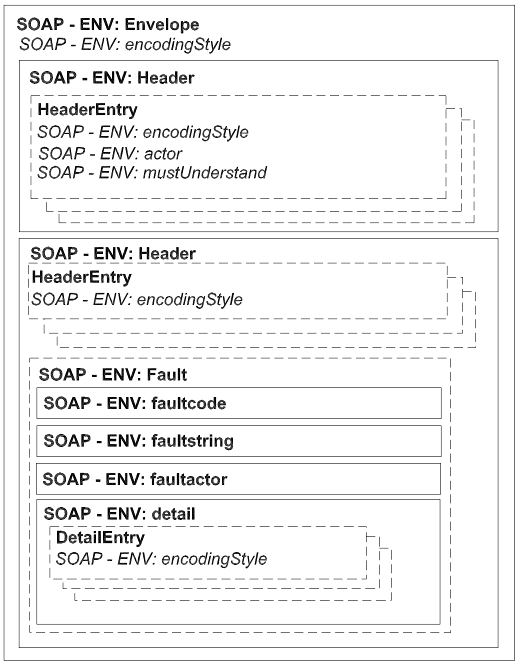


Figure 2.2: SOAP message structure [Soa00]

on syntactic aspects of a service, thus making it is impossible to locate a Web service on the basis of what problems it solves [PKPS02].

Web service invocation. The Web services architecture uses Simple Object Access Protocol (SOAP) [Soa00] to enable communication between Web Services. SOAP is an XML-based standard mechanism for communicating document-centric messages and remote procedure calls using XML.

Figure 2.2 shows the structure of a SOAP message, encapsulated in an envelope. It contains a header and a body for the message. The header stores information about the message. For example, a header can contain the date the message is sent or authentication information. It is not required, but, if present, must always be included at the top of the envelope.

Composite Web service description. A modelling language is needed to express the logic of a composite Web service. Recently languages such as Business Process Execution Language For Web Services (BPEL4WS) [CAD+05] and Web Service Choreography Interface (WSCI) [Wsc02] have emerged to provide a means of defining long-lived composite processes.

BPEL4WS, an XML-based language, models composite Web services in terms of the interaction between the participating Web services and the user. It specifies the role of the partners who provide each Web service and the flow of the messages they exchange. Each partner definition includes their name, role, and the link to the service definition in their WSDL file. BPEL4WS supports control flow constructs to enable sequential and parallel execution of services, conditional choice and if clauses. Furthermore it provides a scoping system to allow the encapsulation of logic with local variables, fault handlers, compensation handlers and event handlers.

Semantic Web service description

Composition requires the ability to specify sequences by selecting suitable building blocks in the form of Web services. WSDL provides a purely syntactical description of the interface in terms of messages, operations, and protocols used by a Web service. A more comprehensive information model of a Web service is therefore required to reason about Web service capabilities. A number of efforts towards semantic markup languages for describing Web Services are reviewed below.

OWL-S. The Web Ontology Language for Services (OWL-S) [MBH$^+$04], previously known as DAML-S [ABH$^+$02], is an OWL based [CvHH$^+$01] ontology for describing capabilities and properties of Web services. OWL-S is used respectively by the service providers to advertise their services and by service requestors to specify the desired services they wish to access.

Figure 2.3 shows the top level of the service ontology for the `Service` class, which provides a simple means of organizing the parts of a Web service description. Three components constitute the description of a Web service in OWL-S. `ServiceProfile` represents information about the service provider. `ServiceModel` specifies service functionality, in terms of its inputs, outputs, preconditions and effects (commonly abbreviated as IOPE). Details about the communication protocols used for invocation of the Web service are stored in `ServiceGrounding`.

There are three types of process in OWL-S, atomic, composite and simple. `AtomicProcess` is a basic building block and is not decomposable. It is associated with `ServiceGrounding` and can be directly invoked. The `CompositeProcess` is formed by other atomic and composite processes. Its decomposition is specified using a set of control constructs expressing sequential, conditional or iterative execution of atomic services. `SimpleProcess` represents the "black box" view of a process, and cannot be directly executed.

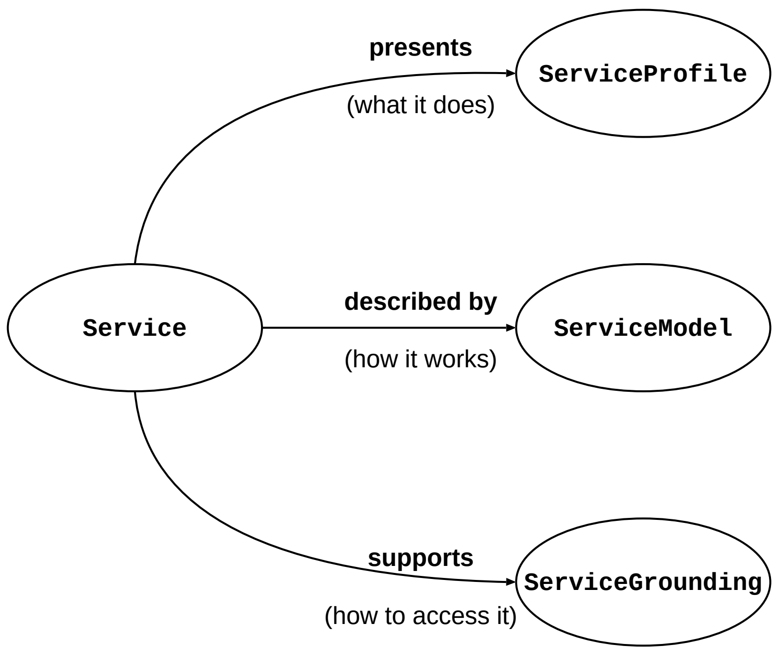


Figure 2.3: OWL-S service ontology. Figure taken from [ABH+02]

WSMO. Roman *et al.* [RKL+05] proposed Web Service Modelling Ontology (WSMO) [Wsm04] as a means for semantically annotating Web services to enable automatic discovery, selection, invocation and composition. It is based on Web Service Modelling Framework (WSMF) [FB02], which consists of the following four elements:

1. *Ontologies.* These provide the domain terminology in terms of concepts and their relationships.

2. *Goal Repositories.* These store problem definitions, each consisting of expected Web service *preconditions* and *postconditions.*

3. *Web Service Descriptions.* These contain non-functional properties, capability descriptions (IOPEs) and interface descriptions of Web services.

4. *Mediators:* facilitate Web service interoperability by resolving heterogeneity problems of different ontologies.

SESMA. In our prior work [PV04], we proposed SEmantic Service MArkup (SESMA), an XML-based annotation format that allows tight integration with the existing Web service standards WSDL, SOAP, BPEL and XML. SESMA can be used to annotate service descriptions and process interfaces based on WSDL, as well as to directly annotate fragments of BPEL process definitions.

SESMA annotations, like OWL-S and WSMO, can be separated according to the type of information described as functional and non-functional. The functional profile allows for description of service operations in terms of preconditions and results. Furthermore each service effect may have a secondary precondition, describing a precondition for a specific effect and its corresponding success condition. The non-functional profile is a collection of entries describing different aspects of the profile, such as provider details and Quality of Service parameters.

Comparison. OWL-S explicitly defines a set of ontologies that support reasoning about Web services. By contrast, WSMO defines a conceptual framework within which such ontologies can be created. SESMA markup defines a set of elements to describe a Web service and augments existing description formats, such as WSDL and BPEL4WS.

At the core of WSMO are mediators, mapping programs that solve interoperation problems between Web services. OWL-S and SESMA do not make any distinction between different types of Web services. Instead, OWL-S provides Web services with the information needed to find existing mediators that can resolve their mismatches. Moreover, OWL-S can even create mediators through the process of Web service composition.

OWL-S introduces its own process modelling ontology and WSMO proposes the use of Abstract State Machine (ASM)-based process framework. As a result processes both in OWL-S and WSMO require the use of their respective process description concepts, making it difficult to reuse existing process descriptions, such as BPEL definitions. On the other hand, SESMA, is designed to augment existing representations such as BPEL. Mandell *et al.* [MM03] provided an extension of BPEL's meta model to use semantic Web constructs inside BPEL documents.

2.2.5 Planning-based service composition

Web service composition has recently gained considerable attention to support business-to-business or enterprise application integration. It has been applied in a number of domains ranging from travel planning [MS02], dining and entertainment booking services [DC02, PF02], content and news conversion services [SRvS03], to managing supply-chain operations [ZBL+03] and automating IT processes in the telecommunications domain [ACD+05].

Available methods for composing Web services include scripting and coordination languages [GC92], rule-based systems [PF02], planning [WSH$^+$03], situation calculus [BCGM03], data view integration [TKA03], and integer programming [ZBN$^+$04], to name but a few. They vary in their ability to represent and model non-functional properties of the service, to verify the correctness of the composite service, and to automate the process of service composition fully.

Planning offers a scalable and efficient approach to service composition. It allows for a composition request to be expressed in terms of goal conditions that specify a set of constraints and preferences. This section introduces planning technology and discusses the main research efforts in its application to automate service composition.

AI Planning

Planning is a problem solving technique, where knowledge about available actions and their consequences is used to identify a sequence of actions, which, when applied in a given initial state, satisfy a desired goal [RN95]. There are three main inputs to a planner: initial state, goal state and domain description. The initial state describes the starting state of the application domain, commonly called world. The goal state describes the desired world state. The domain describes actions that, when invoked, transform the world states. The output of the planning process is a plan, a sequence of actions that can be executed in order to achieve the desired goal state.

Planning domain and problem definition. For planning systems to be able to reason about problems a formalised domain, describing the semantics of available actions is necessary. Domain definitions describe causal laws and relationships between actions formally.

The simplest form of domain formalism is based on the state-transition model. The state describes the world at a certain point in time, such as an initial state or goal state. Actions perform transitions between states. The specific representation used by STRIPS [FN71], an early automated planner has become a base for representation languages. This specification is now known as STRIPS.

STRIPS describes each action with a precondition list, add list, and delete list. An Add and `delete` lists are dynamic lists of states. The add list describes the list of the action's postconditions, which will be added to the description of the world state following the execution of the action. By contrast, the delete list contains the list of the actions's postconditions, which will be deleted from the world state description once the action completes its execution.

Preconditions and postconditions can contain arbitrary well-formed formulae in first-order logic. A state is modelled as a finite set of ground literals and there are no external actors in the domain. STRIPS applies the *closed-world assumption*, where the only atoms or state-variable descriptions that are true in a state are the ones explicitly specified in the state. Action execution changes the truth value of the ground literals describing each state, such as adds or removes them from the world state.

Such a formalism is suitable for describing *restricted state-transition systems*, which are deterministic, static, finite, and fully observable with restricted goals and implicit time (where actions and events have no duration). To describe and model more complex and realistic domains, extensions to this representation are required. Pendault [Ped94] developed Action Description Language (ADL), which has constructs to specify quantification over preconditions and effects, and conditional effects.

In order to optimise their performance, most planners use their own representation formalism. Ghallab *et al.* [GHK$^+$98] developed the Planning Domain Description Language (PDDL) in an effort to provide a standard language for describing the planning domain and problem specification and to enable interoperability between different planning systems. Two variants of the PDDL representation are available: STRIPS-based and ADL-based.

In order to facilitate modelling of real world domains, a number of different versions of PDDL were developed. They support more advanced features, such as concurrently executing actions and conditional choices. PDDL 2.1 [FL03], based on ADL planning, is an extension of PDDL for expressing temporal planning domains separated into different levels of expressiveness. PDDL 2.2 adds derived predicates and timed initial literals. Bertoli *et al.* [BCLP03] extended PDDL to express nondeterminism, limited sensing and iterative conditional plans. Younes *et al.* [YL04] proposed probabilistic PDDL (PPDDL), an extension of PDDL that permits modelling of probabilistic effects. Gerevini *et al.* [GL05] devise PDDL version 3.0, which adds support for constraints and preferences, expressed in a restricted temporal logic.

Planning systems. Planning technologies differ in the complexity of the problems they can handle and the representations that they use. Furthermore, they employ different search algorithms to synthesise plans and the constraints they observe [GNT04]. A number of different planning methodologies are described below. The list is not meant to be exhaustive, rather the aim is to provide an overview demonstrating the range of available planning technologies and their advantages and disadvantages.

1. *State space planning.* State based planners represent the simplest form of planning algorithm. They are search algorithms in which the search space is a subset of the state space. They can be classified into *forward* and *backward*, based on the starting point of their search, initial state and goal state respectively. A forward chaining planner searches in the space generated by applying to each state all actions whose preconditions are satisfied, starting at the initial state. In contrast, backward search algorithms [FN71] start at the goal and apply inverses of the planing actions to produce sub-goals, stopping if the initial state is reached. The main limitation of state-based planners is that their performance reduces with the size of the search space. To address this limitation researchers employ *heuristic functions* to estimate the usefulness of the alternative actions a planner can choose from. Heuristic methods are found through discovery and observation of the planning process. They guide search algorithms often based on feedback from past executions. This initiated a new type of planning, *planning with control knowledge*, as discussed later on. Heuristic Search Planner (HSP) [BG98] is an example of a forward heuristic planner.

2. *Plan space planning.* Plan space planners, such as Universal Conditional Partial Order Planner (UCPOP) [PW92], search the space of partially specified possible plans. As a result, the searching process becomes a plan refinement operation. There are two kinds of step that can be taken in constructing a plan: adding an action, or adding an ordering constraint between actions. This type of planning is called "partial order planning", because until the ordering constraints are added, the order in which actions are taken is not specified. This approach avoids extensive backtracking that slows down a state-space planner.

3. *Planning graph techniques.* Graph based planning algorithms employ graph structures to represent search spaces. Given a problem statement, the planning system explicitly constructs and annotates a compact structure called a *planning graph.* It represents a plan as a flow of truth values through the graph, which has the property that useful information for constraining the search can quickly be propagated through the graph as it is being built. Graph based planners then exploit this information in the search for a plan. Graphplan [BF95] takes the initial conditions and action definitions and uses them to construct a levelled graph. The initial conditions form the first level of the graph, while each subsequent iteration i constructs the level i consisting of actions that might be performed at time i. Graphplan performs a backward-chaining search using the information propagated when

creating the graph, thereby limiting the amount of searching performed. Koehler [Koe99] develops the Interference Progression Planner (IPP) planner, by extending Graphplan to the backward search algorithm to handle conditional effects and proofs.

4. *Hierarchical Task Network Planning.* Hierarchical Task Network (HTN) planners, such as Simple Hierarchical Ordered Planner (SHOP2) developed by Nau *et al.* [NMAC$^+$01], are based on the notion of *hierarchical decomposition*, also known as reduction or expansion of an action. This process decomposes an abstract action into a group of steps that form a plan to implement the action. The main objective is to produce a sequence of actions that perform some activity or task. The description of a planning domain consists of a set of actions, as well as a set of methods, which prescribe how to decompose a task into subtasks. The description of a planning problem contains an initial state; however, instead of a goal formula there is a partially ordered set of tasks to accomplish. Planning progresses as a recursive application of the methods to decompose tasks into smaller and smaller sub-tasks, until primitive tasks, which can be performed directly using planning actions, are reached. For each composite task, the planner selects an applicable method and instantiates it to decompose the task into subtasks. If the plan later turns out to be infeasible, the planner backtracks and tries other applicable methods.

5. *Model Based Planning.* Planning based on model checking is a methodology that aims to address nondeterminism, partial observability and extended goals. It treats the domain as a nondeterministic state-transition system, where an action may have multiple outcomes. Temporal logic formulas are used to express the set of goal states and the conditions of the final plan execution. Planners use a state transition system and a temporal formula to generate a plan that controls the system evolution so that all of the system's behaviours make the temporal formula true. SimPlan [KBSD97] is an example of one of the first such planners.

Planning-based service composition frameworks

By explicitly declaring Web services as processes in terms of their inputs, outputs, preconditions and effects, goal-oriented inferencing from planning technologies can be applied to the Web service composition problem. This section presents the main technical challenges for automated, planning-based, Web service composition, by extending the set identified by Koehler *et al.* [KS03]. It then compares a number of existing planning based service composition frameworks.

Required features. Planning-based service composition frameworks should support the following features:

1. *Extended goals.* Users' requests may often involve complex conditions affecting the behaviour of a composite service. Aside from specifying the task intention, users should have some means of specifying additional conditions. For example, temporal and Quality of Service constraints.

2. *Complex actions.* The process of Web service composition requires modelling complex executing actions such as concurrently executing actions, varying action durations and conditional choices, as well as sequentially executed services.

3. *Dynamic composition.* Static service composition involves pre-compilation of the composite service prior to a user's request. Dynamic composition is essential for exploiting the current state of available services and making adaptations based on run time parameters, such as bandwidth and the cost of executing the various services.

4. *Recomposition.* As composite services may be executed in a dynamic environment, the context may change and services may become unavailable. Therefore it is necessary to have some means of recomposing the service on the fly.

5. *User interaction.* Whilst service composition is an automated process, it is necessary to allow users to provide feedback when they so wish or moreover be integrated in the composition process. For example, aside from providing input parameters, users may need to guide the composition, by selecting the services and re-defining the goals or guide the failure recovery process.

6. *Automatic service discovery.* Working with a limited domain of services or predefined service types limits the potential of service composition. Moreover, new services, possibly with new capabilities, may become available or existing ones may change their functionality. Having an automated means of service discovery is therefore an essential feature.

7. *Nondeterminism.* The planning system cannot foresee the exact interaction that will take place between Web services, the outcomes of service execution are unpredictable. For example, the planning system does not know in

advance whether or not a booking can be made in the desired restaurant, as it does not have access to information about current table availability.

8. *Implicit task specification.* One of the key challenges in service composition is task inference, the ability to anticipate a user's computational needs, without requiring explicit user input. Unfortunately, while users know what their task intention is, they may not know how to realise it in a particular implementation language. Requiring a user to understand the low-level details of a potentially unfamiliar computing environment is impractical.

9. *Resource constraints.* In the real world, services consume resources, such as network bandwidth, and have a monetary cost associated with their execution. Moreover, users may have specific Quality of Service requirements on the failure rate, latency, recovery time and generated traffic of participating services. Therefore, a mechanism to effectively handle the generation and consumption of resources properly and to check plans for satisfaction of resource and time constraints is essential.

10. *Composition failure recovery.* Planner failures may arise due to missing service descriptions, wrong goal descriptions or the planner having incomplete knowledge of the world. A more flexible means of dealing with this type of failure is required, instead of returning no response to the user.

11. *Execution failure recovery.* Composite services will be executed in an unpredictable and dynamic environment. It is anticipated that plans may fail during execution, for example as a result of the loss of network connectivity. A mechanism for fault-resilience is required, such as replacement of the unavailable services with alternative ones.

Detailed discussion of planners and their applicability to the Web service composition problem follows in Section 5.1.2, including the selection of the planner to be used in the proposed context aware service composition framework.

Comparison of planning-based service composition frameworks. Wu *et al.* [WSH$^+$03] investigated the applicability of SHOP2 [NMAC$^+$01] to automated Web service composition, applying it to a scheduling scenario. Their service composition framework utilises a mediated approach to service composition. It augments the online execution of information provider services with off-line simulation of world altering ones. A monitoring component handles SHOP2's calls to external information provider services during planning. SHOP2 allows control constructs such as conditional choice and iterative loops to define complex and concurrent, composite services. The HTN facilitates human intervention, allowing a human to assist the composition process if necessary.

McIlraith *et al.* [MS02, MF02, NM02] used and extended Golog [LRL+97], a high level logic programming language built on top of Situation Calculus [MH69]. Golog composes services encoded in DAML-S. Users explicitly submit composition requests, which are expressed as generic ConGolog [GLL00] procedures. These templates are constructed using an off-line planning technique and are then modified based on user preferences and constraints. Generic templates are associated with a situation tree, which denotes a partial specification of the behaviour of the desired composite service. Each node in the situation tree denotes a snapshot of the desired service configuration at each point of its execution. This approach uses knowledge-gathering services to obtain the outcome of the service execution. The instantiated user specification is a sequence of primitive services which are then executed by a ConGolog interpreter. The user request is specified once before the composition, and during the execution of the composite services users have no control on the executed sequences of actions.

Ponnekatni *et al.* proposed SWORD [PF02], a toolkit for Web service composition. SWORD employs a rule-based expert system based on the Rete algorithm [For82]. This system automatically determines if a desired service can be realised as a composition of existing, predefined, services. Each service is described in terms of its conditional inputs and outputs, which are defined in a entity-relationship based domain model. A rule is then generated to define which outputs can be obtained given particular inputs. The main limitation of their approach is the lack of support for conditional effects of Web services.

Berardi *et al.* [BCGM03, Ber05] considered a Web service as a tree of all possible interactions with clients and develop E-Service Composer (ESC). They used these templates in Situation Calculus [MH69] to provide automated procedures for performing composition. This is one of the very few frameworks that includes interaction with the user directly in the composition process, by presenting a user as a Web service. Berardi takes a two layered approach in the design of the framework, separating the composition task between the *Abstraction*, *Synthesis Engine* and *Realization* modules. The Abstraction module uses the WSDL description of the service, together with behavioural descriptions described in Web Service Transition Language (WSTL) [BRSM03] to generate a finite state machine (FSM) describing the composition problem. The Synthesis Engine, the central composition module, takes the FSM to generate an abstract specification of the composite service, which is then instantiated by the Realization module.

Akkiraju *et al.*, [AVG+04] devised a two layered workflow composition architecture. This work focuses on abstract business process flow specification, where processes are described at a high-level using BPEL4WS semantically annotated with DAML-S. The specific service binding is left to run-time flow execution, which performs run-time discovery, composition, binding and execution. The core of this system is the *Generic Web Service Proxy*, which takes the semantic descriptions of the service requirements represented in DAML-S, the domain constraints and pointers to UDDI registries. During the execution of a high-level BPEL4WS specification the Generic Web Service Proxy is invoked at each node to locate suitable services and automatically bind and invoke the feasible sets. Akkriraju *et al.* treat the binding of abstract services as a planning problem, where the description of the abstract service serves as a goal definition. Their system employs Planner4J [Sri04], a Java planning infrastructure which embodies STRIPS-based planners.

Pistore *et al.* [PBB+04] presented a service composition framework, grounded in the concept of *Planning as Model Checking*, also known as Model Based Planning (MBP). Later they develop the ASTRO [TPC+05] toolset supports automated service composition, monitoring and execution. ASTRO allows for all levels of domain observability by employing sensing actions. ASTRO uses the EaGLe goal language [LPT02], which supports complex goals, specifying extended temporal conditions on user goals. ASTRO is implemented in a centralised manner and generates a composite service definition in BPEL4WS.

Table 2.3 summarises the comparison of planning based service composition frameworks. Not all of the identified features are present in a single architecture, as they focused on different research problems. All surveyed architectures employ a centralised composition model. They address dynamism in the composition, primarily from the perspective of unavailability of selected Web services, and deal with the issues of how to replace them with other equally capable Web services to perform the desired task.

2.3 Summary

This chapter reviewed the fields of context aware computing and service composition. It highlighted challenges facing legacy application models regarding taking context into account to adapt application behaviour.

A survey of related work in task driven computing, an approach grounded in service composition, identified a number of features essential for a dynamic and flexible model for building context aware applications, such as task inference. The main research challenges for service composition frameworks were identified including automated construction of explicit representations of user task goals, adaptation, and composition and execution failure tolerance.

The chapter analysed the shortcomings of existing systems against a list of characteristics that are desirable for automated, planning-based Web service composition. Existing service composition frameworks present composition as a one off effort. Furthermore, most composition frameworks do not fully handle composition failures, but focus on execution failures.

By contrast to existing service composition frameworks, this dissertation proposes a framework for context aware service composition based on facilitating composition, continuous monitoring, failure management and composition adaptation. The next chapter describes the design of the proposed framework.

	Service framework					
Feature	**Wu's**	**mcIliarth's**	**SWORD**	**ESC**	**Akkrijau's**	**ASTRO**
Composition method	SHOP2	ConGolog	Rete	Situation calculus	State planner	MBP
Service markup	OWL-S	OWL-S	XML	WSTL	OWL-S	BPEL4WS
Composition model	central	central	central	central	central	central
Extended goals	*	✓	×	*	✓	✓
Complex actions	✓	✓	×	✓	✓	✓
Dynamic composition	*	*	*	✓	*	✓
Re-composition	×	×	×	✓	×	×
User interaction	*	*	*	✓	*	*
Automatic service discovery	×	*	×	✓	✓	*
Non-determinism	*	*	×	*	*	✓
Implicit task specification	×	×	×	×	×	×
Resource constraints	*	*	×	×	*	*
Composition failure recovery	×	×	×	×	×	×
Execution failure recovery	✓	✓	×	✓	✓	✓

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

Table 2.3: Main research challenges and features of automatic Web service composition and which of these are met by the surveyed composition frameworks.

Chapter 3

Service composition framework

The review of related work, presented in the previous chapter, has identified a number of challenges for conventional architectures for the development and deployment of context aware applications to accommodate dynamic, context-rich computing environments. Furthermore, the analysis of existing service composition frameworks has identified a number of their shortcomings. Firstly, they do not provide adequate resilience to failures arising both at composition and execution times. Secondly, they do not allow for re-composition. Finally, they do not sufficiently accommodate multiple service composition methodologies.

To address these limitations, I propose a novel framework for context aware service composition. This approach is grounded in the process of dynamically locating, selecting, composing and coordinating atomic services, based on the current context of a user. Services are continuously recomposed as the context changes, enabling the automated development and adaptation of extensible and fault-tolerant context aware applications. Furthermore, the proposed framework embodies a composition failure management system, which attempts to reformulate composition requests into alternative ones that can be solved by the composition methodology in use.

This chapter describes the design of the service composition framework [VR04a, VR05a] in terms of the operations it provides. It presents all system components and their functionality together with the interfaces facilitating their interactions. Chapters 4 and 5 discuss the implementation and internal structure of the framework components.

3.1 Usage scenario

To illustrate how context aware applications can be built using the proposed framework, this section introduces the following scenario in the scope of an infotainment application domain. An infotainment is a term used for services combining context, most commonly location information, with entertainment applications, such as a point of interest facility.

A user, called Miles, subscribes to a mobile network provider, which hosts an infotainment portal. This portal offers users a broad array of resources and services, such as point of interest information, on-line shopping, and search engines. It is a single starting point for retrieving information from multiple, diverse sources.

To accommodate user requests, the portal employs a composition framework to coordinate atomic, disparate services. For example, some of the services in this scenario include:

- `RestaurantFinder`: provides a directory of restaurants.
- `DirectionsFinder`: computes the driving directions.
- `TranslationService`: translates the content from one language to another.
- `SpeechSynthesizer`: converts from text format to speech.

User requests are enriched with context information. For example, the infotainment portal takes into account the following context types: user location, user activity, and the computing device in use, provided by context middleware. Figure 3.1 describes the following two use cases.

Use case 1. In the first case, Miles is using his SmartPhone while walking around Market Square in Cambridge, UK. Miles has a subscription to an infotainment portal, available from his local mobile provider. This provides Miles access to a restaurant recommendation service, for example, to make lunch plans with his college friends. The portal then uses the composition framework, which assembles a composite service to deal with Miles' request. The resulting service, tailored to help Miles locate a Spanish restaurant, is composed from atomic services, such as `RestaurantFinder`, a UK-based restaurant directory, and `DirectionsFinder`, the navigation service.

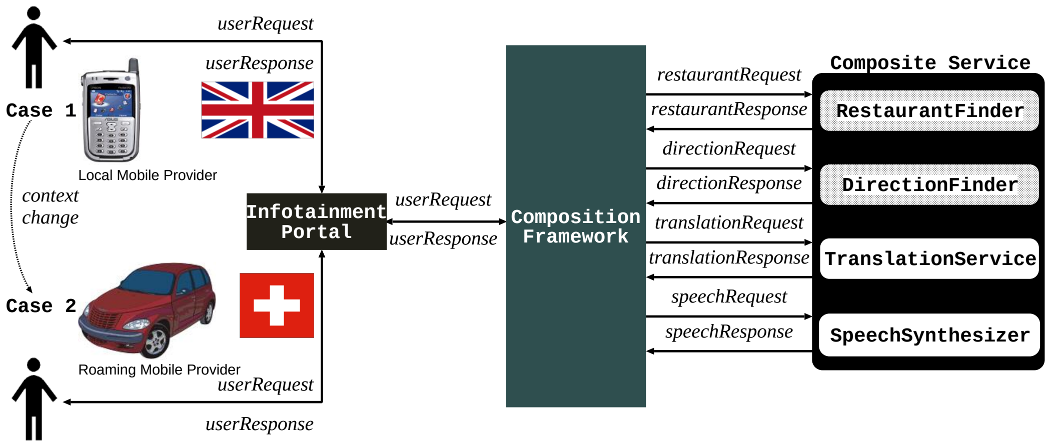


Figure 3.1: Usage scenario: context aware restaurant finder

Use case 2. Later in the day, Miles lands in Zurich. He wishes to catch up with his friends at a Lebanese restaurant. Miles is now registered with a roaming mobile network provider, which provides the local restaurant guide service; however it does so only in the German, French and Italian languages. Furthermore, this information is formatted for presentation on a mobile phone. As Miles is driving, he would prefer the restaurant directions to be routed to his in-vehicle information system (IVIS) and delivered in speech. A special new service for Miles will be assembled from atomic services, such as: `RestaurantFinder`, `DirectionsFinder`, the `TranslationService`, and `SpeechSynthesizer`.

In both cases, Miles has the same goal: he wishes to get directions to a chosen restaurant. 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*. By contrast, in the second case, Miles is *driving* through *Zurich*, and using *IVIS*. Because of the context changes, in the latter case, two additional services are required to fulfill Miles' request, namely the `TranslationService` and the `SpeechSynthesizer`.

To summarise, Miles submits both composition requests in the same way. He selects the desired task, context is automatically acquired, and the requested composition process is performed. However, the resulting application is different, because of the different contexts in which the two requests are submitted.

3.2 Design requirements

To enable the development of extensible context aware applications, the proposed composition architecture needs to be *general-purpose*, and must not embed any scenario specific dependencies. Furthermore, the system design principles of context aware service composition must hold independently of the particular infrastructure implementing the framework. To fulfill this requirement the design must allow for extensibility in terms of component technologies and composition methodologies used.

Context has a central role in this approach and implies a number of design requirements. Firstly, the framework must allow a system designer and a user to select what context types an application should take into account. The framework needs to acquire context from *different context sources* in a disciplined way. A mechanism is also needed to describe what *action* to take when the user enters a certain context.

Another important challenge in providing a context aware service composition facility is dealing with failures. For instance, failures may occur at composition time, as a result of context changes and missing service descriptions. Failures may also arise at run-time, for example, because of the loss of network connectivity. The framework must be resilient to both types of failures.

When deployed in production environments the framework will be exposed to a large number of concurrent composition requests. The design of the framework must ensure its ability to operate under increasing load, increasing complexity of requests and increasing size of resulting composite services.

From these requirements I identify a number of required framework operations, such as specification and construction of context aware composition requests, service composition, execution of composite services, and failure-recovery.

3.3 Framework architecture

This section introduces the design of the proposed framework, building on the requirements set out in the previous section. It describes the main components of the system's architecture using the scenario described in Section 3.1. It presents the functionality that each component delivers, and the interactions between them to facilitate fault-tolerant, context aware service composition.

Figure 3.2 shows an overview of the system architecture, which employs a *layered approach* to service composition, to fulfill the design requirements outlined in the previous section. The four layers in the system architecture map to the four main stages in the service composition process. The first layer is

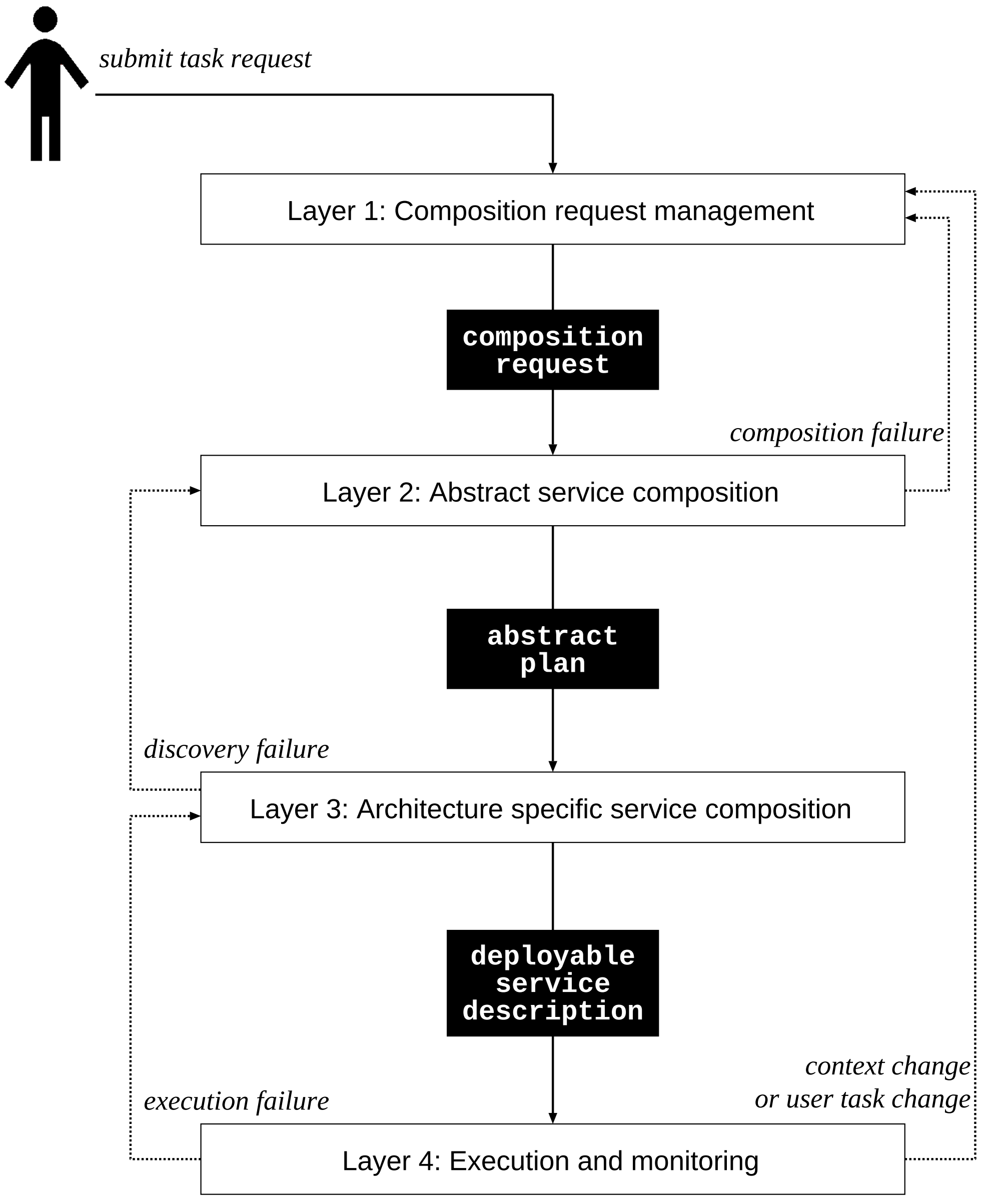

submit task request
Layer 1: Composition request management
composition request
composition failure
Layer 2: Abstract service composition
abstract plan
discovery failure
Layer 3: Architecture specific service composition
deployable service description
execution failure
context change or user task change
Layer 4: Execution and monitoring

Figure 3.2: Overview of the proposed service composition architecture

the *composition request management layer*, which assembles and, if necessary, modifies a `composition request`. Each `composition request` is a formal definition of the user's task intention. The next layer is the *abstract service composition layer*, which generates an `abstract plan`. An `abstract plan` is a set of abstract services and their control flow, comprising the composite service. Abstract services are high-level descriptions of service operations and cannot be directly invoked. The *architecture specific service composition layer* instantiates the `abstract plan` and generates a `deployable service description`, which represents a service instance. The `deployable service description` is passed to the *execution and monitoring layer*, which invokes the specified service instance and monitors its execution.

Designing the system in layers, separating the functions required by each stage in the composition process, presents several advantages. Firstly, this system design together with internal representations, such as the `composition request` and `abstract plan`, allows the framework to use multiple component technology, composition methodologies and run-time environments. Secondly, it also aids the isolation of failures, which are passed to the appropriate layers to be dealt with.

3.3.1 Composition request management layer

Figure 3.3 shows the structure of the first layer of the composition framework. The `composition request` is an entry point to the composition process. It specifies the user's task and consists of two parts. The first part is a description of the core user task, for example, Miles' request in Case 2 of the usage scenario for directions to the nearest Lebanese restaurant, selected from the `Goal Service` (Step 1 – Figure 3.3). The second part contains contextual parameters. For example, if Miles is using an IVIS, this would specify a computing device. He is currently *driving* down Limmatstrasse in *Zurich*, which are context types activity and location obtained from the `Context Service` (Step 2). Such contextual parameters further customise the composition request. For instance, in this context, it may be more appropriate to read out the driving directions to Miles. This layer constructs the `composition request` and feeds it to the abstract service composition layer (Step 3).

If the abstract service composition process fails (Step 4a) control is passed back to the composition request management layer, which attempts to transform the `composition request` into an alternative request that can be satisfied. For example, Miles' original `composition request` to find the nearest Lebanese restaurant may be replaced by a more generic request of finding any type of restaurant nearby. Furthermore, the requirement to present the output in speech format may be removed, if the speech synthesiser service is missing.

In addition, the `composition request` may be transformed in order to improve the user experience, and not only when a failure occurs. For example, when reading out the driving directions to Miles, the system could automatically lower the music volume, despite the fact that this was not explicitly set as a part of Miles' goal. Chapter 4 provides more details about the implementation of the internal mechanisms used in each of the components in this layer.

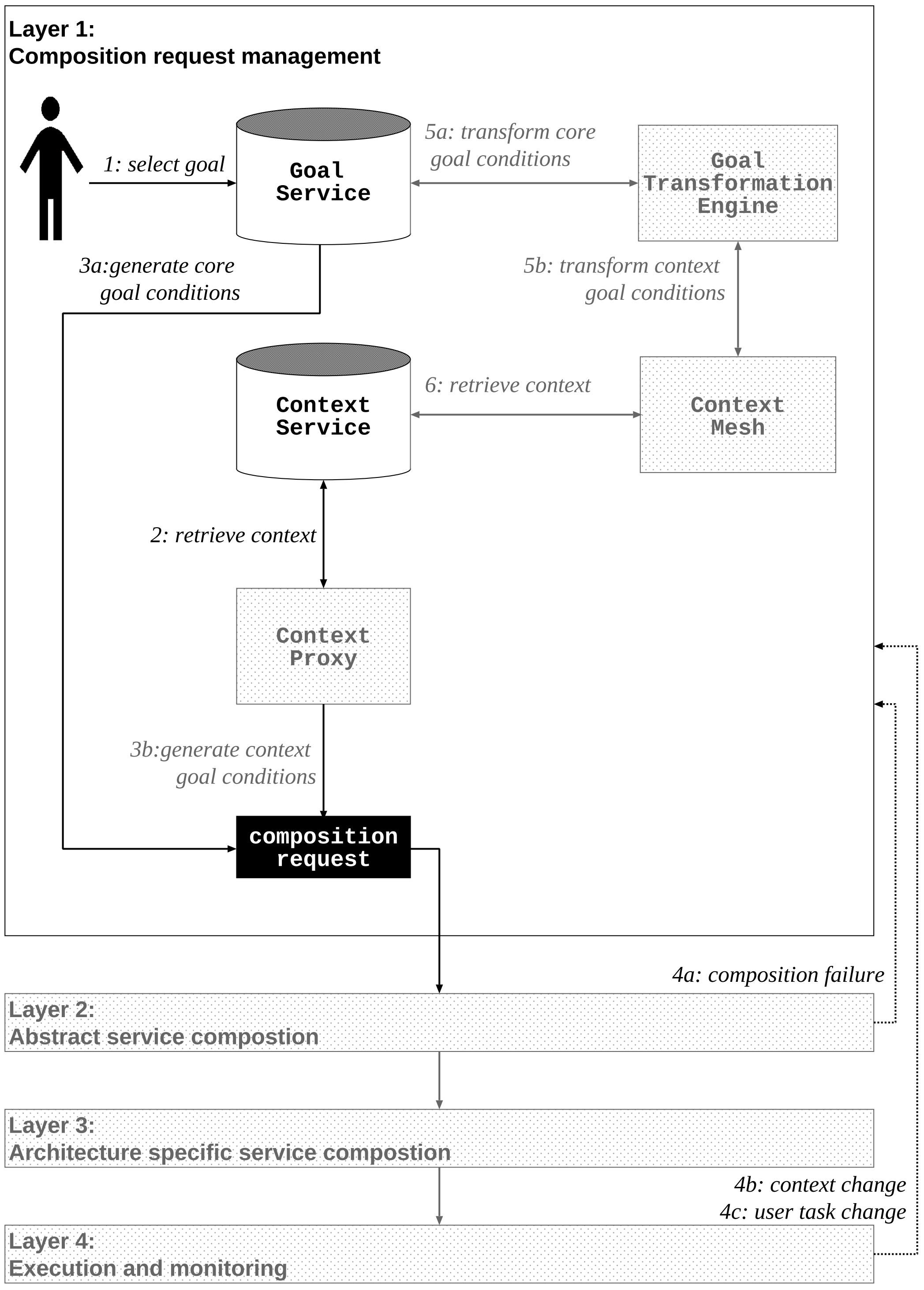

Layer 1:
Composition request management
1: select goal
Goal Service
5a: transform core goal conditions
Goal Transformation Engine
3a:generate core goal conditions
5b: transform context goal conditions
Context Service
6: retrieve context
Context Mesh
2: retrieve context
Context Proxy
3b:generate context goal conditions
composition request
4a: composition failure
Layer 2:
Abstract service compostion
Layer 3:
Architecture specific service compostion
4b: context change
4c: user task change
Layer 4:
Execution and monitoring

Figure 3.3: Composition request management layer

3.3.2 Abstract service composition layer

The service composition process is split into two stages: *abstract* and *architecture specific*. Abstract service composition is the process of assembling abstract services, which are generic operations each satisfying different parts of the overall `composition request`. Architecture specific composition layer instantiates these abstract services and constructs an executable composite service.

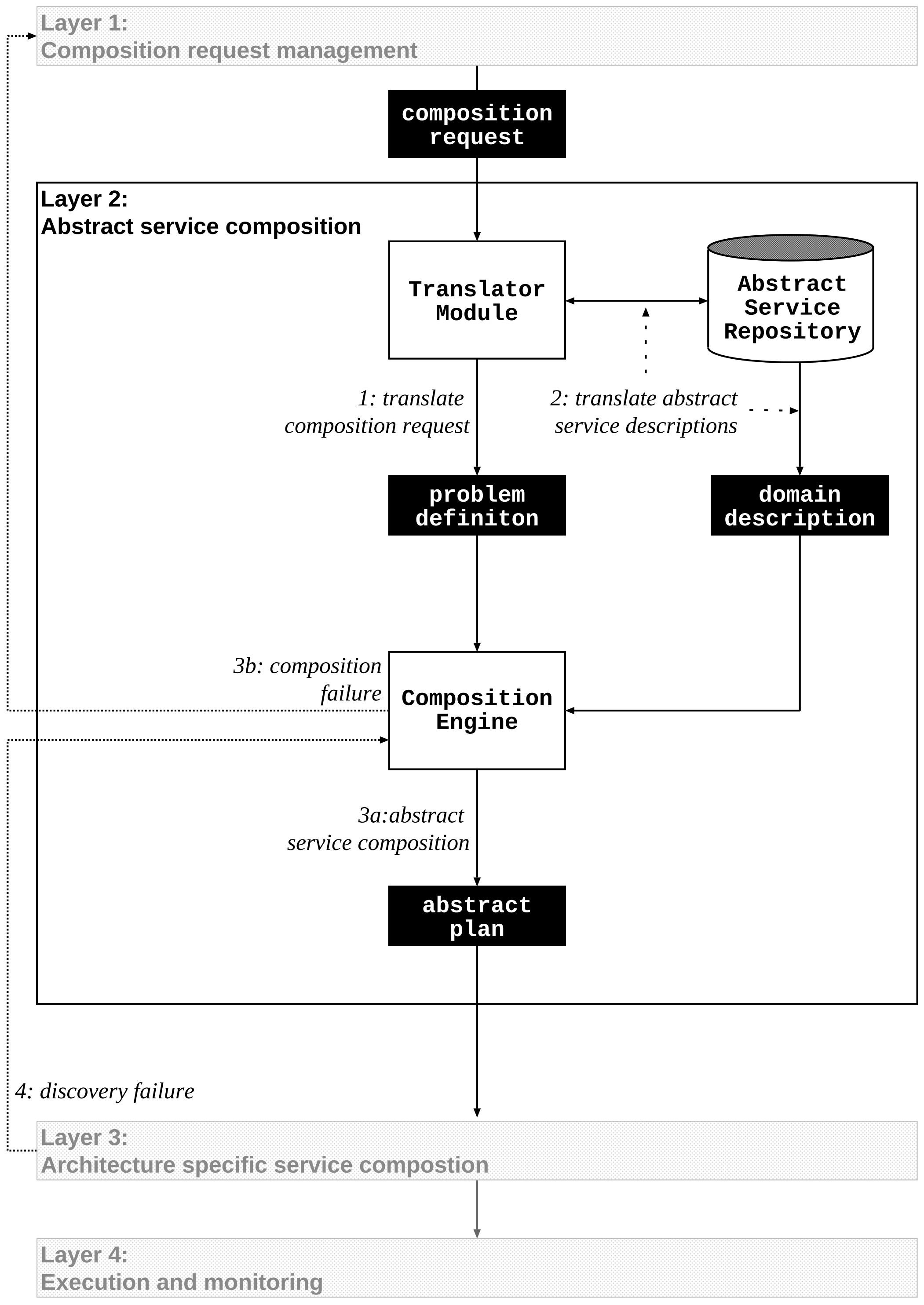
This two-layered approach has been introduced for a number of reasons. Firstly, this approach enables the framework to be implemented using any type of composition methodology, component technology and run-time environment. Secondly, it facilitates recovery from service discovery and service execution failures, by isolating the different stages in the composition process. Finally, it enhances the scalability of the framework, as abstract service composition is performed only on a subset of abstract services, rather than all available service instances. Chapter 6 analyses this in detail.

Figure 3.4 shows how the abstract service composition generates an `abstract plan`, which defines the control flow of abstract services. Firstly, the `Translator Module` converts the `composition request` to a `problem definition`, which is in the representation format supported by the composition methodology in use (Step 1 in Figure 3.4).

The `Abstract Service Repository` stores and manages abstract service descriptions. In our usage scenario the sample abstract services provided include a restaurant directory service and a speech synthesiser service. Abstract services are semantically annotated, their descriptions contain the types of parameters they expect, as well as preconditions and expected postconditions for their successful execution. Each abstract service also points to the files carrying the descriptions of the domain concepts used, such as a definition of restaurant in our usage scenario. The `Translator Module` converts the available abstract service descriptions and domain concepts from the `Abstract Service Repository` to generate the `domain description`, in the representation format supported by the composition methodology used (Step 2 in Figure 3.4).

The `Composition Engine` uses the `problem definition` (Step 1) and the `domain description` (Step 2) to generate the `abstract plan` (Step 3a), which consists of a list of abstract services to be executed, described in the composition language. It is then stored in the internal representation format which is independent of the composition methodology. Finally, the `abstract plan` is fed to the architecture specific service composition layer for instantiation.

If the system fails to create an `abstract plan` (Step 3b) control is passed back to the composition request management layer, where the `composition request`



Layer 1:
Composition request management
composition request
Layer 2:
Abstract service composition
Translator Module
Abstract Service Repository
1: translate composition request
2: translate abstract service descriptions
problem definiton
domain description
3b: composition failure
Composition Engine
3a:abstract service composition
abstract plan
4: discovery failure
Layer 3:
Architecture specific service compostion
Layer 4:
Execution and monitoring

Figure 3.4: Abstract service composition layer

is transformed into one that may also be satisfiable. If in the architecture specific composition layer (layer 3) the process of service discovery and instantiation fails, control is passed back to the abstract service composition layer, which initiates a recomposition process (Step 4).

The `Composition Engine` may be implemented by a number of different composition methodologies, as described in Section 2.2.5. For example, AI Planning has proven to be a valuable and effective tool for service composition [MF02, WSH$^+$03]. Abstract services can be represented in terms of their non-functional and functional properties. Non-functional properties describe service provider details and Quality of Service parameters. Functional properties contain descriptions of service operations in terms of inputs, outputs, preconditions and effects, which makes it easy to convert them into planning actions. The `Translation Module` converts a `composition request` into `problem definition` and abstract service descriptions in the `domain description`, which are formats supported by the planner. Chapter 5 describes in more detail how the framework employs AI planning to handle the service composition problem in an efficient and scalable way.

3.3.3 Architecture specific service composition layer

Figure 3.5 shows the system components involved in the process of architecture specific service composition and their interactions.

The `Plan Translator` (Step 1 — Figure 3.5) converts the `abstract plan` into an `abstract execution plan`, which describes a composite service in architecture specific format. As the framework stores the `abstract plan` in an internal representation format, it is necessary to have translation mechanisms for different run-time technologies used and their corresponding representation formats. The `abstract execution plan` describes each service in terms of its parameters, expected preconditions and postconditions, and any other semantic tags such service categorisation codes, as well as Quality of Service parameters.

The `Plan Instantiator` executes the `abstract execution plan` and mediates the process of service discovery and instantiation. The `Service Registry` allows service providers to submit descriptions including their identifiers, name, interfaces provided, and time-to-live information. It exports interfaces for service discovery and publishing. It performs service discovery and returns the `service binding` information for each service instance.

The `Plan Instantiator` processes the `abstract execution plan` and passes the information about abstract services and the required Quality of Service parameters to the `Service Registry` (Step 2). For example, the `abstract`

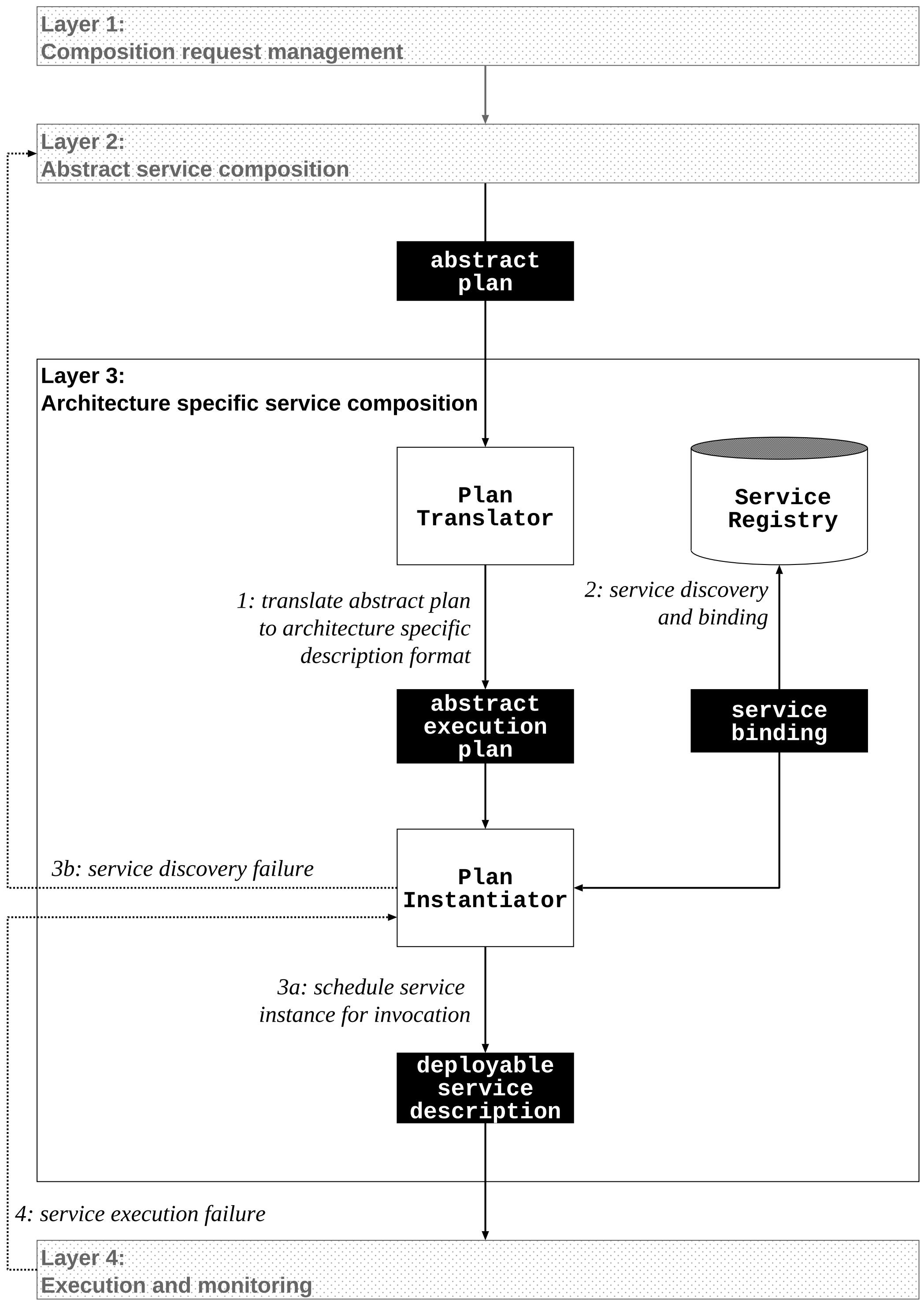

Layer 1:
Composition request management
Layer 2:
Abstract service composition
abstract plan
Layer 3:
Architecture specific service composition
Plan Translator
Service Registry
1: translate abstract plan to architecture specific description format
2: service discovery and binding
abstract execution plan
service binding
3b: service discovery failure
Plan Instantiator
3a: schedule service instance for invocation
deployable service description
4: service execution failure
Layer 4:
Execution and monitoring

Figure 3.5: Architecture specific service composition layer

execution plan may contain an abstract service representing a restaurant directory. Following the discovery process, this abstract service may be instantiated by, for example, Zagat's [ZZ99] restaurant directory service. Once the abstract service is instantiated the Service Registry returns its service binding. Plan Instantiator uses this service binding as a basis for the deployable service description (Step 3a) and passes it to the execution and monitoring layer, which schedules its invocation.

If service discovery fails (Step 3b) control is passed to the abstract service composition layer, which triggers recomposition. However, if the service fails during execution, control is passed back to architecture specific composition layer, where a replacement service is fetched (Step 4).

3.3.4 Execution and monitoring layer

The Execution Engine provides the run-time environment in which services can be executed. It invokes scheduled services as specified in the deployable service description (Step 1 — Figure 3.6).

The Monitoring Engine is bound to the Execution Engine to track changes in the run-time environment, service performance and composition request status. The Monitoring Engine verifies the service preconditions before being invoked by the Execution Engine. During the service lifetime it observes changes in the environment and propagates any failures to the upper layers in the framework, where they are dealt with. Finally, once the service completes its operation the Monitoring Engine verifies service effects, against the expected outcomes.

Service execution may fail due to network disconnection. If a service instance cannot be invoked the system tries to execute a replacement service, if one has been previously cached. Pointers to replacement services may be included in the deployable service description. If the cached service fails as well, control is passed to the architecture specific composition layer, which replaces it with a suitable service of the same type (Step 3a). If this operation fails too, the system continues propagating the failure up the layered framework structure.

Should an unanticipated change in context occur (Steps 3b), or should the user change the task specification (Step 3c), control is passed to the composition request management layer, where a new composition request is generated and recomposition triggered.

The Monitoring Engine updates the state of the Composition Engine and the Execution Engine. There are several different events that may take place during the service execution. For example, the actual outcome of the service may not be as anticipated or the primary aim of the service may be unexpectedly

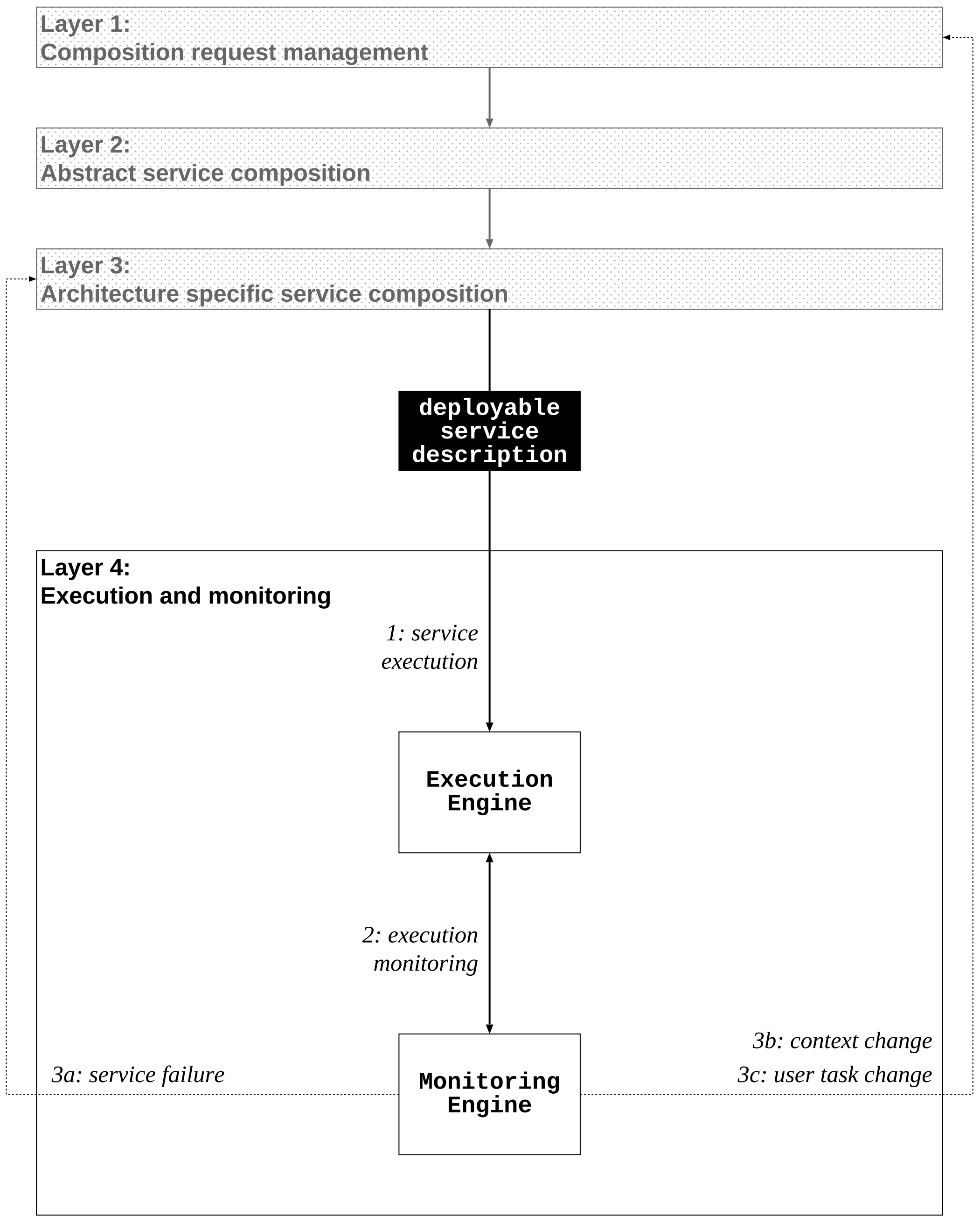


Figure 3.6: Execution and monitoring layer

satisfied by another service. For instance, the expected outcome of the scheduled service is to automatically lower the volume of the in-vehicle stereo. Before the service is executed, the user manually adjusts the stereo volume and therefore achieves the outcome of the scheduled service. In such cases the `Monitoring Engine` adds the information to the state description in the `Execution Engine`, which ensures it does not trigger the service execution. Finally, if a required service precondition is no longer true the service will not be invoked.

To observe context changes and service execution the `Monitoring Engine` employs `monitoring procedures` proposed by Haigh *et al.* [HV96]. The next section describes in detail the fault recovery mechanisms employed by the composition framework.

3.4 Failure tolerance

A number of different failures may occur during the service composition process, including `composition request` failures, context middleware failures, network level disconnections, service discovery failures and service execution failures. Failures can be classified into three groups based on the stage of the system operation in which they occur. *Composition* failures arise during the assembly and instantiation of the composite service, on the first two layers of the framework. Inability to instantiate abstract services results in *discovery* failures. *Execution* failures occur at run-time, during the operation of the composite service in the execution and monitoring layer.

This section describes how the system design achieves resilience towards composition, discovery and execution failures. It also presents how the system applies appropriate fault control mechanisms to enhance the probability of successful processing of `composition requests`.

3.4.1 Composition failures

Two types of composition failures may occur. Firstly, the system may fail to assemble a `composition request`. Secondly, the system may fail to successfully produce a composite service for a given `composition request`.

Composition request assembly failure. Several context failures may occur, impairing the process of `composition request` construction. One problem is that the accuracy and reliability of the context data varies. To facilitate a high level of confidence about the accuracy of context information, the current implementation of the `Context Service` augments a Quality of Information model, as described in Lei *et al.* [LSD$^+$02].

Network disconnection and sensor failure may cause context providers to become unavailable. The system deals with this type of failure in a number of ways. Firstly, it attempts to locate a different provider for the context type for which there is no availability of context information.

If no providers are available for the context type in question the system acquires historical contextual data from the `Context Service`. It may also construct a `composition request` based on a previously executed task.

In case all automated means of context acquisition and inference fail, the user is prompted to supply the context value manually either by selecting from the most common values previously used, or by entering a new value. As a last resort, the `composition request` will not include the contextual condition for which there is no information.

Finally, as part of future work, I anticipate devising an algorithm for inferring the missing context values by correlating the existing context information.

Composition request failure. If a `composition request` cannot be satisfied, due to a missing service composition or unanticipated context, the abstract service composition layer passes the `composition request` back to the composition request management layer. There the composition failure management system applies goal transformations to modify the `composition request` into one that may be solved by the `Composition Engine`. The implementation of this composition failure management system is discussed in detail in Chapter 4.

3.4.2 Discovery failures

Service discovery failures may occur because of an inaccessible `Service Registry`, as a result of network disconnection or system overload. Furthermore, the requested service simply may not be available in the registry.

To avoid the `Service Registry` being a single point of failure, the system has been designed to operate with multiple service registries. This can be achieved both by deploying a number of replicas of the same registry, as well as by mediating among a number of different registries.

A discovery failure may be caused if a suitable service instance cannot be found through the discovery process, or if a service operation no longer matches its interface. This may be a result of changes in the service functionality, and can be addressed by using cached services, which were previously invoked, if they are still available. To facilitate this approach, the system keeps its own log of services executed, together with the context in which they were invoked and the specification of the original `composition request`.

For example, if the directions service in English language is not available, the system may opt to use a German one together with a translation service to English, and then add a speech synthesiser service. This invokes the composition of a new sub-request in the abstract service composition layer. If that also fails control is passed to the composition request management layer, where the `composition request` is transformed into one that may be solved.

3.4.3 Execution failures

Unexpected environment changes. Composite services may be executed in an environment, which is different from the one where composition occurred. Therefore some assumptions about context values may no longer be valid at the time of execution. Furthermore, there may be multiple, often unexpected, service outcomes.

The `Monitoring Engine` is a component that observes context changes during the execution of the composite service. It employs `monitoring procedures`, proposed by Haigh *et al.* [HV96], which encapsulate the following data:

- `type:` A parameter that determines whether a monitoring procedure is observing an event or a service.
- `precondition_state:` A service precondition state that must be fulfilled prior to execution.
- `expected_state:` For service monitoring this represents a service postcondition. Otherwise it denotes a definition of the event to be monitored.
- `recovery_procedure:` A standard method for reacting to events of the above types of failure.

A number of different events may occur during the execution of composite services. These environment changes 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. In another example, following the composition process the user's location changes. As a result the previously generated driving directions are no longer valid and the translation service should not proceed with their translation.

The `Monitoring Engine` observes context changes using event monitors, which are invoked when relevant goal conditions are introduced or removed. It then updates the state of the `Composition Engine` and the `Execution Engine`.

Section 5.3.2 provides more details about the implementation details of internal mechanisms used to facilitate event monitoring.

Service execution failure. Before a service is invoked, the `Monitoring Engine` determines if all its preconditions are satisfied. After the execution of the service, the `Monitoring Engine` verifies its postconditions, as there may be multiple outcomes or the service may fail unexpectedly.

If the service fails the `Monitoring Engine` invokes recovery mechanisms, as specified in the `monitoring procedure`. If the attempted recovery method fails, control is passed to the abstract composition layer, together with the current context in which a recomposition may occur. Finally, if the recomposition process fails as well, control is passed back to the composition request management layer, in which a `composition request` is modified or a new one assembled.

Service unavailability. Services may become unavailable as a result of network disconnection. This type of failure is handled in two ways: locally, by each service instance and globally, with respect to the composite service. To deal with service unavailability, the framework employs the approach proposed by Gu *et al.* [GNY04]. They implement two methods to facilitate fast localised service recovery: *proactive* and *reactive*. The proactive approach keeps a number of cached copies to avoid the delays inherent in re-discovering a suitable service instance. In the absence of a proactive approach the system invokes a reactive approach, where control is passed back to the architecture specific composition layer, in which a new service is fetched from the `Service Registry`.

To reduce the probability of losing service availability and to meet Quality of Service requirements, the system may also opt to run redundant instances of composite services to ensure that the overall execution will not be drastically impaired by delays caused by the failure of one or more atomic services. This facilitates fast failure recovery and reduced composition overhead, resulting in a more resilient system. This mechanism will be investigated as part of the future work described in Chapter 7.

Chapters 4 and 5 describe further implementation details of the fault-recovery mechanisms.

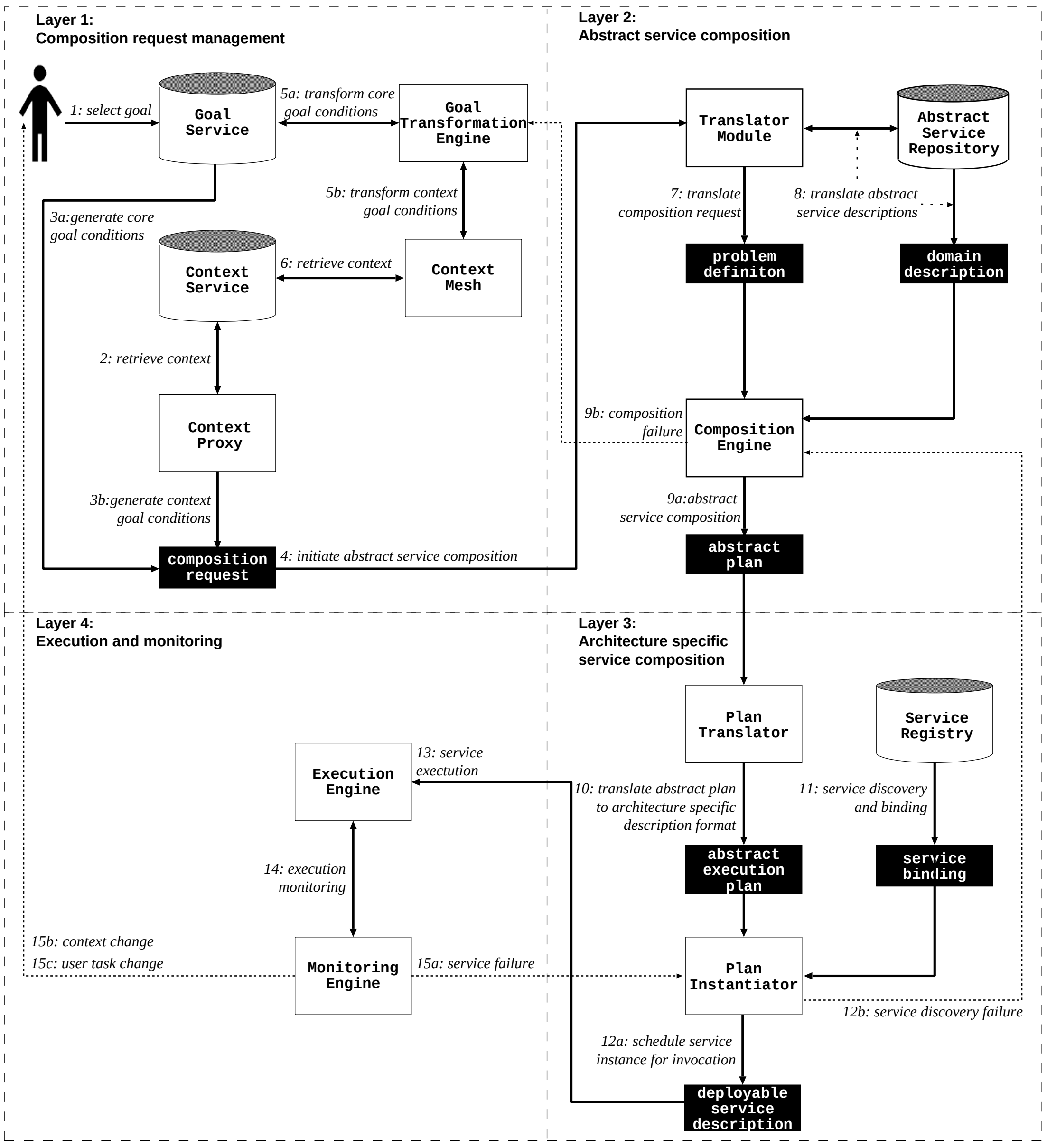
3.5 Summary

This chapter has described the architecture of the proposed framework for flexible and failure-tolerant context aware service composition. The service composition framework employs a layered design approach to separate the following four stages of the composition process: composition request management, abstract service composition, architecture specific service composition and execution and monitoring. Figure 3.7 shows an overview of the layered system design in terms of its components and their interactions.

This chapter has analysed the operations that the framework supports by means of a usage scenario in the infotainment domain. Users select the computational task, which, together with the context, forms the `composition request`. The `Composition Engine` locates, selects and composes abstract services that meet the constraints of this composition request to construct an `abstract plan`. The `Plan Translator` converts an `abstract plan` into an `abstract execution plan`. It instantiates abstract services by querying available service registries. Instantiated services are then deployed by the `Execution Engine` and their behaviour is observed by the `Monitoring Engine`. Changes in the environment and service performance are handled by `monitoring procedures`.

The framework does not rely on a specific composition methodology. It has been designed for openness and extensibility in terms of allowing multiple implementations of the `Composition Engine` and component technologies to coexist.

One of the main challenges in developing context aware service composition facility is being able to handle composition failures. The service composition process may not be able to successfully assemble a composite service, because of missing service descriptions or unexpected context. The next chapter analyses this problem in more detail and describes the implementation of a comprehensive composition failure management solution.



Layer 1:
Composition request management
Layer 2:
Abstract service composition
1: select goal
Goal Service
5a: transform core goal conditions
Goal Transformation Engine
Translator Module
Abstract Service Repository
5b: transform context goal conditions
3a:generate core goal conditions
7: translate composition request
8: translate abstract service descriptions
Context Service
6: retrieve context
Context Mesh
problem definiton
domain description
2: retrieve context
Context Proxy
9b: composition failure
Composition Engine
3b:generate context goal conditions
9a:abstract service composition
composition request
4: initiate abstract service composition
abstract plan
Layer 4:
Execution and monitoring
Layer 3:
Architecture specific service composition
Plan Translator
Service Registry
Execution Engine
13: service exectution
10: translate abstract plan to architecture specific description format
11: service discovery and binding
abstract execution plan
service binding
14: execution monitoring
15b: context change
15c: user task change
Monitoring Engine
15a: service failure
Plan Instantiator
12b: service discovery failure
12a: schedule service instance for invocation
deployable service description

Figure 3.7: System architecture overview

Chapter 4

Composition failure management

Providing context aware service composition facility encompasses research problems related to handling composition failures and specifying context behaviour. The service composition process may terminate unsuccessfully, without assembling a composite service as expected, if, for example abstract service descriptions are missing. Another challenge is constructing context commands, which describe the desired application behaviour, while preserving application's extensibility. This chapter describes `GoalMorph`, a system that can be used to tackle these two problems.

The prototype implementation of the proposed framework for context aware service composition employs planning technology to assemble composite services. The `composition request`, which is a complex structure consisting of a number of *goal conditions*, forms what is termed the `problem definition` in planning technology. Each goal condition is a specific requirement that must be met. For example, Miles' request for driving directions to the nearest restaurant includes specifying that the desired type of restaurant must be found, gathering the address details of the selected restaurant and calculating and presenting driving directions to navigate Miles from his current location to the selected restaurant.

The success criterion for a planning process is whether all goal conditions have been satisfied. Often, because of a context change or a missing service description, planning can fail to fulfill some of the goal conditions. As a result the problem defined by the `composition request` is not solved. Because each goal condition represents a partial solution to the `composition request`, satisfying some goal conditions instead of all can be more useful than satisfying none of the goal conditions at all. This chapter presents *GoalMorph* [VR05b], a composition failure management system that applies context aware goal transformations to failed `composition requests` to convert them into ones that can potentially be solved by the AI planner.

As part of the `GoalMorph` system, this chapter also describes `cogotags`, a mechanism for automated construction of context aware goals. The `Composition Engine` deals with formally defined `composition requests`. The formal definition of a `composition request` varies as a result of different contexts, such as resources available, time constraints, and user location. A common approach in service composition is the use of goal repositories, which store pre-defined, formalised goal descriptions. As the number of context types and their values grows this method becomes impractical. It is also not always possible to anticipate all situations in which the user may submit a request.

The next section motivates the work described in this chapter. Section 4.2 analyses the proposed goal taxonomy and presents the architecture of `GoalMorph` in terms of the functions of each component and its interfaces. Section 4.3 discusses evaluation results and demonstrates that the implementation of `GoalMorph` provides a practical and scalable solution. Finally, Section 4.4 compares `GoalMorph` to GTrans [CZ04], a goal transformation framework, and positions `GoalMorph` in the research context of partial goal satisfaction.

4.1 Motivation

Two driving factors behind the work presented in this chapter are (a) the need for supporting partial satisfaction of composition requests and (b) the requirement for flexible construction of context goal conditions.

4.1.1 Partial satisfaction of composition requests

Tables 4.1 and 4.2 describe the sample context input and expected application behaviour for three different cases of the usage scenario, introduced in Section 3.1.

Let us consider Case 2 of the scenario, where Miles requests driving directions to the nearest Lebanese restaurant in Zurich. Because he is driving, Miles expects the directions to be read out to him, using the in-vehicle information system (IVIS). The planning process for this `composition request` may fail for a number of reasons. As one example, the planner may have the wrong descriptions of goal conditions or incomplete knowledge of the domain. In another example, some of the required services may be unavailable; there may not be any speech synthesiser facility in this in-vehicle system or any other instance available from the present computing resources.

Rather than terminating the composition process altogether, it may be possible to partly satisfy the `composition request`. The system could acquire further information about Miles' context, such as his social setting and presence and

	Input: Context data			
Case	**Activity**	**Time**	**Device used**	**Location**
1	Walking	12.30pm	SmartPhone	Market Square, Cambridge
2	Driving	7pm	IVIS	In-vehicle, Limmatstrasse, Zurich

Table 4.1: Sample context input in the usage scenario

	Output: Expected behaviour	
Case	**Presentation**	**Translation**
1	Text form	n/a
2	Speech form	Translate to English

Table 4.2: Sample application behaviour for context in Table 4.1

availability of other computing devices. For example, the system finds out that Stephanie is also in the car with Miles, and that her mobile phone and Miles' PDA are also available for use in the vehicle. By taking these facts into account, the system could then forward turn-by-turn directions to Stephanie's mobile phone, and have her guide Miles through Zurich. This removes the speech synthesis requirements from the original `composition request`, and the modified version of the `composition request` can then be solved by the planner.

4.1.2 Automated construction of composition requests

In all cases Miles' task intention is to obtain directions to a nearby restaurant. However, description of Miles' task intention maps to a different system `composition request` in different contexts.

In the proposed framework goal conditions are represented in Planning Domain Description Language (PDDL). This allows for easy import of goals into the software component that stores them. As a result the framework can be used with a large variety of PDDL-based planners. Each goal condition is a predicate or function specification in first-order-logic. There are two parts to the predicate definition: name and parameters. Figure 4.1 shows the formalised description of the `composition requests` in planning language, for Cases 1 and 2.

Figure 4.1(a) shows the formal representation of the `composition request` for Case 1, which contains goal conditions [1] such as `(restaurant_booking_made 3 1300)` and `(directions_found current_address restaurant_address)`. The first goal literal `(restaurant_booking_made 3 1300)` indicates that the restau-

[1]Operator definitions include preconditions that place additional constraints on the variables themselves, e.g. there are additional conditions on each operation to ensure that the booking is made for the same restaurant as the directions.

rant booking should be made for three people and for 13h. The second goal literal specifies that the directions should be found, navigating the user from current location to the restaurant location.

Figure 4.1(b) shows the goal condition `(directions speech_out)`, which is added to the `composition request` in Case 2 of the usage scenario. This is because when Miles is driving, directions should be in speech-synthesised form, as opposed to text form when Miles is walking on the street.

In both cases Miles wanted to obtain the driving directions to the nearest restaurant, however of different types. The same task intention results in different formal definitions of `composition requests` because of the difference of context in which it is invoked. Formally describing a user's `composition request` is an open challenge for two reasons. Firstly, pre-defining and storing formalised `composition requests` becomes unsuitable as the number of context types and their values grows. Secondly, it is not always possible to foresee all the contexts of the user. Therefore, a mechanism for flexible, dynamic assembly of context aware `composition requests` is essential.

4.2 GoalMorph: composition failure management using context aware goal transformations

This section introduces the goal taxonomy, describes the main components of the `GoalMorph` system and presents the operations they provide.

4.2.1 Goal taxonomy

Goal conditions that form the `composition request` may result from the task intention of the user or from the context of the user. For example, the goal `(directions_found current_address restaurant_address)` indicates a user's desired task. In contrast, the `(directions speech_out)` goal condition is triggered when the user is driving. `GoalMorph` separates goals into intention-driven *core* and context-driven *context* goals. The following is the resulting taxonomy of goal conditions:

Core goal. Any goal condition that purely describes a user's task intention, independent of the current context, is a core goal. In Case 2 of the usage scenario, shown in Figure 4.1(b), examples of core goals are the `(restaurant_found)` and `(direction_found)` goal conditions.

```
; Initial world
    (define (initial_state_Case_1)
        (cuisine spanish)
        (location cambridge)
        ...
        (persons 3)
        (time 1300)
        (activity walking)
    )

; Goal world
    (define (goal_state_Case_1)
        (restaurant_found spanish cambridge)
        (restaurant_booking_made 3 1300)
        (restaurant_booked restaurant_name)
        (directions_found current_address restaurant_address)
    )
```

(a) Formal description of composition request for Case 1

```
; Initial world
    (define (initial_state_Case_2)
        (cuisine lebanese)
        (location zurich)
        ...
        (persons 2)
        (time 2000)
        (activity driving)
    )

; Goal world
    (define (goal_state_Case_2)
        (restaurant_found lebanese zurich)
        (restaurant_booking_made 2 2000)
        (restaurant_booked restaurant_name)
        (directions_found current_address restaurant_address)
        (directions speech_out)
        (directions language english)
    )
```

(b) Formal description of composition request for Case 2

Figure 4.1: Formalised composition requests for Cases 1 and 2 of the usage scenario

Base core goal. The absolute minimal core goal condition that needs to be satisfied to achieve a viable solution for a given `composition request` is termed a base core goal. It can not be removed from the `composition request`. For example, in the usage scenario in which Miles requests directions to the restaurant, the base core goal is to find a restaurant. To fulfill the `composition request` and supply the user with a feasible solution, this base goal or its respective transformation must be satisfied.

Dependent context goal. A context goal condition that can be seen as an attribute of a core goal condition or directly related to it, is a dependent context goal. For example, the goal literal `(directions speech_out)` relies on the presence of the goal literal `(directions_found)`. If the core goal is removed from the goal set, any related dependent context goals are also removed. For example, the removal of the core goal `(direction_found)` implies removal of the dependent context goal `(directions speech_out)`.

Independent context goal. A context goal condition that does not necessarily directly affect the user's request is considered to be an independent context goal. For example, in the usage scenario it may be useful to add the goal condition of lowering the volume level of an in-car audio system while reading out the driving directions.
Classification of goal conditions according to the goal taxonomy is user and application specific. A qualified domain engineer specifies a set of core goals that describe the user's task intention. For example, in our restaurant finder scenario, the core goals include `(restaurant_found)` and `(direction_found)`. These task descriptions can further be customised by context goals, such as `(directions speech_out)`, which are often user specific, and are generated based on context information acquired from the context middleware. Furthermore, the classification of goal conditions can be extended to take the context of the user into an account, for example where directions are essential for a user in a foreign environment, compared to a user in a local environment.

The next section introduces the architecture of the composition request management layer and describes how core and base core goals are stored in the `Goal Service`, and context goals are generated by the `Context Proxy`.

4.2.2 GoalMorph overview

Figure 4.2 shows the architecture of `GoalMorph`. The entry point in the `GoalMorph` system is a `composition request`. The user selects the task from the `Goal`

`Service` (Step 1 in Figure 4.2), which contains the core goal conditions for this `composition request`. For example, in the usage scenario, Miles requests driving directions to the nearest restaurant. Once Miles selects the task, the `Goal Service` returns a list of corresponding core goal conditions for the selected task (Step 3a).

The `Context Proxy` generates context goal conditions that customise the `composition request` based on a user context, which is provided by the `Context Service` (Step 2). To implement the `Context Service`, `GoalMorph` uses the context middleware solution proposed by Lei *et al.* [LSD+02]. The `Context Service` retrieves context information, such as the user's location, the device in use and the user's activity. This specific implementation of the `Context Service` supports both push and pull models for provisioning of context information. It provides access control mechanisms to protect user data. Furthermore it contains a Quality of Information model, which is used to reason about the accuracy and freshness of context. The final `composition request` is assembled from the core goal conditions from the `Goal Service` (Step 3a) and the context goal conditions (Step 3b) from the `Context Proxy`, which are assembled based on the information provided by the `Context Service` (Step 2).

If the abstract service composition layer fails to assemble a composite service given a `composition request` it passes control to the `Goal Transformation Engine` (Step 4a in Figure 4.2). This component reformulates the `composition request` into a problem that can be solved by reformulating core (Step 5a) and context (Step 5b) goal conditions. The transformed `composition request` is then fed back to the planner and the composition process resumes.

4.2.3 Context Proxy: automated construction of context goals

Context **go**al **tags**, termed `cogotags`, are introduced for representing context goals, which are goal conditions that arise in a specific context. In order to facilitate publishing and making `cogotags` generally portable, they are presented in XML form, as shown in Figure 4.3.

Each `cogotag` has three parts, a context type, context value and goal condition that it introduces. There are two types of effect that `cogotag` may have on the `composition request`: *additive* and *subtractive*. An example of an additive `cogotag` is situation "when user is driving read out the directions". This results in the additive condition `(directions speech_out)`, requesting the directions in speech synthesised form. An example of a subtractive `cogotag` would be "when user is driving do not display new e-mail notifications". As a result, the goal

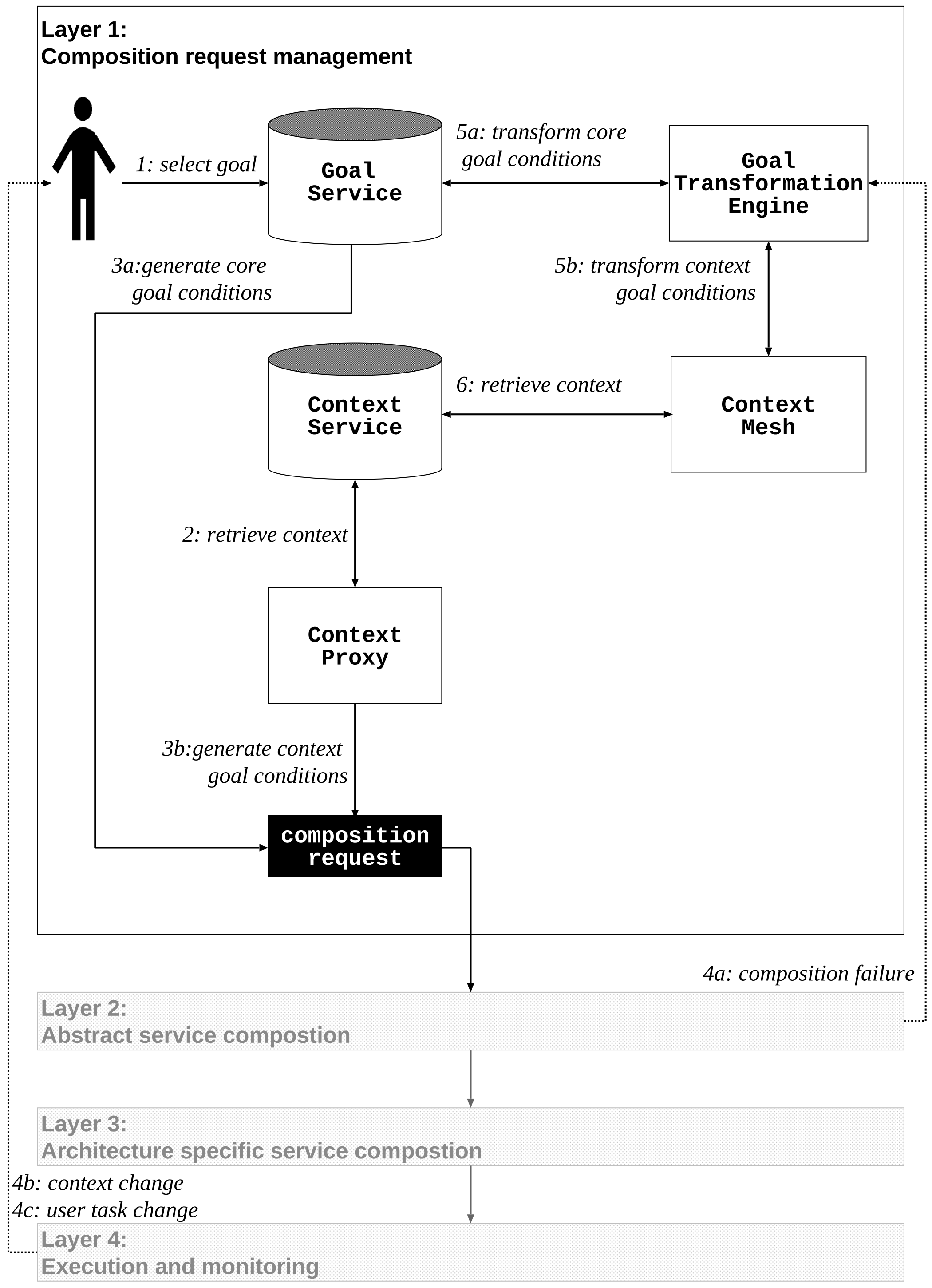

Layer 1:
Composition request management
1: select goal
Goal Service
5a: transform core goal conditions
Goal Transformation Engine
3a:generate core goal conditions
5b: transform context goal conditions
Context Service
6: retrieve context
Context Mesh
2: retrieve context
Context Proxy
3b:generate context goal conditions
composition request
4a: composition failure
Layer 2:
Abstract service compostion
Layer 3:
Architecture specific service compostion
4b: context change
4c: user task change
Layer 4:
Execution and monitoring

Figure 4.2: GoalMorph: composition failure management using context aware goal transformations

```
<COGOTAG>
  <CONTEXT-TYPE> activity </CONTEXT-TYPE>
  <CONTEXT-VALUE> driving </CONTEXT-VALUE>
  <ADD-GOAL-CONDITION PREDICATE="directions" ARGUMENT="speech_out"/>
  <REMOVE-GOAL-CONDITION PREDICATE="email_notification" ARGUMENT="on"/>
</COGOTAG>
```

Figure 4.3: Sample context goal condition in cogotag format

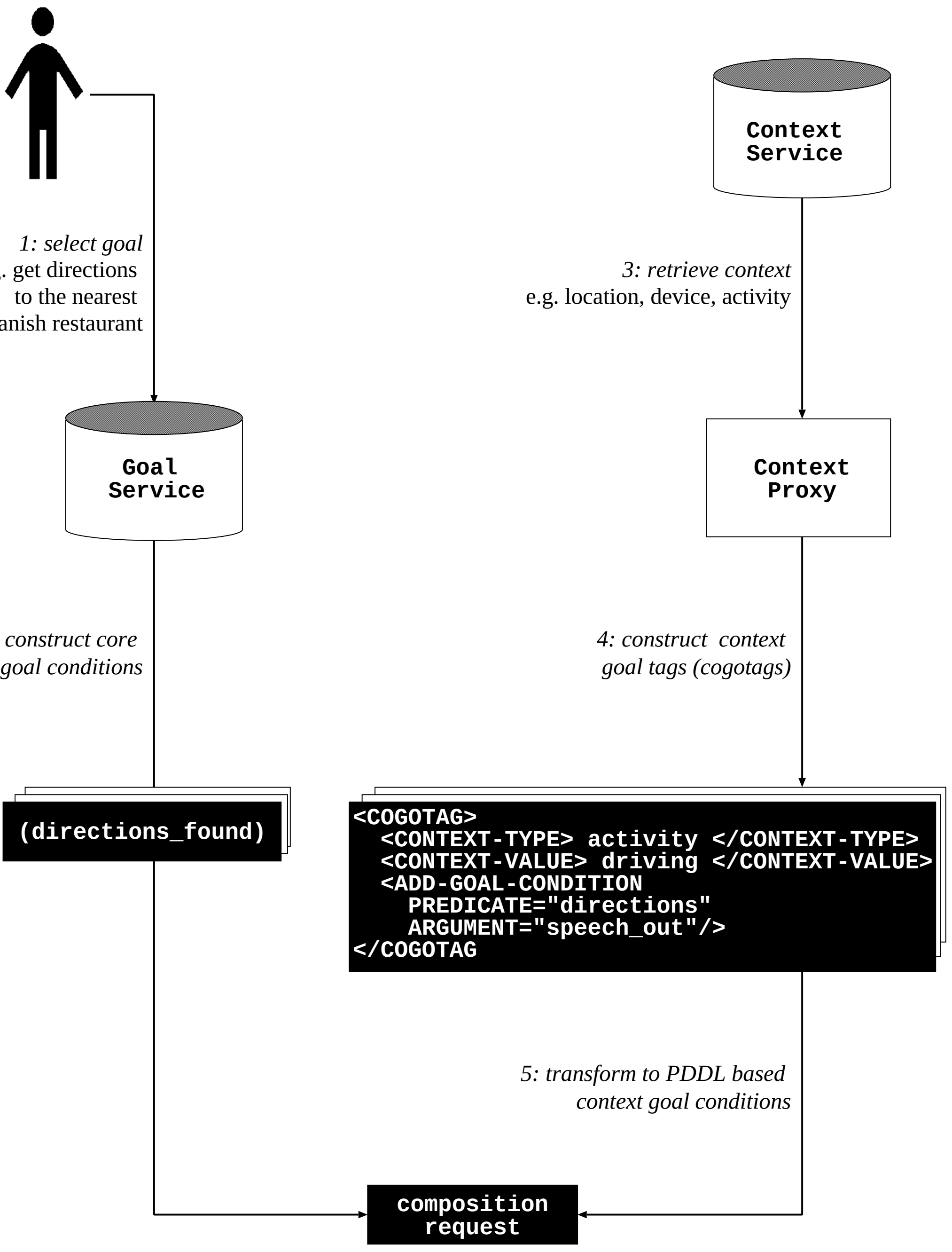
condition `(email_notification on)` is removed from the set of goals forming the `composition request`.

The use of `cogotags` facilitates the dynamic association of context with user goals, without requiring pre-built contextual dependencies in the `Goal Service`. This approach allows for flexible, automated context goal generation, independently of the `Context Service` used.

Following is the description of construction of a `composition request`. Once a user selects a desired task (Step 1 – Figure 4.4), corresponding core goal conditions are fetched from the `Goal Service` (Step 2). The `Context Proxy`, shown in Figure 4.4, is a software component that fetches the current context data from the `Context Service` (Step 3) and constructs `cogotags` (Step 4). The framework then converts XML-based `cogotags` to planner readable PDDL-based goal conditions (Step 5). At the moment tools such as eXtensible PDDL (XPDDL), an XML-based representation for PDDL, introduced by Gough [Gou04] are being developed. The PDDL-based context goal conditions together with core goals constitute the formalised `composition request`.

Several different sources of `cogotags` can exist. Firstly, the `Context Service` can attach `cogotags` to the context data. The `Context Proxy` can add or remove goal conditions based on the user's past interaction with the system. Finally, users may create and carry their own `cogotags` in their personal profile. The current implementation of `GoalMorph` includes a `Context Service` that attaches `cogotags` to the context data.

The `Context Service` may fail to provide access to context data, for example, because of a failure of respective context provider as a result of sensor unavailability or network disconnection. If a desired context value cannot be retrieved from the `Context Service`, during construction of `cogotags`, the `Context Proxy` attempts to obtain a past context value from the `Context Service`. The `Context Service` tracks past contexts in which the user has submitted `composition requests`. The software component that implements the `Context Service` has the facility to cache the historical values. The `Context Service` provides historical values of context type in the following order. Firstly, it returns the most recent


1: select goal
e.g. get directions to the nearest Spanish restaurant
Goal Service
2 construct core goal conditions
(directions_found)
Context Service
3: retrieve context
e.g. location, device, activity
Context Proxy
4: construct context goal tags (cogotags)
5: transform to PDDL based context goal conditions
composition request

```
<COGOTAG>
  <CONTEXT-TYPE> activity </CONTEXT-TYPE>
  <CONTEXT-VALUE> driving </CONTEXT-VALUE>
  <ADD-GOAL-CONDITION
    PREDICATE="directions"
    ARGUMENT="speech_out"/>
</COGOTAG
```

Figure 4.4: Context Proxy: automated construction of context aware composition requests using `cogotags`

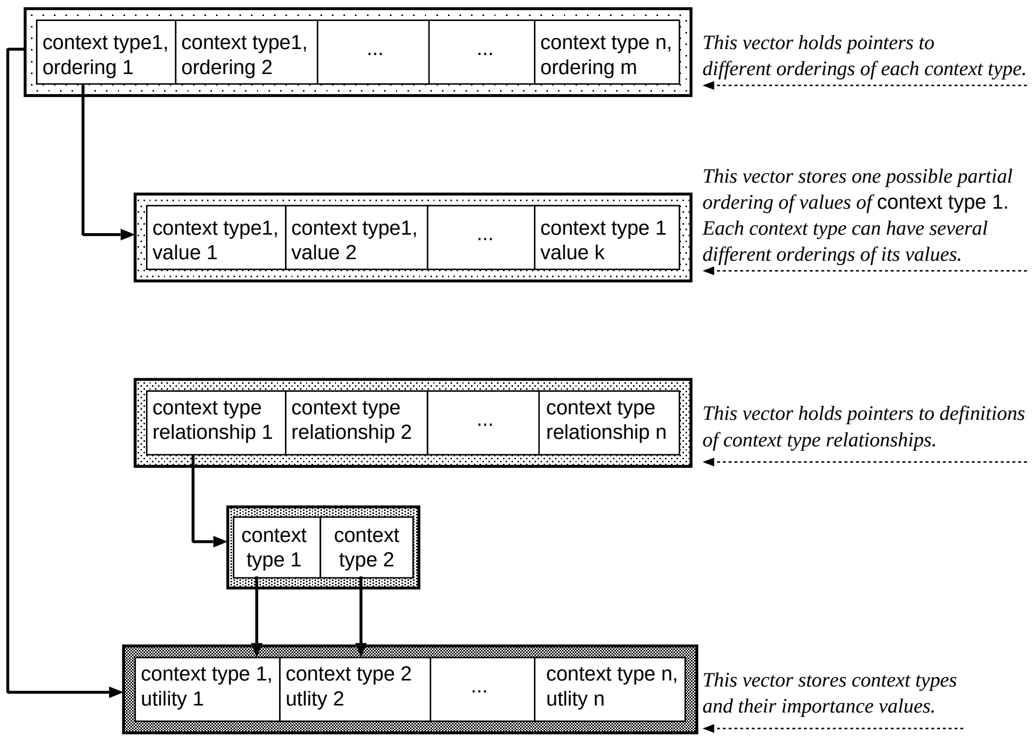

context type1, ordering 1
context type1, ordering 2
...
...
context type n, ordering m
This vector holds pointers to different orderings of each context type.
context type1, value 1
context type1, value 2
...
context type 1 value k
This vector stores one possible partial ordering of values of context type 1. Each context type can have several different orderings of its values.
context type relationship 1
context type relationship 2
...
context type relationship n
This vector holds pointers to definitions of context type relationships.
context type 1
context type 2
context type 1, utility 1
context type 2 utlity 2
...
context type n, utlity n
This vector stores context types and their importance values.

Figure 4.5: Context Mesh: structures for reconfigurable specification of context value orderings and context type relationships.

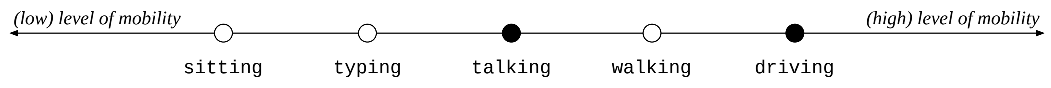
context value of this type. Secondly, it supplies the most frequently occurring context value of this type.

This historical context data can also be used to generate probabilistic predictions about current and future contexts. Development of a system to compute probabilistic predictions, by inferring the missing context based on the available context types,is part of the future work.

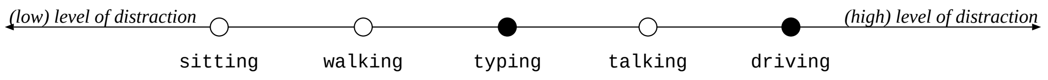
4.2.4 Context Mesh

The `Context Mesh` is a reconfigurable specification of partial orderings of values for each context type, relationships between context types and importance of each context type. It is not an ontology, it does not posses semantics about context types, but rather contains customizable orderings of the context values and context type relationships, as shown in Figure 4.5.

Context value orderings facilitate the substitution of related context values. By substituting context values new context goals are created. This is useful for transformation of unreachable context goals into ones that can be solved.



(a) Activity ordering based on the level of mobility



(b) Activity ordering based on the level of distraction

Figure 4.6: Two orderings for the activity context type

Context orderings. The `Context Mesh` partially orders context values along custom defined hierarchies. For example, possible values of the context type *activity* of a user can be ordered depending on the *level of user mobility. Walking* and *sitting* are sample values of mobile and stationary *activity* context type, as shown in Figure 4.6(a). Shaded circles in Figures 4.6(a) and 4.6(b) represent values that are true in the current context.

The `Context Mesh` allows multiple, scenario-specific orderings of each context type. Aside from the conventional, natural ordering in abstraction type-ordering, context types can also be organised as an enumerated set, a numbered line, or a containment of values. For example, the context type *activity* can be arranged in the following ways. Firstly, as a *set* of activities that occur at a specific location, such as office. Secondly, for instance, activities can be ordered along a numbered line according to the estimated duration of each activity. Thirdly, activities can be represented as a component partonomy, in which activities can be organised in a graph using *part-of* relations. For instance, typing and sitting form the activity called *working on the laptop.* In the example usage scenario *activity* values are ordered along a numbered line, according to the level of distraction they cause to the user. Figure 4.6(b) depicts how *typing* in this case may be considered less distracting than *driving.*

Values of a specific context type are not mutually exclusive. For instance, Miles could well be *talking* to the passenger or on the cellphone and *driving* at the same time, as shown by the shaded values in Figures 4.6(a) and 4.6(b). This gives rise to a number of interesting issues. Deciding which of these two or more activities is of *higher relevance* to the current scenario becomes the problem of identifying which ordering of activity context type should be referred to. For example, according to the "level of mobility" ordering in Figure 4.6(a), *typing* is lower on the scale than *walking*. By contrast, *typing* is higher on the scale than *walking*, when considered according to the ordering by "level of distraction", as shown in Figure 4.6(b). One solution would be to have this, scenario-specific, importance measure specified at the domain engineering stage, as means of identifying preferences. Another issue is how the *conjunction* of more than one context value affects the anticipated behaviour of the application, for example, the conjunction of *talking* and *driving* is in itself more distracting than driving alone. This challenge which will be investigated as part of future work.

Context ordering model. Figure 4.7 depicts how the context orderings and their elements are stored in the `Context Mesh`. Values of each context type can be ordered in several different ways, comprising several different context value orderings. Each of the context ordering vectors stores the context values in a partially ordered manner. These context orderings are reusable accross different application scenarios. The figure shows two vectors, containing two different orderings of the context type *activity*, one for *level of mobility* and one for *level of distraction*. The actual context values, such as *driving*, are represented by a class containing the name of the context type and three lists of pointers `XOR`, `OR` and `AND`. `XOR` is a list of values that are mutually exclusive with the context value in question (e.g. driving and walking). `OR` list holds values that may coexist (e.g. driving and talking), and `AND` lists the values that have to coexist (e.g. driving and sitting). Furthermore, each context value may have a utility value associated with it, denoting its relative importance in relation to other values of this type.

Context relations. Context is by nature highly interleaved. A relationship may exist between different context types, such as an *activity* occurring at a *location*. Figure 4.8 shows how the `Context Mesh` stores such relationships between context types. A vector holds all relationships, and each relationship is represented by a class containing its properties. For example, relationship `happens_at` relates context types *activity* and *location*. For each context type a preferred ordering may be specified as well. In some cases, such as context type *weather*, context values are not ordered. This is represented by context ordering attribute "none". Furthermore, there is a user and scenario-specific utility value assigned to each relationship, which is used to determine which relationships are of higher importance. The context type relationships drive the *context layering* process, described in detail later in this section.

The relative importance of different context types is overwhelmingly *scenario-specific*. For example, consider the case where a user requests driving directions

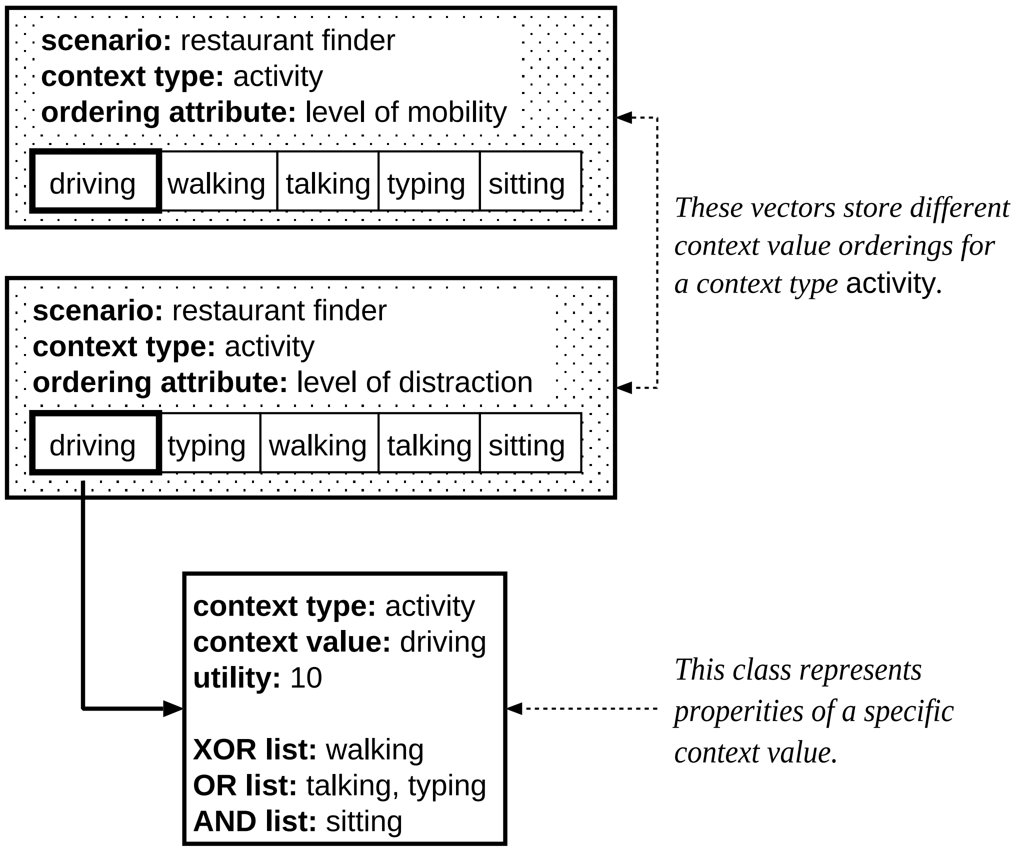

scenario: restaurant finder
context type: activity
ordering attribute: level of mobility
driving
walking
talking
typing
sitting
scenario: restaurant finder
context type: activity
ordering attribute: level of distraction
driving
typing
walking
talking
sitting
These vectors store different context value orderings for a context type activity.
context type: activity
context value: driving
utility: 10
XOR list: walking
OR list: talking, typing
AND list: sitting
This class represents properities of a specific context value.

Figure 4.7: Data structures representing context orderings

to the nearest restaurant. In this case *location*, *activity* and *device used* may be of higher importance than *weather conditions* and *lighting* or *noise levels.* By contrast, the most applicable context types for an e-learning application, an interactive computer-based training software, could be *time requirements*, *user profile* and *device used.* For this reason, a utility value is associated with each context type on a scenario basis.

Context Mesh encoding. The `Context Mesh` requires that initially a qualified domain engineer encodes context values orderings and application-specific context-type relationships. Although this does require additional manual effort, the application programmers' effort saved by enabling applications to be dynamically composed and to evolve is greater. The encoding of `Context Mesh` and `Goal Service` is analogous to the encoding of library functions in high level programming languages, where experienced engineers encode the most commonly used methods to achieve higher code reusability. Context data orderings in Con-

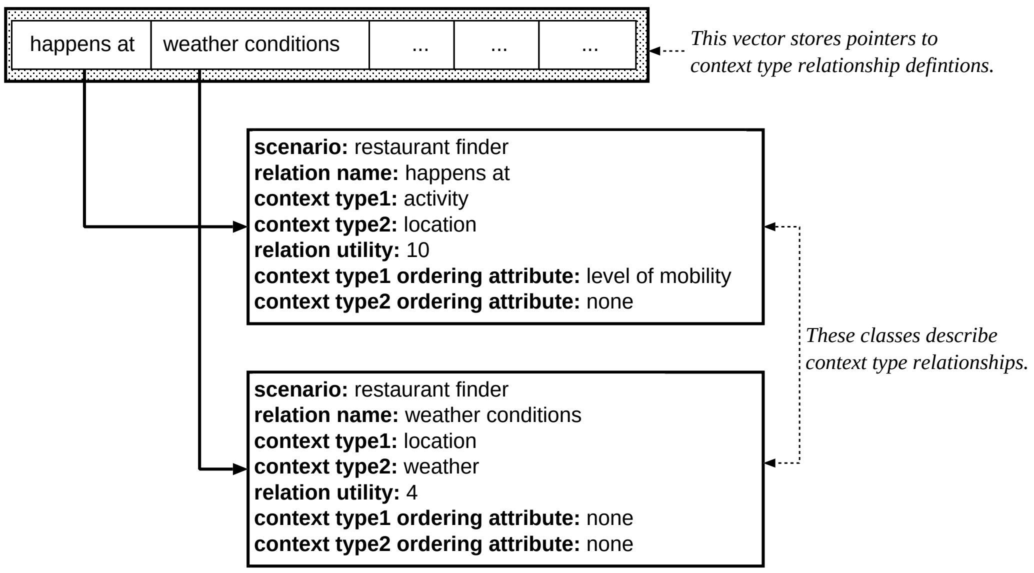

happens at
weather conditions
...
...
...
This vector stores pointers to context type relationship defintions.
scenario: restaurant finder
relation name: happens at
context type1: activity
context type2: location
relation utility: 10
context type1 ordering attribute: level of mobility
context type2 ordering attribute: none
These classes describe context type relationships.
scenario: restaurant finder
relation name: weather conditions
context type1: location
context type2: weather
relation utility: 4
context type1 ordering attribute: none
context type2 ordering attribute: none

Figure 4.8: Data structures representing context type relationships

text Mesh are encoded once by qualified domain engineer and can be reusable accross different application scenarios. As an extension, users or other customize `Context Mesh` orderings, and even define their own.

A large number of orderings, resulting from a variety of user and scenario-specific preferences may in a worst-case scenario result in a high overhead in encoding the structure for organising such data. However, in most real applications the needs of individual users are not entirely unique. For example, the ordering of context type *activity* based on the *level of distraction* will be the same for a number of different users and scenarios. I envisage that by associating orderings with *user profiles* instead of individual users many `Context Mesh` orderings can be highly reusable. `Context Mesh` can thereby be viewed as a set of application and user profile templates, which can be further customized directly by the users.

The `Context Service` provides the information about context types and values, i.e. activity is cycling. The domain engineer (or user) assigns this context value to the corresponding context ordering, e.g. defines that activity cycling is more distractive than driving.

Context layering. Supporting utility-annotated relationships between context types enables the operation of *context layering*, shown in Figure 4.9. The bold framed values, such as *in-vehicle information system* (*IVIS*) and *in town*, represent values that are true in current context. Context layering refers to the process

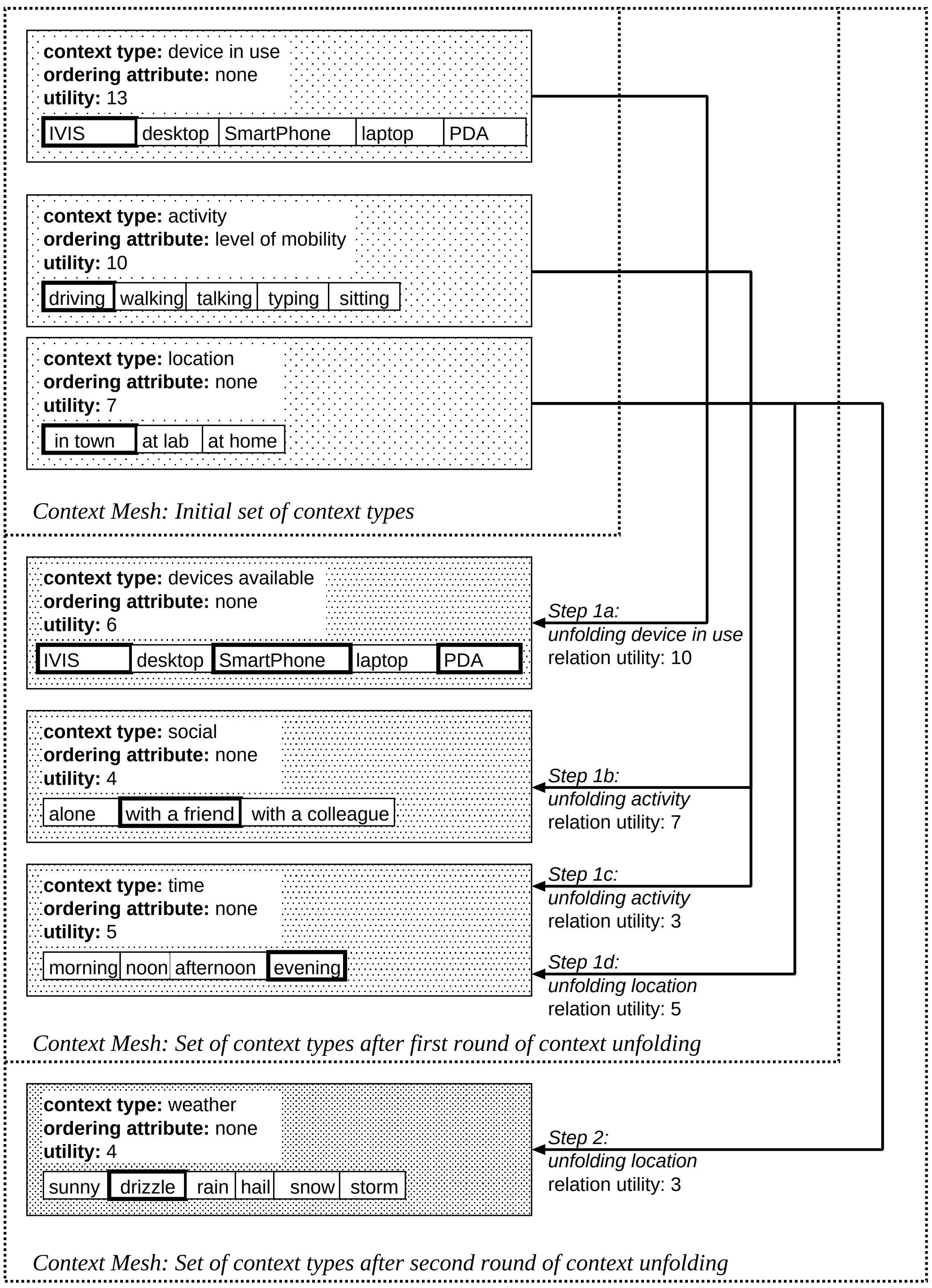


Figure 4.9: Context unfolding process.

of dynamically expanding and contracting the set of context types that are taken into account by the service composition framework. Context layering consists of two operations: *context unfolding*, which introduces additional context types that are related to the current set of context types considered, and *context folding*, the process that removes context types from the same set.

The algorithm that performs context unfolding works in the following way. It starts with an initial list of context types considered, which we term the *original context*. This includes "device in use", "activity", and "location" context types in the example in Figure 4.9. Starting with the context type with the highest utility, and for each context type in the original context, the algorithm finds the context types that are *related* to it. Then, it adds the related context types to the set of considered constraints, and moves to the next context type in the original context. A threshold may be specified to ensure that only relationships of at least a certain utility will be considered when unfolding the context.

In the example in Figure 4.9, based on the utilities of the context types the *device in use* context type is unfolded first, introducing the new type *devices available*. This is followed by unfolding of the *activity* context type, which introduces *social context*. The resulting set of considered context types is called the *single-unfolded context*. Potential consecutive unfolding steps can be taken; in the example the *weather* context type is introduced by unfolding *location*, resulting to a *double-unfolded context*.

As pervasive computing environments may introduce a practically unlimited number of context types related to a user task, organising context by relevance in layered context type sets is useful for controlling the number of context types taken into account by the composition framework.

Context goal transformations. Selecting a different context value from context type ordering vector enables substitution of a context value with a *weaker* or *stronger* one. In the same way, moving through the vector of context relationships facilitates the process of context layering, which expands or reduces the set of context types taken into account by an application.

These structures and the operations they provide for traversing through their values, facilitate transformation of *failed* context goals, i.e. goal conditions that were not achievable during the planning process. The context goal transformation is a process of substitution of a specific context value. Once the new context value is obtained, it is passed to the `Context Proxy`. As a result this triggers a new, modified context goal.

Following is a list of transformations provided by the `Context Mesh`:

1. *Weakening.*

   Definition: Movement along the specified context type ordering towards a weaker value, i.e. lower on the scale.

   Example: Goal to display information on an LCD screen may be substituted by a goal to present information on any display device visible and available.

2. *Substitution.*

   Definition: The process of obtaining an equivalent substitute context value, called a sibling value.

   Example: Goal to display information on a desktop PC's LCD screen may be substituted by a goal to display information on a TabletPC's LCD.

3. *Strengthening.*

   Definition: Movement along the specified context type ordering towards a stronger value, i.e. higher on the scale.

   Example: Goal to display information on any type of display device available in the environment may be substituted by a goal to display information on an LCD or CRT.

4. *Context unfolding.*

   Definition: The process of expanding the set of context types taken into consideration. This may often result in a refined goal, where the plan may eventually overachieve the original goal. This is useful when there are operators with partially satisfied preconditions. Retrieving additional context types may enable the selection of operators previously not applicable in the planning process.

   Example: The goal condition requires that information is to be displayed on an LCD screen. When an LCD screen cannot be found in the current environment of the user, the goal cannot be satisfied. However, a CRT screen is discovered in the environment, but an examination of the available operators shows that using this CRT screen has a precondition where `(location in_kitchen)`. At the moment, the location of the user is unknown. By expanding the `Context Mesh` and unfolding the location context type, the user's present location is detected and the applicability of displaying the information on the available CRT can be established.

5. *Context folding.*

   Definition: Removal of context types, i.e. reduction of number of context types taken into an account. As a result a context goal is removed from the current set of open goals that the planner must achieve.

   Example: No text to speech service is available. By removing the *activity* context type from the `Context Mesh`, as a goal condition for displaying directions in the speech form is also removed.

6. *Historical value substitution.*

   Definition: Obtains a substitute, past value of context at a specified point in time.

   Example: If a sensor has failed and a context value can not be obtained, a context history is accessed to try and retrieve a past context value.

Summary. The `Context Mesh` is a reconfigurable specification of orderings of context type values, and the relationships between context types. This allows for substitution of related context values, thereby facilitating reformulation of unreachable context goals into ones that can be solved by the planner. Encoding of the relationships between context types further enables context layering, process of expansion and reduction of context types taken into consideration by the framework. Consequently this process introduces new or removes old goal conditions from the original `composition request`.

4.2.5 Goal Service

The `Goal Service`, as shown in Figure 4.10, stores task descriptions and their respective formalised definitions. For example, in our usage scenario the task is to find restaurant directions. The goal conditions required to be reached for this task are associated with the task description. The `Context Mesh` as previously described specifies relationships and orderings between context types and their values. Similarly the `Goal Service` organises core goals and their arguments into reconfigurable orderings, which facilitate core goal transformations.

Core goal selection and storage. Each `composition request` is a list of goal conditions, specified in a language used by the planner. One way to construct a `composition request` is to have users manually select the individual goal conditions to be included. However, while the users know what task they want to execute, they may not know how to realise the `composition request`, requiring

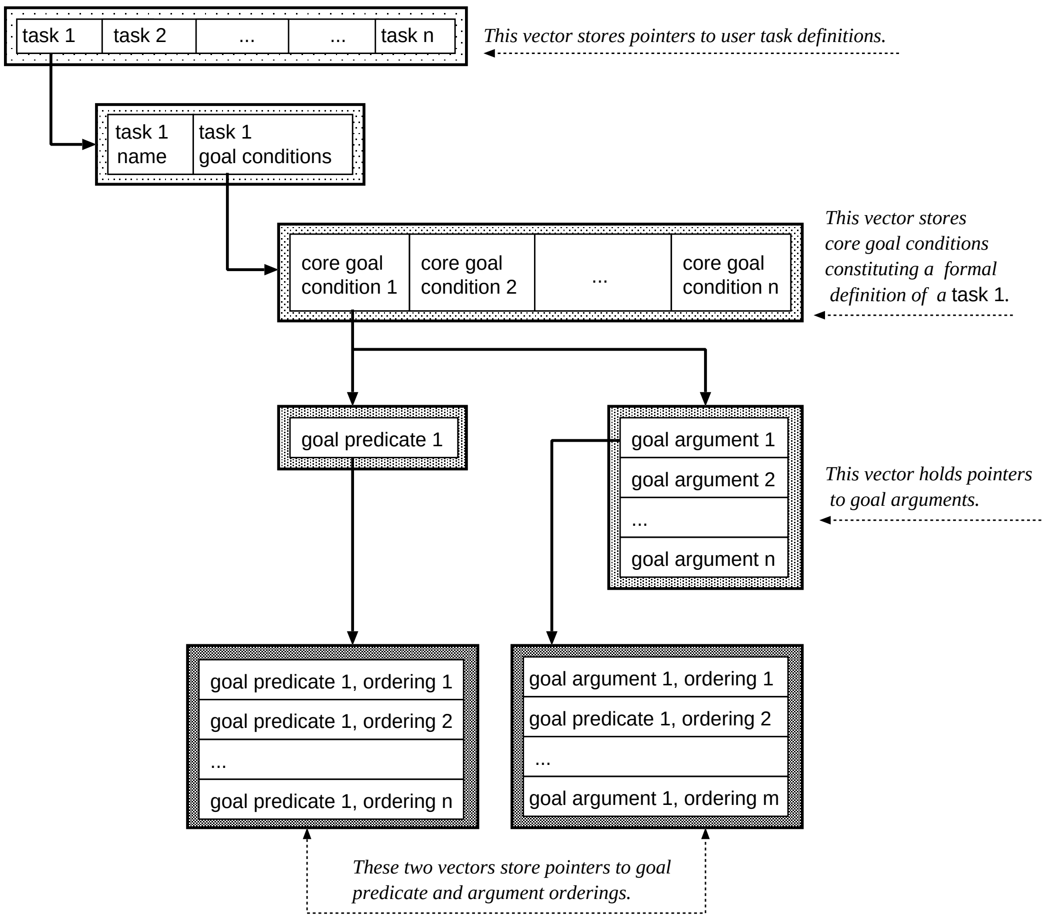
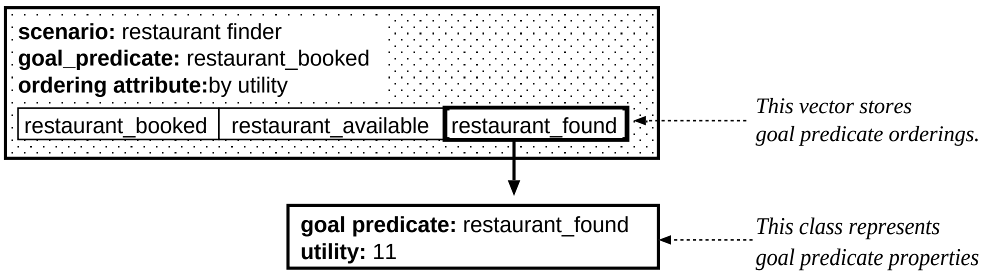

task 1
task 2
...
...
task n
This vector stores pointers to user task definitions.
task 1 name
task 1 goal conditions
core goal condition 1
core goal condition 2
...
core goal condition n
This vector stores core goal conditions constituting a formal definition of a task 1.
goal predicate 1
goal argument 1
goal argument 2
...
goal argument n
This vector holds pointers to goal arguments.
goal predicate 1, ordering 1
goal predicate 1, ordering 2
...
goal predicate 1, ordering n
goal argument 1, ordering 1
goal predicate 1, ordering 2
...
goal argument 1, ordering m
These two vectors store pointers to goal predicate and argument orderings.

Figure 4.10: Goal Service: structures for reconfigurable specification of tasks, goal predicates and arguments.

a user to understand the low-level details of an unfamiliar planning syntax, and the details of the domain knowledge, is not desirable.

There are several stakeholders in defining a `composition request`. Users specify their task intention. Service providers define service operation descriptions, and thereby restrict the set of available goals. Finally, context data limits the applicability of existing goal conditions.

The `Goal Service` stores available core goal conditions and organises them into tasks supported by the framework. These goal conditions are explicitly defined by domain engineers using the planning language. Once the user selects a task using a provided Graphical User Interface (GUI), the corresponding goal conditions are retrieved from the `Goal Service` and the user is subsequently prompted to supply the necessary arguments to complete the construction of a `composition request`.



scenario: restaurant finder
goal_predicate: restaurant_booked
ordering attribute:by utility
restaurant_booked
restaurant_available
restaurant_found
This vector stores goal predicate orderings.
goal predicate: restaurant_found
utility: 11
This class represents goal predicate properties

Figure 4.11: Data structure for goal predicate orderings

Moreover, the `Goal Service` keeps track of all `composition requests`, the context of their invocation and the number of times each was submitted. This information can be used to order the previous `composition requests` by their invocation frequency, when displaying them in the GUI for goal selection. Furthermore, it can be used to proactively make `composition request` recommendations and automate their selection.

Goal predicate orderings. The `Goal Service` holds a number of vectors, which partially order core goal predicates. Figure 4.11 shows some of the possible forms that the `(restaurant_booked)` goal condition may take. For example, given the ordering in Figure 4.11, the goal condition `(restaurant_found)` is the *weakest* replacement for the `(restaurant_booked)` goal condition. The goal condition `(restaurant_available)` represents a state when the restaurant can be found and its availability queried, however the booking cannot be performed. Therefore this is also a weaker form of the original `(restaurant_booked)` goal condition. The goal predicates are represented as a class holding a name and relationships between predicates, as shown in Figure 4.11.

Goal conditions have utility values assigned to them, initially by a domain engineer, which can be updated using feedback from the planning system. These can be used to identify goal conditions that contribute to the effectiveness of the overall `composition request` when devising a partial solution. For instance, the utility value of `(restaurant_found)` in 11, as shown on Figure 4.11. Goal utilities are application- and user-specific. Designing a system to create and maintain these utility values is part of the proposed future work.

Argument orderings. Goal predicates may contain a number of arguments. Therefore, similar to goal predicate orderings, each argument may be associated with one or more *argument* orderings, thereby facilitating transformations of goal arguments. For example, the `(restaurant_found cuisine_argument location_argument)` goal is associated with two arguments. The first argument, `cuisine_argument`, denotes the type of the cuisine to be found, such as Spanish. Similarly, the second argument `location_argument` is used to define the location of the restaurant, such as Cambridge.

Figure 4.12 shows the possible partial ordering for the `cuisine_argument`, which is associated with the goal predicate `(restaurant_found)`. For example, `spanish` cuisine is a more specific value of a `mediterranean` cuisine type. Argument ordering values are associated with utility values, like goal predicate orderings. For instance, a `croatian` restaurant, having a utility 13, may be preferred to a `portuguese` one, which carries utility value 6.

Figure 4.13 shows a vector storing a possible ordering of the `cuisine_argument` argument. The argument orderings can be scenario, goal and user-specific. Similarly to context values and goal predicates, a utility value can be associated with an argument value, which is used to determine its importance.

Core goal transformations. By moving up and down the goal predicate and argument orderings, the `Goal Service` enables substitution of goal predicates and arguments respectively. As a result, new *core* goal conditions are assembled, thereby substituting the failed *core* goals in the original `composition request`. Like *context* goal transformations provided by the `Context Mesh`, core goal transformations can take several forms:

1. *Weakening*

   Definition: Movement along the goal type or argument ordering respectively towards a weaker value, i.e. lower on the ordering scale.

   Example 1: Type weakening

   The `(restaurant_booked)` goal condition may be substituted by the goal condition `(restaurant_available)`.

   Example 2: Argument weakening

   Similarly, if the Spanish restaurant requirement, from the usage scenario, cannot be satisfied, it can be substituted by the goal condition in which the argument is generalised. As a result the goal condition `(restaurant_found mediterranean)` is instantiated.

   It is important to note that once the top most element in the argument ordering is reached it will not be removed. Argument removal is considered as a separate operation, in order to allow (less important) arguments

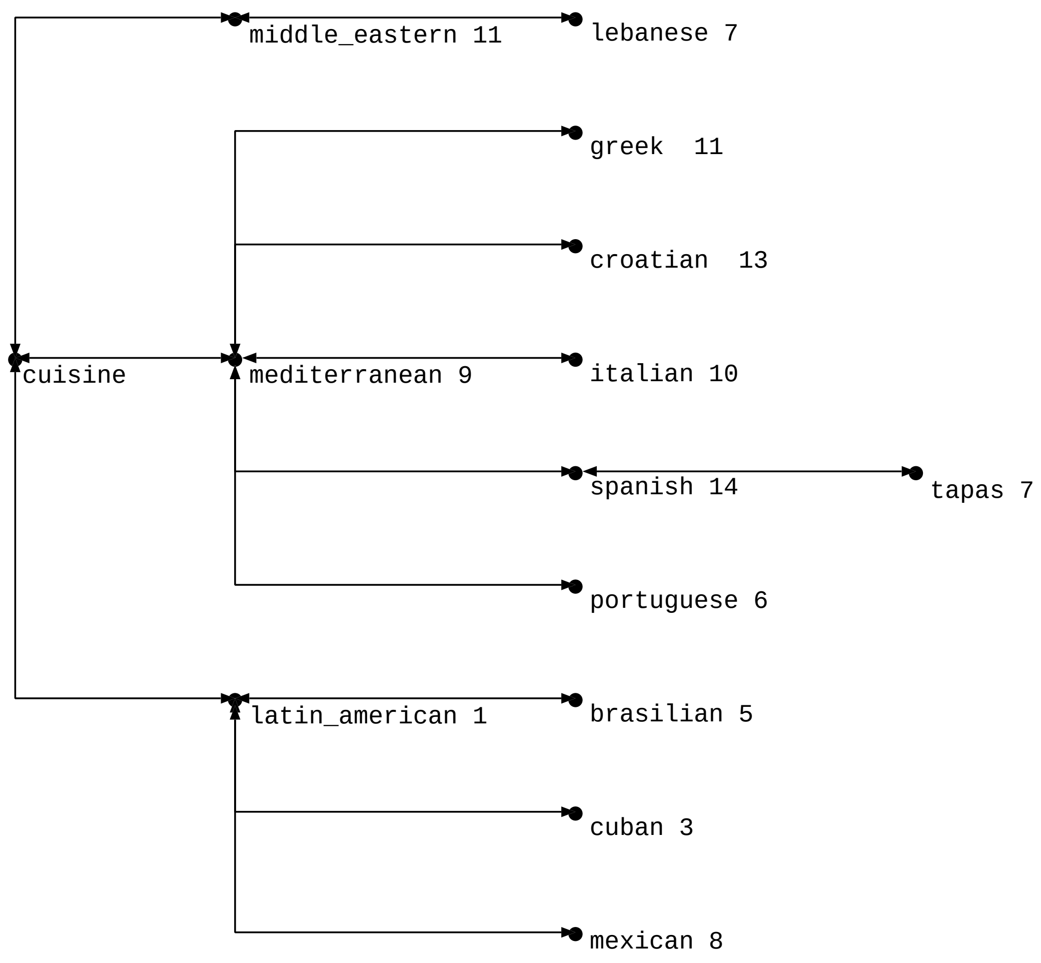


Figure 4.12: Sample argument ordering for argument (`cuisine_argument`)

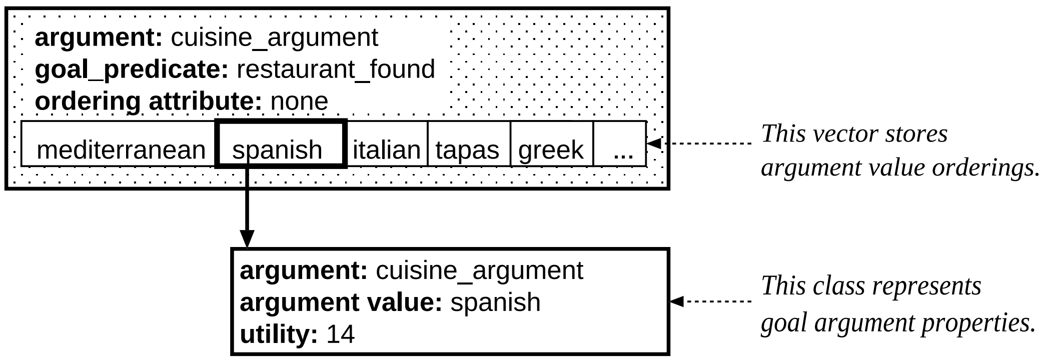

argument: cuisine_argument
goal_predicate: restaurant_found
ordering attribute: none
mediterranean
spanish
italian
tapas
greek
...
This vector stores argument value orderings.
argument: cuisine_argument
argument value: spanish
utility: 14
This class represents goal argument properties.

Figure 4.13: Data structure for argument orderings

to be removed sooner in the goal transformation process and to prevent unnecessary traversing through argument ordering.

2. *Strengthening*

   Definition: Movement along the goal predicate or argument ordering respectively towards a stronger value, i.e. higher on the ordering scale.

   Example 1: Type strengthening

   The `(restaurant_found)` goal state may be substituted by the "stronger" goal condition `(restaurant_available)`.

   Example 2: Argument strengthening

   Once the base goal conditions are satisfied, further constraints on the goal may be imposed. For instance, requesting that the desired restaurant must be Spanish and have a parking facility.

3. *Relaxation*

   Definition: Removing the goal type or goal argument constraint

   Example 1: Predicate relaxation

   Removal of this goal condition from the set of goal states to be satisfied.

   Example 2: Argument relaxation

   For example finding any restaurant, as opposed to a Spanish one. This is a result of removing the constraint on the restaurant cuisine.

4.2.6 Goal Transformation Engine

Overview. The `Goal Transformation Engine` is a component that coordinates the process of reformulation of failed `composition requests`. This process consists of *core* and *context* goal transformations. There are two circumstances under which goal transformations are performed. Firstly, when *no plan is found* for a given request. Secondly, when a plan with a *higher utility* can be applied to solve a `composition request` and as a result overachieve the original one.

Transformation algorithm. Algorithm 4.1 outlines the operation of the `Goal Transformation Engine`, which provides composition failure management facility. The `Goal Transformation Engine` takes three input parameters: S_{GOALS}, a list of goal conditions forming the `composition request`, $planningSuccess$, the outcome of the planning process and $compositionRequestID$, used to store

Algorithm 4.1 Composition failure management

```
1: procedure CompositionFailureManagement(
       $S_{GOALS}$ ,
       $planningSuccess$,
       $compositionRequestID$)
2:     while ($planningSuccess \neq true$
       or $timeOut$
       or $maxNumTransformationsReached$) do
3:         $S_{ACHIEVABLE} = 0$
4:         $S_{CORE} = 0$
5:         $S_{CONTEXT} = 0$
6:         $S_{ACHIEVABLE}, S_{CORE}, S_{CONTEXT} \leftarrow separateGoals(S_{GOALS})$
7:         $S_{TRANSGOALS} = 0$
8:         $T = 0$
9:         $S_{TRANSGOALS} = ContextAwareGoalTransformation($
       $compositionRequestID$, $S_{CORE}, S_{CONTEXT})$
10:        $S_{TRANSGOALS}.add(S_{ACHIEVABLE})$
11:        $planningSuccess \leftarrow plan(S_{TRANSGOALS})$
12:        $updateTransformationUtilities(compositionRequestID,$
       $planningSuccess)$
13:        $S_{GOALS} = S_{TRANSGOALS}$
14:    end while
15: end procedure
```

data related to this `composition request`. `Goal Transformation Engine` processes S_{GOALS} if planning has failed and separates the failed from the achievable goal conditions. It generates two sets of failed goal conditions: S_{CORE}, a set of failed *core* goal conditions, and $S_{CONTEXT}$, a set of failed *context* goal conditions. It stores achievable goals into $S_{ACHIEVABLE}$. Two sets of failed goals, S_{CORE} and $S_{CONTEXT}$, are used by context aware goal transformation operation, outlined in Algorithm 4.2, which generates $S_{TRANSGOALS}$, the set of transformed goal conditions. $S_{TRANSGOALS}$ is then passed to the planning system. If the planning process succeeds the algorithm terminates, otherwise it enters another round of transformation. The result of the planning process is used as feedback to update the utilities of transformations stored in set T, which keeps track of all transformations applied to the failed goals.

Algorithm 4.2, which is invoked in line 9 of Algorithm 4.1, describes the process of context aware goal transformation in which transformations are selected

```
(define (goal_3_original)
    (restaurant_found lebanese zurich)
    (restaurant_booking_made 2 2000)
    (restaurant_booked restaurant_name)
    (directions_found current_address restaurant_address)
    (directions speech_out)
    (directions language english)
)
```

(a) Original composition request for Case 2

```
(define (goal_3_transformed)
    (restaurant_found middle_eastern area_3_zurich)
    (restaurant_booking_made 2 2100)
    (restaurant_booked restaurant_name)
    (directions_found current_address restaurant_address)
    (directions language english)
    (directions stephanie smartphone)
)
```

(b) Transformed composition request for Case 2 request shown in 4.2.6

Figure 4.14: Sample composition request and transformed request

based on their utilities. To illustrate this methodology consider again what happens when the `Composition Engine` fails to satisfy the request in Case 2 of the usage scenario, shown in Tables 4.1 and 4.2.

For example, the `Composition Engine` cannot satisfy two goal conditions from the `composition request` shown in Figure 4.14(a). The first one is the *core* goal condition `(restaurant_found lebanese zurich)`. The second one is the *context* goal condition `(directions speech_out)`.

The goal `(restaurant_found lebanese zurich)` is a base core goal, meaning that without satisfying some form of this goal condition the whole request is of no utility to the user. The utility-driven algorithm, selects the transformation, t, with highest-utility, as shown in line 7 of Algorithm 4.2. For example, by applying an argument weakening transformation, following the ordering shown in Figure 4.12, it constructs the goal `(restaurant_found middle_eastern zurich)`. Each selected transformation is added to the set of final transformations T. The transformed goal, g_t is added to $S_{CORETRANS}$, the set of transformed core goals.

Once core goal conditions have been reformulated, *context* goal transformation is performed, as shown in line 12 of the algorithm. The failed context

Algorithm 4.2 Context aware goal transformation using utilities

```
1: procedure ContextAwareGoalTransformation(
        $compositionRequestID$, $S_{CORE}$, $S_{CONTEXT}$)
Require: $S_{CORE} \neq 0$ or $S_{CONTEXT} \neq 0$
2:     $T = 0$
3:     $S_{TRANSGOALS} = 0$
4:     $S_{CORETRANS} = 0$
5:     $S_{CONTEXTTRANS} = 0$
6:     for all $g \in S_{CORE}$ do
7:         $t = GoalService.getTransformationWithHighestUtility(g)$
8:         $g_t = GoalService.transformGoal(g, t)$
9:         $T.add(t)$
10:        $S_{CORETRANS}.add(g_t)$
11:    end for
12:    for all $g_c \in S_{CONTEXT}$ do
13:        $S_{CONTEXTPAIRS} \leftarrow ContextProxy.getContextValuePairs(g_c)$
14:        for all $s \in S_{CONTEXTPAIRS}$ do
15:            $c = ContextProxy.getContextType(s)$
16:            $v = ContextProxy.getContextValue(s)$
17:            $o = ContextMesh.getKeyContextOrdering(c)$
18:            $t = ContextMesh.getTransformationWithHighestUtility(c, v, o)$
19:            $s_t = ContextMesh.transformContext(s, t)$
20:            $S_{NEWPAIRS}.add(s_t)$
21:            $T.add(t)$
22:        end for
23:        $g_{ct} = ContextProxy.generateGoal(S_{NEWPAIRS})$
24:        $S_{CONTEXTTRANS}.add(g_{ct})$
25:    end for
26:    $S_{TRANSGOALS}.add(S_{CORETRANS})$
27:    $S_{TRANSGOALS}.add(S_{CONTEXTTRANS})$
28:    $GoalTransformationEngine.notify(compositionRequestID, T)$
        return $S_{TRANSGOALS}$
29: end procedure
```

goals in the usage scenario is the goal condition `(directions speech_out)`. The `Goal Transformation Engine` interacts with the `Context Mesh` and the `Context Proxy` to identify which context type and value pairs trigger this goal condition, shown in line 14. In the usage scenario, it is the *activity* of the user that implies that the directions should be in the synthesised speech format. For each context value pair s, obtained in line 14, the algorithm acquires the ordering o, which will be used for transformations of this context type c, and its value v.

The `Context Mesh` may perform one of the transformations as described in Section 4.2.4. For example, given context types *activity*, *location*, and *device in use*, it expands the context types and includes *social setting*, *time*, *weather* and *devices available.* It uses context utilities to determine in which order context types will be unfolded and in which order new context types are to be included in the transformation process. For instance, as shown in Figure 4.9, firstly the social context and devices available will be considered. Once all the context value pairs have been transformed for a specific context goal condition, Context Proxy, as shown in line 23 of Algorithm 4.2, assembles a new context goal based on the new set of context value pairs, called $S_{NEWPAIRS}$.

By acquiring the social context, the `Context Mesh` may find out that Miles is driving together with Stephanie, represented by the literal `(social with_friend stephanie)`. Furthermore by acquiring information about other devices available in the environment, it may find out that both Miles and Stephanie own a SmartPhone each, in addition to Miles' laptop being in the back seat. It may therefore choose to replace the goal of reading out driving directions with the goal condition `(directions stephanie smartphone)`. This will forward the directions to Stephanie's SmartPhone, so that she can guide Miles through Zurich. It is the purpose of the `Context Service` to filter out and show only the visible and reachable devices in the environment. For example, if the directions are to be routed to Miles' laptop in the back seat, the corresponding notification about the routing will be displayed on the current device in use. Figure 4.14(b) shows the transformed `composition request`, which retains the same size, however its utility is changed as a result of transformations.

For each context goal, the set $T_{CONTEXT}$ of applicable transformations is generated. As with core goal conditions, the highest utility transformation, t, is selected and applied to generate a transformed goal condition, g_{ct}. The transformations and the substitute goal are appended to the set of final transformation T and the set of transformed context goals $S_{CONTEXTTRANS}$.

Procedure `unfoldContextType`, shown in Algorithm 4.3 outlines the unfolding process, shown in Figure 4.9, which incorporates additional context types and their values into `GoalMorph`. Context types are sorted in decreasing utility order,

Algorithm 4.3 Supporting procedures for context aware goal transformation

```
1: procedure UNFOLDCONTEXTTYPE($c$)
2:     $S_{CTYPES} = ContextMesh.findRelatedContextTypes(c)$
3:     $O_{CTYPES} = orderContextTypesByDecreasingUtility(S_{CTYPES})$
4:     for all $c_r \in O_{CTYPES}$ do
5:         $v = ContextProxy.value(c_r)$
6:         $g_c = ContextProxy.generateContextGoal(v)$
7:         $GoalTransformationEngine.S_{CONTEXT}.add(g_c)$
8:     end for
9: end procedure

10: procedure UPDATETRANSFORMATIONUTILITIES($T$, $planningSuccess$)
11:     for all $t \in T$ do
12:         $u = GoalTransformationEngine.getUtilityValue(t)$
13:         if $planningSuccess$ then
14:             $u = u + 1$
15:         else
16:             $u = u - 1$
17:         end if
18:     end for
19: end procedure
```

so that context types can be unfolded in the order of their importance.

At present, `GoalMorph` examines the result of the planning process, and transforms the goal conditions that have not been met, and then triggers replanning. However, `GoalMorph` can be configured to transform all the goal conditions, regardless whether or not they were initially satisfied during planning, or to transform only one goal condition in each run (e.g. the least or most important one).

Transformation selection. Table 4.3 summarises the operations provided by the `Goal Service` and the `Context Mesh`, which facilitate *core* and *context* goal transformation respectively.

There are four types of input to the process of computing the set of applicable transformations for each goal condition: (1) domain control knowledge, (2) utility function, (3) user input and (4) randomised algorithm.

Domain control knowledge is provided at the domain engineering stage. It includes control structures expressing the priorities among goal transformations for a given scenario. This approach however introduces overhead, which arises

Core goal transformations performed by Goal Service	**Context goal transformations performed by the Context Mesh**
1. Weakening	1. Weakening
2. Strengthening	2. Substitution
3. Relaxation	3. Strengthening
	4. Context unfolding
	5. Context folding
	6. Historical value substitution

Table 4.3: List of core and context goal transformations

from the requirement to encode domain control knowledge.

A *utility function* may be applied together with feedback from the planner, as shown in Algorithm 4.3. Firstly, the utilities of substitute goals are determined to calculate the overall cost-benefit of the substitute solution. The higher utility of a given tranformation, the higher its priority. There may be situations where two or more transformations have the same utility value, in which case a random selection is made. If there is a case where both goal weakening and strengthening are of the same utility, strengthening transformation is preferred. When conflicting context goal conditions emerge from the transformation the original goal has precedence over the transformed one.

Additionally, the utilities of the corresponding transformation are updated, based on the success or failure feedback from the planning system. For each transformation that contributed to a satisfaction of the `composition request`, the utility is increased, and vice-versa. When evaluating a number of different transformed `composition requests`, a cumulative utility of their goal conditions is used. More sophisticated and advanced methods such as machine learning techniques together with user feedback can be used, however they are out of scope of this work.

User input may also be used to guide goal transformations. In that case, the domain control knowledge would be used to identify the set of applicable transformations. The user is then prompted via the graphical interface to select the transformation. This method, however, requires the user to be familiar with the internal representation of goal conditions.

Finally, it is possible to use a *randomised* algorithm, which randomly selects the transformation to apply to each goal condition.

The current implementation of `GoalMorph` supports random and utility based search for selecting transformations. The next section evaluates these two approaches.

Goal dependence and transformation selection. Certain goals can be inter-dependent, being a precondition of one another. As a result, it is possible that transforming one goal may make the other (un)achievable, and cause the transformation algorithm to potentially enter an infinite loop.

One possible way of dealing with loops would be to encode goal relationships in the domain control knowledge. This would allow planner to avoid such loops, by having the information about all the pairs of dependent goals. However this approach introduces prohibitive manual overhead, requiring that interrelations between all the goals, including their corresponding transformations, are explicitly stated.

A special case of goals are *context dependent goals*, as described in Section 4.2.1. These goals are directly associated with core goals. The goal transformation handles context dependent goals by associating their transformation with that of their "parent" core goals. For example, if the core goal is removed from the goal list, so will any of its context dependent goals.

The challenge arises when handling transformations of core and context goals which may be preconditions to each other. For example, goal condition g_a is the precondition of the goal condition g_b. Goal g_a is initially achievable, and g_b is not. However, after the transformation of g_a is no longer achievable. At present the system imposes a limitation on the maximum number of transformations applied to a pair of goals, as well as a timeout period. If that occurs, the system backtracks to the previous transformation run, and preserves the goal conditions of higher importance. It uses feedback from the planner to compare the achievable goals against the list of pending goals in each iteration of the transformation process. It uses goal utilities to determine the goal importance. For example, if following a transformation a (core) goal with higher priority is satisfied and the previously achievable goal with lower priority is no longer achievable, it will proceed with this solution. In case of a randomised transformation algorithm, the system will attempt to preserve core goals.

4.2.7 GoalMorph in use

This section discusses how `GoalMorph` can be used to realise the motivating scenario described in Section 4.1.1. In this case, Miles requests driving directions to the nearest Lebanese restaurant in Zurich. Because Miles is driving, he expects the directions to be read out to him. However, as the speech synthesiser facility is not available, the planning process for this `composition request` fails.

At this point in time `GoalMorph` is triggered. Firstly, `GoalMorph` examines the output produced by the planning system. It analyses goals and groups them into sets of failed goals, and achievable goals. In this case the failed goal was the one requesting directions to be read out to Miles. This goal was triggered by the fact that Miles was driving.

Once core goals have been reformulated by applying goal transformations, `GoalMorph` passes the control to `Context Mesh`, which may perform *context layering*, the process of expanding and reducing number of context types taken into consideration by the system. At some point in time the algorithm performs *context unfolding*. As a result aside from initially considered context types such as *activity*, *location* and *device in use*, it includes other related context types. For example, the algorithm discovers the relation between *activity* and *social setting* of the user. After unfolding of the context type *activity*, it acquires the social context of Miles it establishes that he is in the car with Stephanie. At the next iteration of *context unfolding*, `Context Mesh`, also introduces another context type, which describes *the available devices* in the environment. By acquiring the current values of this context type, it establishes that there are other devices present in the environment that can be used, such as Stephanie's mobile phone or Miles' PDA. These new context types and their values, when introduced in the system, trigger new context goals. For example, directions can now be forwarded to Stephanie's phone. This new, reformulated `composition request` is then passed back to the composition system.

4.3 Evaluation

Experimental evaluation was conducted on the prototype implementation of the `GoalMorph` system to determine its effectiveness, scalability and impact on the overall performance of the proposed service composition framework. The experiments were performed on a dual Pentium III 800 MHz processor with 2 GB RAM.

The domain model of the usage scenario, described in Sections 3.1 and 4.1, contained 100 facts and 20 operators. To evaluate `GoalMorph`, *failure injection* was performed, in which actions were randomly removed from the domain to simulate missing services. During each failure injection round, the domain size was reduced by 20% on average.

The `Context Mesh` contained seven context dimensions, such as the ones described in Section 4.2.4, and shown in Figure 4.9. Each context type contained up to ten different values and their corresponding utilities, which were initially assigned random generated values. Furthermore, each context type was associated with one or more orderings.

The size of `composition requests` varied from 10 to 40 goal conditions. The system was exercised with the following sample `composition requests`:

1. Find a dining or entertainment venue (location-based)
2. Find an entertainment venue (event-based)
3. Find a dining venue (cuisine-based)
4. Find directions to the venue
5. Book dining or entertainment venue
6. Make booking and find directions for dining and entertainment venues

All the planning was performed by TLPlan [BK95] in breadth first search mode with no search control knowledge. TLPlan normally uses domain specific search control information to guide simple forward chaining search, where the planning operators are applied to the current state to generate its successors. Bacchus *et al.* [BK95] demonstrate that control strategies can be a considerable aid in speeding up the planning up to twenty times in TLPlan compared to planning without search knowledge. In this work, however, the focus was on performance of the goal transformation algorithm itself and the planner was used without control knowledge.

Two different approaches in the selection of transformations in `GoalMorph` were compared. The first one applied a *random* search algorithm, where transformations were selected in a random way. The second one was *utility-driven* mode, in which the goal transformation algorithm was run for each failed goal until a transformed goal with the highest utility that could be solved was found. If none of the transformed goals could be solved the `GoalMorph` algorithm was set to terminate after two seconds.

4.3.1 GoalMorph effectiveness

The effectiveness of the goal transformation algorithm was evaluated by comparing the transformed and original `composition requests`. Firstly, the size of the original and transformed `composition requests` was measured and compared. Secondly, the utility of the transformed `composition request` and original `composition request` was measured and compared. The effectiveness of `GoalMorph` was evaluated both in random search mode and utility mode for selecting transformations. It is important to note that when the planning system fails, the achievable size and utility of the original `composition request` are

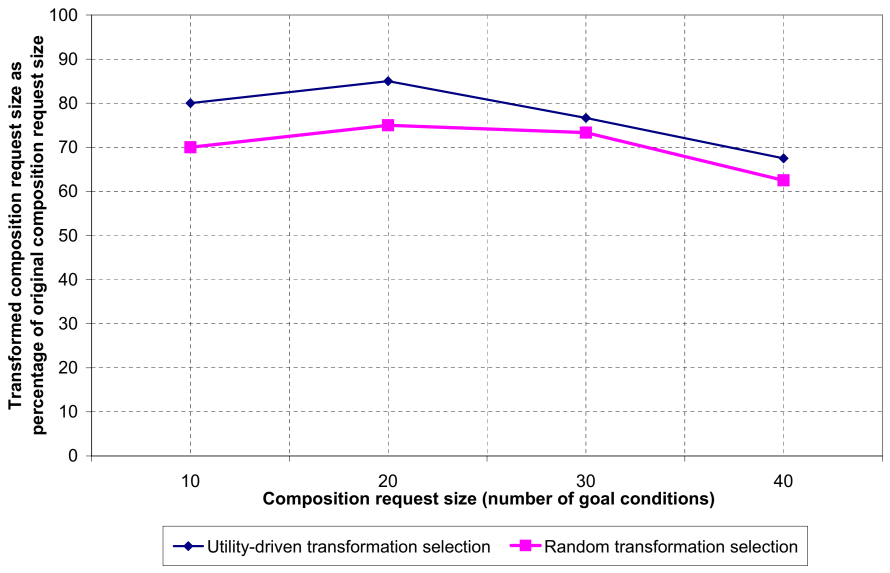


Figure 4.15: Percentage of achievable transformed composition request size (number of goal conditions) as a function of original composition request size

both considered equal to zero, as the request is only regarded satisfied by the planning system if all goal conditions are met.

Size of the transformed composition request. Firstly, the number of goal conditions that can be satisfied after goal transformation was considered. This was compared to the total number of goal conditions that are to be satisfied. Figure 4.15 shows the results of this experiment. Both random-search and utility-driven goal transformation found goals that could be solved and retained at least 60% of the size of the original `composition request`.

As expected, the lowest reduction occurs when utility-driven transformations are applied. However, even the randomised transformation selection algorithm is an improvement to the planning with the original request only, as the `composition request` can be partially satisfied. Sometimes transformed `composition request`s may even be larger than the original requests. That can be due to context unfolding, the operation that may introduce new, additional goal conditions.

Figure 4.14 shows the sample `composition request` and transformation of this request, for Case 2 of the usage scenario. In this specific experiment, the transformation request retained 100% of the size of the original request. All failed goal conditions were substituted by transformed ones.

Utility of the transformed composition request. To evaluate the utility of the transformed `composition request`, a model, representing partial fulfillment of the original `composition request` as well as individual goals, was employed.

Haddawy *et al.* [HH93] separate atemporal goals, which describe what needs to be achieved, into goals with symbolic and quantitative attributes. For example, a symbolic goal would be `(restaurant_found spanish)`, denoting that the restaurant must serve Spanish cuisine. An example of a quantitative goal would be `(restaurant_booking_made 3 1300)`, representing that table in the selected restaurant should be made for 3 people. Haddawy *et al.* further define a degree of satisfaction function (DSA) for a symbolic atemporal goal. This is defined in terms of an application-supplied sequence S of mutually exclusive goal literals g_1, g_2 ... g_n, such that g_n is the actual component of the goal and g_i represents a greater degree of satisfaction than g_j if $i < j$. DSA is in the range [0.0-1.0], where 0.0 is representing no satisfaction and 1.0 is full satisfaction of the goal literal.

`GoalMorph` uses this utility model for atemporal goals to reason about the extent to which each goal is satisfied. Goal ontologies and utility values from the `Context Mesh` and the `Goal Service` provide a base for devising a function for *core* and *context* goals respectively, specifying partial satisfaction of atemporal goals.

The overall goal utility is evaluated by measuring the sum of the utilities of all goal literals. Figure 4.16 shows the utility of the `composition request` after random and utility based goal transformations, when varying the problem complexity from 10 to 40 goals. Reduction in goal utility with goal transformation does not increase with the number of goal conditions introduced. This is due to the fact that there is a higher probability of successfully transforming goals with a higher number of goal conditions. It is important to note that higher goal size may not necessarily imply a higher utility. Depending on user preferences, shorter goals may incur higher utility.

4.3.2 GoalMorph performance

Transformation time This experiment compared the time that it takes the planner to find no solution for the `composition request` to the time it takes `GoalMorph` to transform the failed `composition request` into one that can be

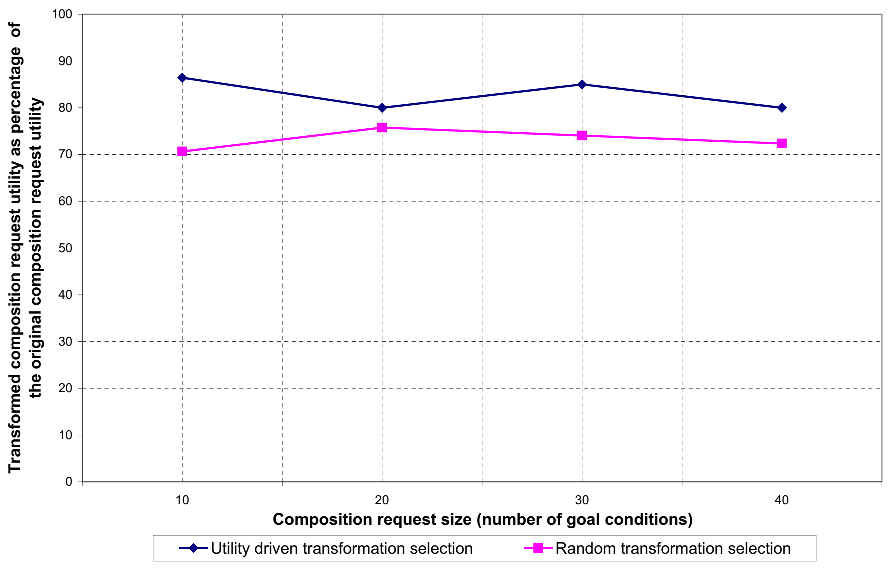


Figure 4.16: Percentage of achievable utility of transformed goal as a function of original goal size

solved and to replan. Figure 4.17 shows the results. As expected, the time difference increases with the number of transformations. However, the transformation time remains sufficiently small, less then 1600ms for goals of size 40, to justify the overhead introduced by goal transformation.

It is important to mention that planning time can be manually bound. Therefore, the framework can regulate how much time it would allow for the planning process to fail, before triggering `GoalMorph`.

Core and context goal transformations performance This experiment considered the transformation time for core and context aware goals and their impact on the overall transformation time. As core and context goals may be dependent, only core and context aware goal transformations that can be performed in isolation were compared.

Context aware goal transformation is expected to take longer than core goal transformation for a number of reasons. Firstly, the communication with the `Context Mesh` and the context layering process are computationally expensive.

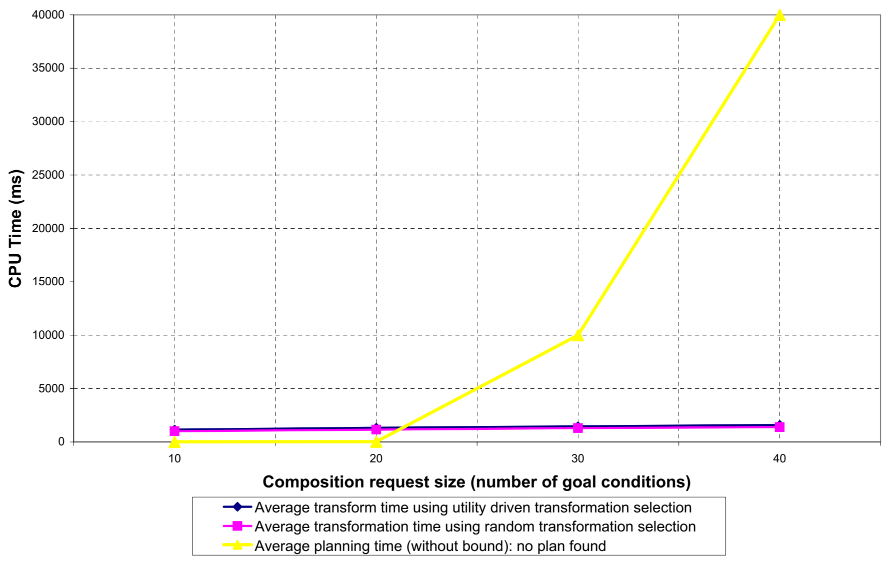


Figure 4.17: Total time to solve a transformed request compared to the planning time of failed request

Interaction with the `Context Proxy` to generate new context goals, as well as interfacing with the `Context Service` to retrieve historical values, can introduce further overhead. Finally, the latency of context goal transformations depends on a number of parameters, such as the number of related context types, the size of the context ordering, and the number of different hierarchies of each context type.

Figure 4.18 shows the results obtained when measuring context and core goal transformations. Total transformation time shows that the `GoalMorph` modifies composition requests efficiently. As anticipated context goal transformation takes longer than core goal transformation, due the more comprehensive ordering structure and communication overhead among `GoalMorph` components, such as the `Context Mesh` and the `Context Service`. The oscillations in the latency result from the different complexities of the context goal conditions used and the corresponding context orderings.

Finally, the time `GoalMorph` spends on generating possible transformations can be bound, as well as the number of transformations that will be performed.

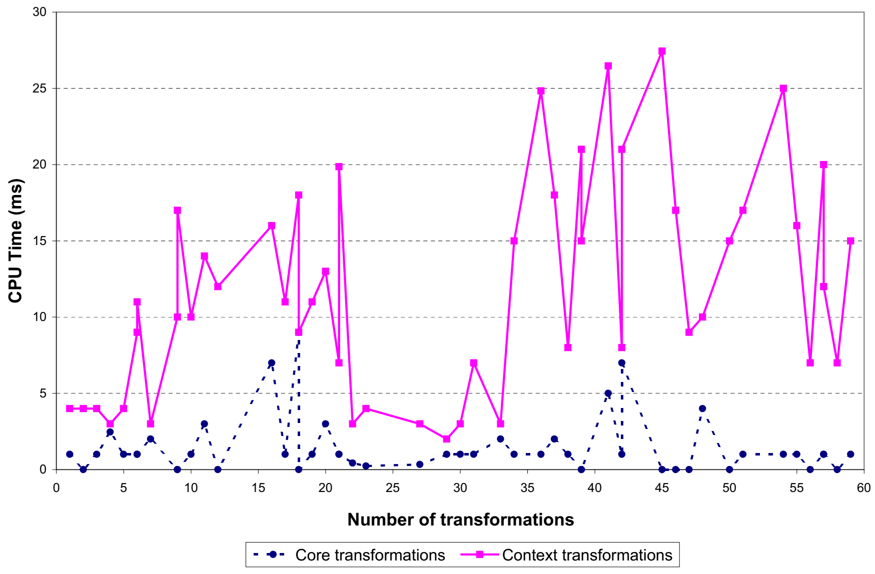


Figure 4.18: Performance of utility-driven core and context goal transformations as number of transformations increases

Furthermore, the planning problem can be solved sufficiently fast, `GoalMorph` can be extended to generate a number of modified requests at the same time and present the alternatives to the user.

4.3.3 GoalMorph scalability

In this experiment an increase in the goal size was simulated and the system's behaviour was observed in terms of the number of transformations generated and the running time of `GoalMorph`. Figure 4.19 shows that the system scales well, being able to generate up to 240 core and context transformations in 0.4 seconds in the random transformation selection mode. As expected, the utility driven transformation selection mode has a higher running time, due to the search for the transformed goal with the highest utility, whereas the random search algorithm terminates once the first feasible goal that can be solved is found.

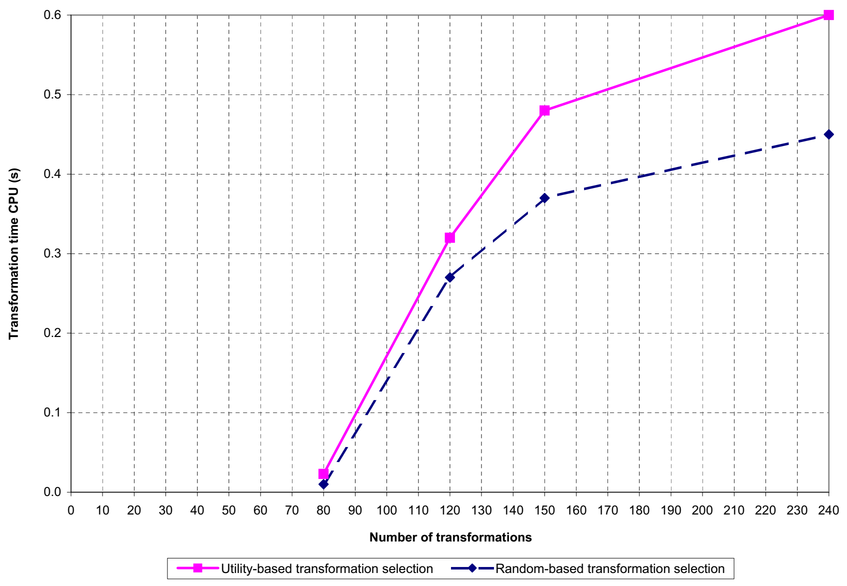


Figure 4.19: GoalMorph performance as a function of number of transformations

4.4 Related work

Most previous work in goal oriented service composition [MS02, WSH$^+$03] has assumed a static environment where plans can always be solved. In addition they focus on handling service execution failures, by replacing failed service instances. In contrast, this work generates solvable goals even when service replacement is not adequate, for instance, when no services that would satisfy some parts of a `composition request` are available. This section compares the `GoalMorph` solution to the GTrans [CV98] goal transformation system and discusses previous research in the area of partial goal satisfaction.

4.4.1 Comparison with GTrans

Cox and Veloso [CV98] introduce a taxonomy of goal transformations based on an organisation of goals and objects in a goal hierarchy, as an approach to planning in a world under continual change. They extend PRODIGY [VCP$^+$95], a state space nonlinear planner, to select the appropriate goal transformations automatically in response to world changes, in order to completely solve the transformed problem. Based on this work, Cox and Zhang [CZ04] later develop a system called GTrans, which applies goal transformation approach to mixed-initiative planning.

`GoalMorph` extends the ideas presented in GTrans by introducing the `Context Mesh` to enable context aware goal transformation. This allows the system not only to transform the actual goal, such as abstracting the goal from finding a Spanish restaurant to any closest restaurant or substituting Spanish with Mexican restaurant, but also to transform context goal conditions, to allow for satisfaction of the next "nearest" goal in the "nearest" context. This approach also enables the user to specify importance measures across the context types and hierarchies. `GoalMorph` differs from GTrans in a number of ways.

1. *Selection of goal transformations.* When selecting applicable goal transformations, beside the commonly used unguided search, GTrans relies on domain control knowledge. Cox and Zhang [CZ04] extend GTrans to allow users to establish and transform goals manually through visual representation. `GoalMorph` provides a utility model for goal contexts and their corresponding transformations, in addition to a domain control search strategy and unguided search using randomised algorithm.

2. *Automated transformation.* The user explicitly performs goal transformations in the mixed-initiative version of GTrans. By contrast, the `GoalMorph` solution enables the user to provide feedback on the usefulness of a particular solution after goal transformation and correspondingly updates the utilities of performed goal transformations. The design of `GoalMorph` allows the system to be extended to include the user directly in the goal transformation loop, however the work presented in this dissertation is focusing on *automated* goal transformation for providing an improved user experience.

3. *Planner independence.* Cox *et al.* integrate goal transformation with the PRODIGY planning system, which reasons about both multiple goals and multiple alternative operations from its domain definition. `GoalMorph` on the other hand, is planner independent. It uses internal representation for goal conditions, their hierarchies and utilities. `GoalMorph` is the general-purpose solution. It is planner-independent and can be integrated with multiple composition methodology.

4. *Replanning.* GTrans interleaves goal transformation with the actual plan refinement process. `GoalMorph` is planner independent and therefore it is

designed in such a way to rely on replanning. Goal transformations can be performed incrementally from the failed plan or completely from scratch starting with the original `composition request` definition.

5. *Configurable hierarchy types.* GTrans allows for goals, their predicates and arguments to be organised along four key hierarchies: abstraction hierarchy, number line, enumeration set and patronomy. `GoalMorph` extends this to allow for any arbitrary hierarchy to be included. This is achieved by creating a custom interface that imposes a total or partial ordering on the elements of each hierarchy.

6. *Goal prioritisation.* GTrans at present assumes that all goals have the same priority. However, the authors propose as part of the future work to allow the planner to explicitly assign relative priorities to the goals. In the prototype implementation of `GoalMorph`, core goal conditions are transformed first, followed by context goal conditions. However, the goal utility model allows for each goal condition to be prioritised thereby guiding the selection of goal transformations. For example, Miles may express a preference that having directions in synthesised speech form is more important than ensuring that a restaurant serves Spanish cuisine. The system would then perform the transformation of higher ranked goal conditions first. During the lifetime of the system these rankings can be refined through interaction with the user.

4.4.2 Partial satisfaction planning

The `GoalMorph` goal transformation algorithm is related to the area of Partial Solution Planning (PSP). In PSP the planner is not required to completely satisfy all goal conditions. Instead it focuses on achieving the best subset of goals given the resource limitations. PSP relaxes hard-goal constraints from classical planning and associates them with utility values. To solve the problem of partial goal satisfaction, two approaches have been followed: *plan refinement*, where the output of planning is adapted, and *goal refinement*, where the goals are changed to produce a solvable problem.

Haddawy and Hanks *et al.* [HH93] proposed goal-directed utility models to enable decision-theoretic agents to measure plan success in terms of its preferences or a utility function that respects some of those preferences. They developed an extended goal model consisting of atemporal and temporal components, which are goals that define *when* something has to be achieved. The model allows for representation of partial goal satisfaction and provides a structure for representing its utility. Haddawy *et al.* use the goal's utility to establish whether one plan has a higher expected utility than another. Haddawy and Suwadi [HS94] later employ this model to address the plan generation and refinement problems. They develop a *Decision theoRetIc Planning System* (DRIPS), where they incorporate utility and goal structures. One of the limitations of this model is that neither goal nor utility structures provide mechanisms for goal prioritisation.

Williamson [Wil94] built PYRRHUS, a decision theoretic planning system that finds an optimal plan, using a goal and utility structures previously introduced by Haddawy *et al.* Williamson focuses exclusively on Partial Satisfaction of the Temporal Component (PSTC) goal and the corresponding utility model. He uses PSTC to extend the notion of plan quality to take into account partial satisfaction of the goal and the cost of resources used by the plan. PYRRHUS is built as an extension of the partial order causal link planner UCPOP [PW92], which is in turn based on the Systematic Nonliner Planning system [MR91] that uses causal links. PYRRHUS uses the PSTC utility model to devise domain specific heuristic knowledge (in the form of an increasing utility function on each of the goals) for refinement of the plans. The advantage of PYRRHUS is that it relies on the same heuristics as any goal satisfying planner. Also, PSTC supports only partial satisfaction of temporal goals, and not for the atemporal goals. Furthermore, while the PYRRHUS system allows for partial satisfaction planning with goal utilities, it still requires for all the goal conjunctions to be reached.

van den Briel et al. [vdBNDK04] focus on PSP to satisfy only a subset of goals, while ensuring that the resources in the planning problem are not overloaded. They introduce a method for modelling and handling plan quality and devise a taxonomy of partial satisfaction problems, which allows for differentiating between feasible and optimal plans. In addition they categorise goals in terms of whether they are completely or partially satisfied. van den Briel *et al.* develop and compare integer programming, regression planning with reachability heuristics and anytime heuristic search approaches to the PSP. They show that heuristic planners are comparable to the quality plans generated by integer programming and provide a practical solution. Their solutions, however, do not support temporal goal components. There is no prioritisation of goals and no support for interacting goals, they assume goals to be mutually exclusive.

4.5 Summary

Providing context aware service composition facility raises two issues: dealing with composition failures and specifying context behaviour.

Dynamic computing environments involve contextual changes which may cause the service composition to fail. This chapter has presented `GoalMorph`, a composition failure management system, which uses context aware goal transformation to facilitate fault tolerant, context aware, service composition. The central component of the `GoalMorph` is the `Context Mesh`, a multidimensional data structure for hierarchical organisation of context. The `Context Mesh` enables context layering, the process of controlling the amount of context data used to transform the context goal.

Experimental evaluation of `GoalMorph` demonstrates that context aware goal transformation is effective in producing solvable alternatives of `composition requests` that cannot be originally solved. `GoalMorph` generates partially satisfied goals, which achieve more than 60% of the original utility, despite the increase in the goal size in the example environment.Additionally it has shown that the overhead introduced by `GoalMorph`, to the service composition framework is minimal. It can perform up to 240 core and context transformations in 0.4 seconds whilst randomly selecting transformations.

The `GoalMorph` solution was compared to the GTrans goal transformation framework. By contrast to GTrans, `GoalMorph` provides automated goal transformations. It accommodates arbitrary transformation selection methods, such as unguided search and utility based search. `GoalMorph` performs goal transformations independently of any planning technology. It allows multiple, customisable hierarchies for ordering goals and arguments. The chapter has also discussed the current research efforts in partial goal satisfaction.

Finally, this chapter has also presented `cogotags`, an XML representation that allows for flexible and dynamic construction of context aware goals. This removes the need for pre-built contextual dependencies in the `Goal Service`. `Cogotags` consist of the context type and value that trigger invocation of the specific goal condition. They allow for injection of goal conditions originating from the `Context Service`, the `Context Proxy` and the user's profile.

Chapter 5

Implementation

Chapter 3 discussed the layered design of the proposed service composition infrastructure, based on four key operations: composition request management, abstract service composition, architecture specific service composition, and execution and monitoring. Chapter 4 presented the implementation details of the composition request management layer, and proposed the `GoalMorph` system. This chapter describes the internal implementation of the components of the remaining layers.

Section 5.1 presents the implementation It compares a number of existing solutions in terms of their applicability to the Web service composition problem and discusses why Web service composition needs more than what conventional planning systems provide to assemble composite services. This section then describes how the prototype implementation applies goal-oriented inferencing from the TLPlan [BK95] planning algorithm to select atomic services that form a composite Web service.

Section 5.2 discusses the implementation details of the architecture specific composition layer. It presents how the `Plan Instantiator` processes an `abstract execution plan`, which is described in Business Process Execution Language For Web Services (BPEL4WS) [CAD+05] format, and mediates the service discovery and instantiation process.

Finally, Section 5.3 presents the internals of the execution and monitoring layer. It describes how the `Monitoring Engine` uses `monitoring procedures` to verify service execution, continuously update the state of the `Execution Engine` and handle execution failures.

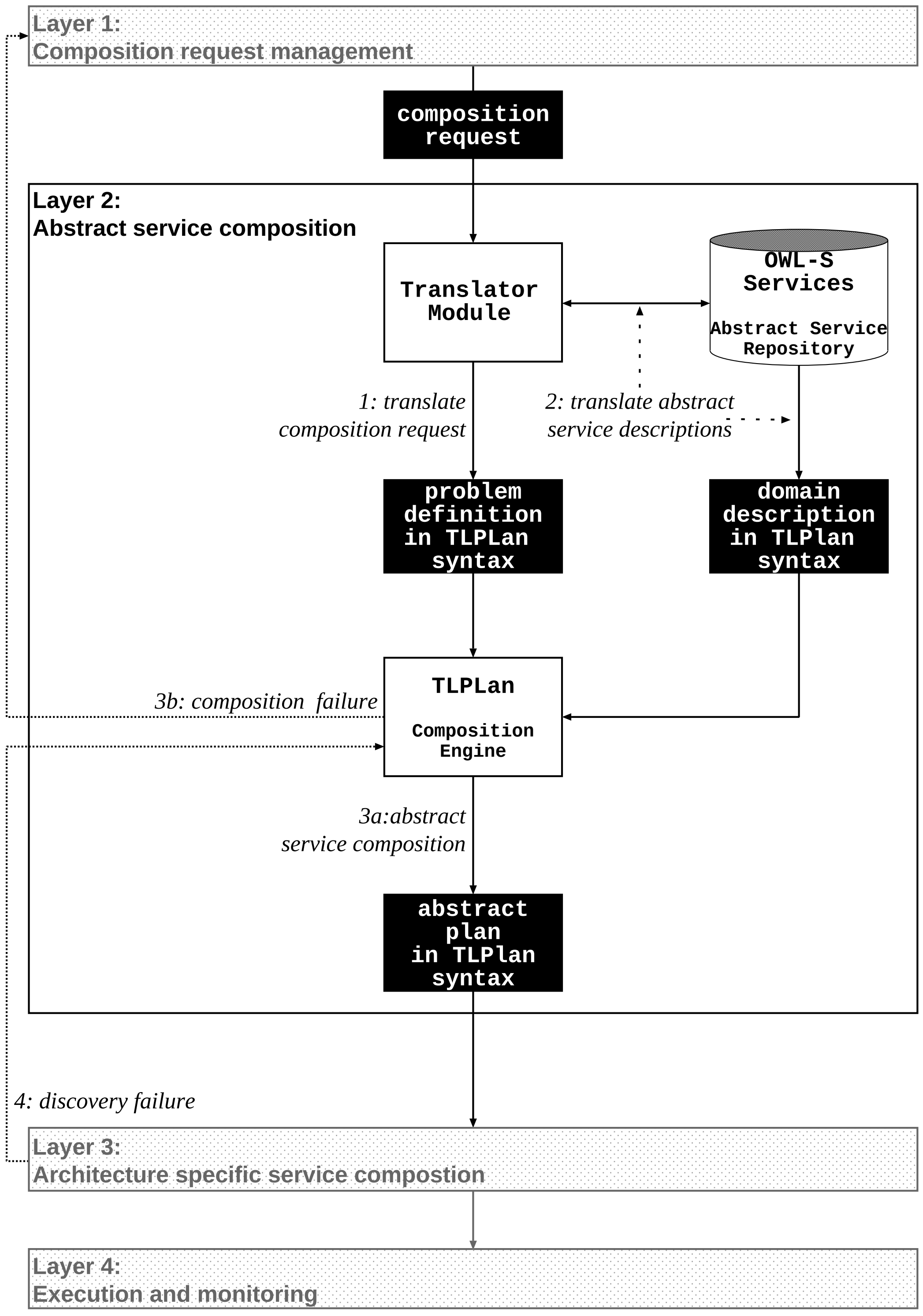

Layer 1:
Composition request management
composition request
Layer 2:
Abstract service composition
Translator Module
OWL-S Services
Abstract Service Repository
1: translate composition request
2: translate abstract service descriptions
problem definition in TLPLan syntax
domain description in TLPLan syntax
TLPLan
Composition Engine
3b: composition failure
3a:abstract service composition
abstract plan in TLPlan syntax
4: discovery failure
Layer 3:
Architecture specific service compostion
Layer 4:
Execution and monitoring

Figure 5.1: Implementation of the abstract service composition layer

5.1 Abstract service composition layer

Once the `composition request` has been assembled in the composition request management layer, the `Composition Engine` proceeds to construct a composite service. Figure 5.1 describes the abstract service composition process, in which a composite service is assembled from abstract services format.

This section analyses the ability of several planning technologies to handle the Web service composition problem. It discusses why Web service composition needs more complex features than those normally provided by planning technologies. Finally, it proposes the use of the TLPlan planner in this framework to address some of the challenges.

5.1.1 Abstract Service Repository

The `Abstract Service Repository` is a directory that stores and manages abstract services, which carry high-level information about their functional capabilities and service categorisation and cannot be invoked. Conceptually, abstract services can be seen as analogous to abstract classes in object-oriented languages.

The abstract service composition layer assembles abstract services to construct `abstract plans`. The use of abstract services has two purposes. Firstly, grouping the service instances and organising them by the type of operation they provide reduces the size of the `domain description`. This makes the search more efficient in comparison to having domains that include all possible instances. Secondly, by utilising abstract service descriptions and producing an `abstract plan`, the replacement of a failed service by another one of the same type is straightforward. The abstract service description forms the search criteria to be used in service instance discovery. When service execution fails these search parameters are used to locate and schedule a replacement service for invocation.

In the prototype framework implementation, it is expected that a domain engineer creates and submits abstract service descriptions in OWL-S format, modelled as `SimpleProcess` structures, to the prototype `Abstract Service Repository`. Abstract service descriptions are then translated into the representation format supported by the `Composition Engine`. Later in the process of service composition they are bound to instances by `Plan Instantiator`, described in Section 5.2.

At present the OWL-S syntax is used purely for *representation* of the abstract services, and no reasoning is performed over the service semantics. The OWL-S format has been selected, because it is becoming a maturing and prevalent means of specifying Web service behaviour, as well as for its portability. Consequently, issues arising from weaknesses of OWL-S with respect to semantic reasoning are not being addressed by this dissertation. For example, supporting functional descriptions in OWL-S is an important challenge as there is a lack of means of describing the relationships between inputs and outputs. Hull et al. [HZB+06] have proposed a formalism for explicitly describing how service inputs and outputs are related. They use an OWL ontology to fix the meaning of terms used in service descriptions. Their approach is designed to be integrated with OWL-S (or WSMO), allowing for automated reasoning approach for matching services.

Finally, if the proposed composition framework employed a language that did not support OWL-S syntax, extending the system later to incorporate semantic Web services would incurr a higher overhead. The ability to import services with OWL-S syntax in the current prototype represents a first step towards utilising semantic descriptions.

5.1.2 Composition Engine

This section describes internal details of the composition process and evaluates the applicability of planning technology to the Web service composition problem. In the scope of the framework implementation it also presents the use of the TLPlan planning system.

AI Planner requirements

Goal-oriented inferencing from planning technologies, defined in Section 2.2.5, can be applied to the Web service composition problem, if the following two observations are made. Firstly, the high-level description of a user's task can be mapped to a planning problem definition. Secondly, Web service descriptions can be mapped to action descriptions using a planning language.

Web service composition requires a number of sophisticated facilities in the planning system, so that goal and application domain models are realistic, complete and comprehensive. This section examines several features that planning systems should support. This list is not exhaustive, however it represents a set of features desirable for planing-based Web service composition to succeed.

1. *PDDL level.* PDDL [GHK+98], described in detail in Section 2.2.5, is the standard language for describing planning domains and problem specifications, designed to enable planner inter-operability. PDDL 2.1. is based on Action Description Language (ADL) [Ped94]. It provides constructs for expressing temporal planning domains separated into different levels of expressiveness. Level 2 allows numerical constructs, that enable testing and update of the values of numerical variables. Levels 3 and 4 provide support

for explicit representation of time and duration for discrete and continuous actions respectively. Service execution takes time, particularly for long transactions such as those found in the business domain, and requires support for actions with continuous effects. Therefore support for all PDDL levels is needed.

2. *Sequences.* Structured composite services prescribe the order in which services are executed. For example, the user will first select a restaurant and then the directions service should identify the driving route, based on the current location of the user and the location of the selected restaurant.

3. *Iteration.* Often a certain service may need to be repeatedly invoked to obtain results successively closer to a desired result. This requires an iteration control construct. For example, a driving direction service will be continuously invoked every time there is a location update, until the user reaches the desired location.

4. *Concurrency.* Planners typically allow for service instances to be sequentially ordered. More complex, reactive processes require more advanced mechanisms to handle concurrency. For example, a user may want to a book a table at a restaurant and at the same time obtain driving directions to it. Such a request can be satisfied by executing two services simultaneously, thus requiring a concurrency construct.

5. *Conditional.* Service effects often depend on the input provided. Conditional constructs can be used to define pairs of conditional preconditions and postconditions. For example, the conditional effect of the `BookRestaurant` service is `(restaurant_booked)`, when `(restaurant_has_space)`. As a conditional operator forms an expression, it can also be used to choose an applicable service for execution depending on the condition.

6. *Nondeterminism.* The assumption of deterministic behaviour in planning with Web services in dynamic computing environments is untenable. Web services may have multiple outcomes, many of which cannot be predicted. For example, a directions service may stop functioning, or the result generated by the restaurant service may not be satisfactory to the user. A facility for nondeterminism is necessary to provide a realistic model of the environment.

7. *Plan optimisation.* In a realistic deployment, services consume resources, such as network bandwidth, and have a monetary cost associated with

their execution. Therefore a mechanism is required to impose metrics and resource constraints on each service as well as the resulting plan, thus allowing for plan optimisation.

8. *Extended goals.* Users may need to express conditional preferences for different goal conditions comprising their task intention. For instance, a user may want to specify that a composite service should try first to reserve and confirm both a restaurant and a cinema from two different service providers. If one of the two services is not available, or there is no availability at the same location, it should fall back and cancel both reservations. The ability to describe complex conditions enables users to place requirements on the behaviour of processes, and not only on their final state.

9. *Partial observability.* The planner may not have complete knowledge of the application domain. For example, a restaurant booking service has as a prerequisite that a certain number of seats must be available. However, the planner does not have any information about seat availability. Mechanisms for dealing with this problem of partial observability are necessary. Common methods for handling this include interleaving of planning and execution and using knowledge gathering actions, commonly called *sensing*, to update the application domain while planning.

10. *Availability for multiple platforms.* When integrating an existing planning system it is necessary to know its availability on different platforms, such as Linux and Windows.

11. *Support and maintenance.* For a specific planner to be used in deployed composition systems, the level of support is an important criteria for selection. It is necessary that the planner employed is actively used and maintained.

12. *Source code availability.* To optimise the planner for solving specific problems, it may be of interest to tweak the internal planning algorithm. For that purpose, access to the source code is required, to define clean interfaces, descriptions, and facilitate customisations.

Comparison of planning systems

There are three broad divisions of planners, based on the type of problems they solve [GNT04]. The first group is the classical planners, which deal with complete information and deterministic dynamics. The second group of planners

	Planning system			
Properties	**SHOP2**	**TLPlan**	**MDP**	**MBP**
Planning methodology	HTN	Forward chaining with control knowledge	Markov decision processes	Model based planning
PDDL 2.1.				
Level 1: ADL planning	✓	✓	✓	✓
Level 2: Numeric	✓	✓	✓	✓
Level 3: Discrete time	*	✓	*	✓
Level 4: Continuous time	*	✓	*	✓
Control constructs				
Sequences	✓	✓	✓	✓
Iteration	✓	✓	✓	✓
Concurrency	✓	*	✓	✓
Conditional	✓	✓	✓	✓
Features				
Nondeterminism	*	*	✓	✓
Plan optimisation	✓	✓	✓	✓
Extended goals	×	✓	*	✓
Partial observation	*	*	*	✓
Availability for multiple platforms	✓	Linux only	✓	Linux only
Support and maintenance	*	*	*	*
Source code availability	✓	×	*	×

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

Table 5.1: Suitability of planners for Web service composition.

can support problems with complete information and nondeterministic dynamics. Finally, the third group handles problems with incomplete information and nondeterministic dynamics.

Table 5.1 shows the extent to which two deterministic planners Simple Hierarchial Ordered Planner 2 (SHOP2) [NMAC$^+$01] and TLPlan [BK95] in the first group, and two nondeterministic planners, Markov Decision (MDP), in the second group, and Model Based Planner (MBP), in the third group, meet the specific technical requirements to provide automated Web service composition.

SHOP2. Nau *et al.* [NMAC$^+$01] devise SHOP2, a hierarchical task network (HTN) planning system, whose objective is to perform a set of tasks, rather then achieve a set of goals. SHOP2 uses HTN methods as its control knowledge. These methods describe how to decompose an abstract task into a group of primitive operators that form the plan implementing the task.

Wu *et al.* use SHOP2 to automate service composition [WSH$^+$03] However, as SHOP2 relies on composite service templates based on HTNs, improvements would be required to apply it to Web service composition in dynamic environments. SHOP2 assumes that the state of the world is always accessible, static and deterministic. In addition, all method descriptions are assumed to be complete and correct, and to precisely describe all the possible effects.

Although SHOP2 does support a limited form of nondeterminism through conditional expressions, this is impractical for context aware systems, where the range of data is more extensive. It is not practical to enumerate all the possible conditions that must be accommodated, as this would grow exponentially with the number of steps in the plan.

SHOP2 supports actions of at least Level 2 in PDDL, however, it does not directly allow for PDDL formatted domains to be imported.. Even though it does not provide explicit support of actions with duration in Level 3 of PDDL, called *durative* actions, it has sufficient expressive power to represent concurrency and durative actions. Its operators can assign values to variables and do numeric calculations.

The HTN approach gives more structure to domains and the way goals should be solved. However, in SHOP2 goals can not be stated declaratively. Hence SHOP2 has to know in advance which HTN method it should call, making it impractical in dynamic context aware scenarios. Consequently the planner cannot solve a completely new, unknown problem for which no method definition exists.

In their more recent work, Sirin *et al.* [SPH05] integrate a description logic (DL) reasoning with an HTN planning system to construct a HTN-DL formalism. This is used to generate compositions of Web services. Web ontologies are used to write service template descriptions that will allow flexible matchmaking of services. Each abstract service is described and also preferences in templates as to which instances are of greater applicability are described.

There are two components of a HTN-DL domain. The first one describes the planning domain and contains the operator and method descriptions. The second one is a DL knowledge base that contains task and preference descriptions.

Such a solution can be employed in our service composition framework to efficiently recognise relationships between different goals, select applicable goal predicates given the requested tasks and to reduce the number of predicates that will be considered, such as cuisine types.

TLPlan. Bacchus and Kabanza [BK95] develop TLPlan, which uses domain specific search control information to guide the search algorithm. TLPlan is based on simple forward chaining search, in which planning actions are applied to the current state to generate its successors. TLPlan therefore knows the current state of the world at every step of planning process. Control rules, which are written in temporal logic, provide domain specific knowledge to inform the planner which states should be avoided, therefore allowing the planner to backtrack and try other paths in the search space. Bacchus and Ady [BA01] extend TLPlan to handle concurrent actions with variable duration.

TLPlan employs a representation language, which is expressive up to Level 4 of PDDL. It is capable of reading a problem definition and generating the plan in PDDL, however it does not support PDDL-based domain specification. TLPlan supports all control structures involving concurrency, iteration and non-deterministic choice to construct complex and expressive composition processes. Furthermore, it allows for concurrently executing actions with varying durations.

Both TLPlan and SHOP2 support a limited form of nondeterminism through the use of conditional actions. Kutur and Nau [KN04] propose a technique for adapting TLPlan and SHOP2 to work in nondeterministic domains. In our prior work [VR05c] we compared SHOP2 and TLPlan in more detail.

Planning based on Markov Decision Processes. Markov Decision Process (MDP) based planners are probabilistic; they create conditional plans only for the contextual situations that are most likely to occur [Put94]. A domain is defined as a set of states and actions and uses a probability function to model the uncertainty about action outcomes. Goals take the form of utility functions, which guide the selection of actions. As a result, the planning process is an optimisation problem: searching for a plan that maximises the utility function specified in the goal. The resulting plan is a policy that specifies the actions to be applied in each state.

Doshi *et al.* [DGAV04] employ MDP to model the problem of workflow composition in a supply-chain scenario. Solution of MDP is a policy that guides a composite service towards its goal. The policy assigns an action to each state of the workflow that is considered as optimal at that point, based on past interactions with the service. Doshi *et al.* extend this approach, by interleaving it with a model that learns the probability of a certain service being satisfied.

Although this method generates fast responses to most contingencies, it may miss potential opportunities that arise from changes in the world environment. For example, a service producing a result in an unexpected format may not be handled adequately. Another limitation is that the size of the state space grows exponentially with the number of features describing the problem, resulting in what is known as the *state explosion* problem.

Model Based Planning. This is a nondeterministic method based on the exhaustive exploration of finite state automata, where actions may have multiple outcomes. To support extended goals, it uses temporal logic formulas to express the set of goal states and the conditions for the final plan execution. The planner uses a state transition system and a temporal formula to generate plans that control the system evolution so that the system's behaviour makes the temporal formula true.

Pistore *et al.* [PBB$^+$04] employ the Model Based Planning [BCP$^+$01] to solve a Web service composition problem in a retail domain. Their system allows for nondeterminism in the initial state and in the outcome of action execution. It can model planning domains with different degrees of run-time observability. Available services are partially specified, and the degree of observability on the current state varies from "full" to "null" observability. Full observability is achieved when the current state is completely specified. Partial observability occurs when only partial information is available. Finally, null observability occurs when no information on the current state is available.

Each service is represented as a nondeterministic finite state machine, characterised by a set of initial states and by a transition relation that defines how the execution of each action leads from one state to a new set of states.

Pistore *et al.*'s approach creates monitors to trace the execution of external processes, in contrast to the approach of Doshi *et al.*, which assumes that monitoring is unnecessary because MDP can deal with any contingencies. Modelling realistic problems using both MBP and MDP may result in a large number of states and trigger state explosion. Current efforts in symbolic representation aim to overcome this problem by employing compact representations of finite-state models. As a result, model checking is performed by exploring sets of states, rather than individual ones.

MBP uses NuPDDL, a language equivalent to PDDL 2.1, which can handle functions, conditional effects, and quantifiers. It also allows for arbitrary nesting of conditional effects and quantifiers. Most existing MBPs can take as an input standard deterministic domains in PDDL [BCP$^+$01]. To model nondeterministic behaviour additional control knowledge, however, must be devised.

Summary The review of planning technologies demonstrates that not all of the identified features required for Web service composition are present in a single planning system. Each planner has been designed and implemented to deal with different types of problems.

Automated domain construction is the main challenge for all reviewed planning systems. While hand-coded search control does help both SHOP2 and TLPlan plan effectively, it incurs a significant overhead. It requires expertise in both the domain representation and the specifics of the planner, and therefore limits the possible extent of automation of the Web service composition process.

MDP and MBP provide flexible nondeterministic approaches to constructing composite services. However, the main challenge is again related to domain modelling, as these approaches may result in state explosion.

TLPlan for Web service composition

The prototype implementation of the framework adopts TLPlan, which supports all necessary control constructs, thus enabling complex Web service processes. It is freely available for Linux at the time of writing and supports PDDL 2.1 up to Level 4. Based on the comparison TLPlan is the simplest of the evaluated planners, which meets our main requirements.

This section describes how TLPlan is used to synthesise plans in the domain of our usage scenario, which has been described in Section 3.1. The fundamental steps in using a planning system include describing the planning domain, specifying the initial and goal worlds, and invoking the planning process.

Domain description. A `domain description` contains details about the literals, predicates and function symbols to be used in the domain. TLPlan takes them in a number of forms. Firstly, as *described* symbols, which are basic predicates and functions that get updated by actions. Secondly, as *defined* symbols, which are defined by first-order formulas. Finally, as *external* symbols, which are used to invoke external C routines. Once all of the symbols have been defined, the domain can be described using the first-order language generated by these predicate and function symbols.

Each literal is defined either as a predicate or function symbol of the domain by first-order formulas. The predicate definition consists of the *name* of the defined symbol, such as a function or a predicate, and *arity* that specifies the number of parameters accepted by the defined symbol. The corresponding world states are then described by the symbol name and arguments.

In our usage scenario symbols and predicates are needed to represent information about restaurant and other domain related concepts. For example, each restaurant has following properties address, cuisine type, and seating capacity.

Action	**Preconditions**	**Postconditions** (*adds*)
restaurant_finder	(restaurant_type r_type)	(restaurant_found restaurant_name)
address_finder	(business_name b_name) (city c)	(address_found) (location to)
direction_finder	(location from) (location to)	(directions_found from to)
translator(from, to, content)	(language from) (language to)	(translated_content language_to)
txt2speech(content)	(txt_form content)	(speech_out content)

Table 5.2: Sample planning actions for the usage scenario

Figure 5.2 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. Similarly the literal `(predicate restaurant_smoking 2)` indicates the restaurant's smoking policy, where the arguments are an identifier for the restaurant and its smoking policy. A defined literal `(predicate restaurant_has_space 2)` is used to determine if the booking can be made. Symbols declared to be defined must subsequently be given definitions, as shown in the last section of the figure.

The next step is to define actions in the domain, which in TLPlan is done using operators in either Stanford Research Institute Problem Solver (STRIPS) [FN71] language or Action Description Language (ADL) [Ped94]. They both generate first-order formulas that are evaluated in the current world to generate successor worlds. TLPlan operators are defined by lists of preconditions and postconditions, where *adds*, the list of facts that will be added upon execution of the operator, and *deletes*, the list of facts that will be removed upon execution of the operator. ADL operators are activated when the planning system evaluates the *add* or *del* clause in the operator. This in turn is controlled by the evaluator's rules for early termination. Functions can be used and updated by including a function specification inside an add clause, and ADL operators can specify a recursive set of updates by invoking recursive defined predicates in their sets of clauses.

Table 5.2 outlines some of the actions in the usage scenario, together with their preconditions and postconditions. Examples include a service that searches restaurants by cuisine type, an address finder, a driving directions service, a translation service, and a speech-synthesizing service. Figure 5.3 shows the operator definition for making the restaurant booking.

```
(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)))
```

Figure 5.2: Sample domain description in TLPlan

```
(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)
      )
)
```

Figure 5.3: Sample TLPlan operator

Problem definition. The problem definition specifies the initial world and the goal world, using lists of domain predicates and function definitions. Figure 5.4 shows a sample problem definition for the usage scenario.

The initial world describes the properties of the domain, and literals that hold true. For example, the literal `(activity driving)` describes that the user is currently driving.

Figure 5.4 shows the goal conditions `(directions_found current_address restaurant_address)` and `(direction speech_out)` as 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.

Planning goals serve two purposes. Firstly, they represent information about the planning problem, in our framework is a `composition request`. They provide criteria for delivering successful plans, consisting of goal conditions that must be satisfied. For example, a user request for driving directions is described by the conjunction of goal states in TLPlan syntax. Secondly, goals limit inference in the planning process. They guide the search algorithm and determine the applicability of operators.

```
; 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)
)
```

Figure 5.4: Sample TLPlan problem

Finally, TLPlan also allows for extended goals, where the goal can be represented as an arbitrary temporal formula. Furthermore, if every initial world in this domain has some special context specific features, such as one off predicates that should only be used in this run of the problem, or one wants to set up some additional described predicates and functions, TLPlan provides special commands to allow for customisation.

Plan. The planner is invoked by loading the `domain description` file and the `problem definition` file. TLPlan is based on forward chaining search and implements both depth-first and breadth-first search. It also supports the following variants of each search algorithm (a) use of operator priority, (b) disabling backtracking, which means that there is only one operator as successor for each world, and (c) use of a heuristic based on costs defined in the operator definitions to guide the search.

The resulting plan is a list of operators and a sequence in which they should be applied. Figure 5.5 shows the sample output given the problem in Figure 5.4.

```
(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)
```

Figure 5.5: Sample TLPlan plan

5.2 Architecture specific service composition layer

This section describes how symbolic planning action descriptions are turned into deployable service descriptions ready to be executed in a run-time environment. Figure 5.6 shows the implementation of the architecture specific service composition layer.

5.2.1 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.

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 5.7 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) [CCMW01] file.

Figure 5.8 shows the `abstract execution plan` in the BPEL4WS format. The partner `partner_user` represents interaction with the user. 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.

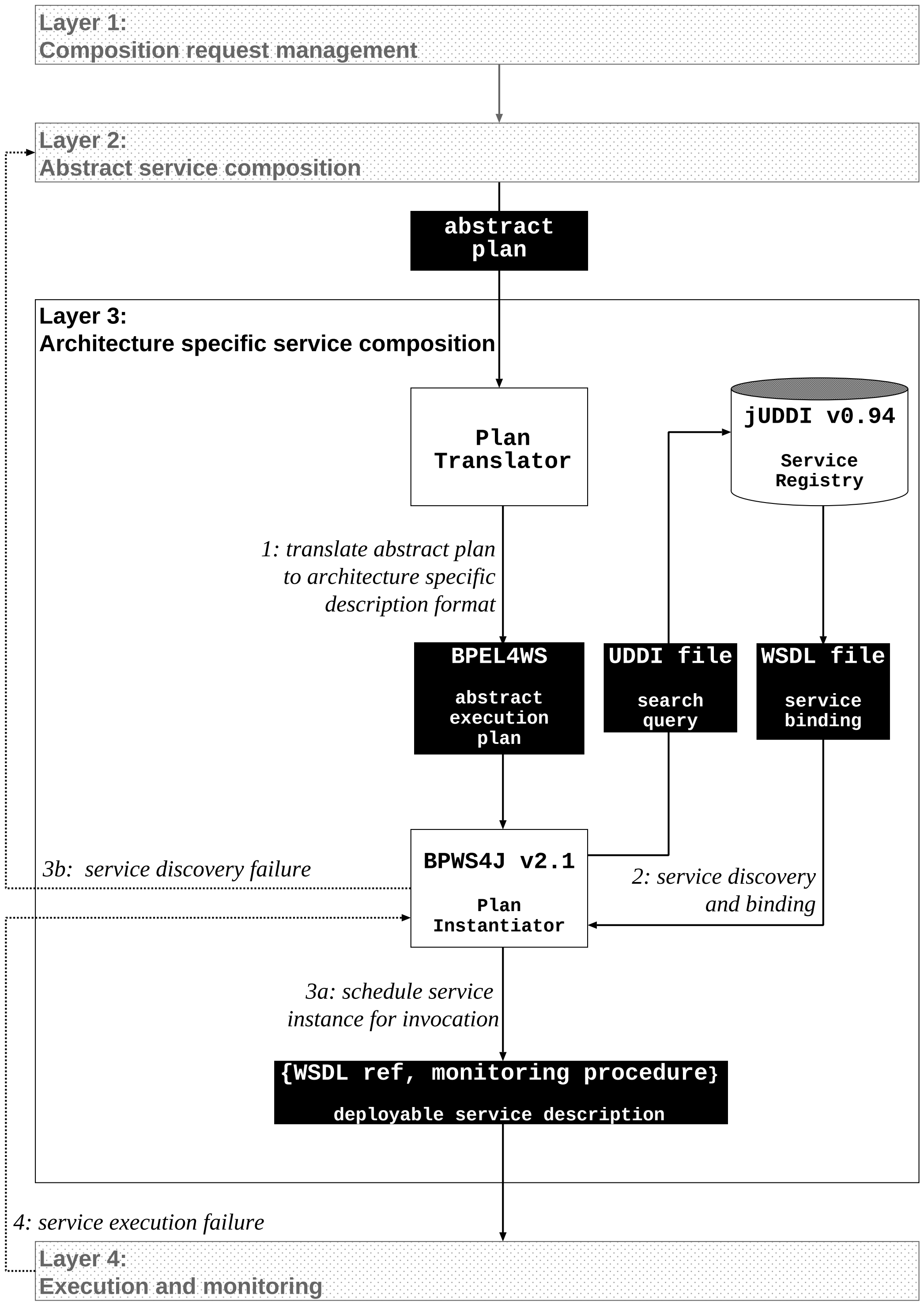

Layer 1:
Composition request management
Layer 2:
Abstract service composition
abstract plan
Layer 3:
Architecture specific service composition
Plan Translator
jUDDI v0.94
Service Registry
1: translate abstract plan to architecture specific description format
BPEL4WS
abstract execution plan
UDDI file
search query
WSDL file
service binding
3b: service discovery failure
BPWS4J v2.1
Plan Instantiator
2: service discovery and binding
3a: schedule service instance for invocation
{WSDL ref, monitoring procedure}
deployable service description
4: service execution failure
Layer 4:
Execution and monitoring

Figure 5.6: Implementation of the architecture specific service composition layer

```
<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>
```

Figure 5.7: Sample partner definition in Business Process Execution Langauge for Web Services

In the usage scenario, `partner_plan_instantiator_proxy` instantiates services corresponding to the operations, such as `getRestaurant` and `getDirections`. The `reply_RestaurantProcess` subprocess ends with a `<reply>` activity. This indicates that the process is to send a message to the partner `partner_user` in reply to a message that was received through a `<receive>` construct.

The `abstract execution plan`, described in BPEL4WS is deployed on IBM Business Process Execution Language for Web Services Java Run Time (BPWS4J) v2.1 [IBM04], 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 [Tom06] 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

```
<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>
```

Figure 5.8: Abstract execution plan in Business Process Execution Language for Web Services

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_plan_instantiator_proxy` partner, shown in Figure 5.8.

In our scenario, the process starts when the request for the operation called `getRestaurantDirections` from the partner `partner_user` has been received, indicated by the element `<receive>`, as shown in Figure 5.8. After this request has been received, a number of services are instantiated and invoked, which are represented by the `<invoke>` elements.

5.2.2 Plan instantiation

The `Plan Instantiator` processes the `abstract execution plan` and contacts the `Service Registry` to instantiate each abstract service, which is part of a composite service.

Service discovery. The `Service Registry` is a network based directory system that contains information about available service instances. It stores contracts from service providers and relays those contracts to interested service consumers. The `Service Registry` may also act as a proxy for the provider, enabling the client to interact with a single point of contact for all required services.

The framework employs Universal Description, Discovery, and Integration (UDDI), an XML-based standard for describing, publishing, and finding Web services. The `Service Registry` is realised using jUDDI [Jud03], an open source Java implementation of the UDDI specification for a Web service registry.

The `Plan Instantiator` constructs a search query based on an abstract service description. It submits the query, together with optional Quality of Service parameters, to the `Service Registry` to locate service instances.

Figure 5.9 shows a sample search request in UDDI format, which can be issued to locate `services` implementing a restaurant finder service. The search can be constrained by a specific `businessProvider`, which is identified by a `businessKey`. Additional `keyedReferences` can be added to the `categoryBag` as filters, to narrow the scope of the service descriptions that are returned in response to this search query. The services are matched if their `categoryBags`

```
<?xml version="1.0" encoding="utf-8"?> <soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <find_service
        businessKey="ADD69E70-8E5C-11DA-A96D-E3E1BE8701E7"
        maxRows="3" generic="2.0" xmlns="urn:uddi-org:api_v2">
      <categoryBag>
        <keyedReference
            tModelKey="UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384"
            keyName="Restaurants" keyValue="90101501" />
        <keyedReference
            tModelKey="UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384"
            keyName="Restaurants and catering" keyValue="90100000"/>
        <keyedReference keyName="keyword"
            keyValue="restaurant-lookup-ch" />
        <keyedReference keyName="precondition"
            keyValue="restaurant_name" />
        <keyedReference keyName="precondition"
            keyValue="restaurant_type" />
        <keyedReference keyName="postcondition"
            keyValue="restaurant_found" />
      </categoryBag>
    </find_service>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 5.9: Sample UDDI Request

are a subset of the `categoryBag` used in the search, which is performed by the `find_service` method. The construct `keyedReference` represents a namespace qualified name-value pair and is associated with a particular `tModel`.

As described in Section 2.2.4, UDDI uses the data structure `tModel` to organise services. Each `tModel` consists of a name, an explanatory description, and a Universal Unique Identifier (UUID). There are a number of predefined `tModel` structures, which represent classification schemes. For example, the `tModel`, with the key `UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384`, is the United Nations Standard Products and Services Code System (UNSPSC) [Uns98], a single global product and service classification system. The sample search query requires that the service instance is assigned both to the `Restaurants` and `Restaurants and catering` service categories , which correspond to `90101501` and `90100000` codes respectively, according to UNSPSC.

The `keyedReference`, which requires the service to find a restaurant, de-

```
<?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <serviceList generic="2.0"
    operator="jUDDI.org" xmlns="urn:uddi-org:api_v2">
   <serviceInfos>
    <serviceInfo
        businessKey="ADD69E70-8E5C-11DA-A96D-E3E1BE8701E7"
        serviceKey="86E8AFF0-8E5D-11DA-A96D-D429756505D9">
        <name>Sample Swiss Restaurant Service 1</name>
    </serviceInfo>
    <serviceInfo
        businessKey="ADD69E70-8E5C-11DA-A96D-E3E1BE8701E7"
        serviceKey="834DAEE0-8E5D-11DA-A96D-E9E5BC9E3DF9">
        <name>Sample Swiss Restaurant Service 2</name>
    </serviceInfo>
    <serviceInfo
        businessKey="ADD69E70-8E5C-11DA-A96D-E3E1BE8701E7"
        serviceKey="8000A6C0-8E5D-11DA-A96D-F5A9FAE03222">
        <name>Sample Swiss Restaurant Service 3 </name>
    </serviceInfo>
    ...
    </serviceInfos>
</serviceList> </Body> </Envelope>
```

Figure 5.10: UDDI response listing services for the query in Figure 5.9

fined by `keyName="postcondition" keyValue="restaurant_found"`, is not associated with a predefined `tModel`. It rather belongs to a generic `tModel`, which is used to define customised classifications.

The UDDI query can specify the parameter `maxRows`, which determines the maximum number of matching service instances to be returned. At present, the framework fetches three instances, one for immediate invocation and the other two as backups, as a proactive measure to mitigate service execution failure. Figure 5.10 shows an XML-based list of matching services, which is a result of the query shown in Figure 5.9.

Service deployment. When a service instance is selected from the UDDI response, such as one in Figure 5.10, the `Service Registry` returns its WSDL file. Figure 5.11 shows the sample WSDL file for a restaurant service. Given this binding information the system generates a `deployable service representation`, which stores a pointer to the WSDL file and the corresponding `monitoring procedure` which is described in detail in the next section. Pointers to additional replacement services, such as the one listed in Figure 5.10, may be included to serve as replicas in case the original service fails.

5.3 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.* [HV96], which includes two types of monitoring procedures: service monitors, discussed in Section 5.3.1, observe service execution, and event monitors, described in Section 5.3.2, track changes in the environment.

5.3.1 Service execution and monitoring

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 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, as

```
<?xml version="1.0" ?> <definitions
targetNamespace="urn:prototype:restaurant"
    xmlns:rns="urn:prototype:restaurant"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:format="http://schemas.xmlsoap.org/wsdl/formatbinding/"
    xmlns:java="http://schemas.xmlsoap.org/wsdl/java/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

   <message name="getRestaurantRequest">
      <part name="type" type="xsd:string"/>
      <part name="location" type="xsd:string"/>
   </message>

   <message name="getRestaurantResponse">
      <part name="restaurantName" type="xsd:string"/>
   </message>

  <portType name="RestaurantLookupService_CH_PT">
    <operation name="getRestaurant">
      <input message="rns:getRestaurantRequest"/>
      <output message="rns:getRestaurantResponse"/>
    </operation>
  </portType>

  <binding name="JavaBinding" type="rns:RestaurantLookupService_CH_PT">
    <java:binding/>
    <format:typeMapping encoding="Java" style="Java">
      <format:typeMap typeName="xsd:string" formatType="java.lang.String" />
    </format:typeMapping>
    <operation name="getRestaurant">
      <java:operation methodName="getRestaurant" methodType="instance"/>
      <input/>
      <output/>
    </operation>
  </binding>

  <service name="RestaurantLookupService_CH">
    <documentation>Restaurant Provider Service</documentation>
    <port name="JavaPort" binding="rns:JavaBinding">
      <java:address
        className="composition.services.RestaurantLookupService_CH"/>
    </port>
  </service>
</definitions>
```

Figure 5.11: WSDL for sample restaurant directory service

shown in Figure 5.1. 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.

As an example, if the selected service cannot find a desired restaurant, the architecture specific service composition layer may try to find an alternative service that can fulfill the request. If that fails, the abstract service composition layer will try to recompose the request if possible. If that fails, the `GoalMorph` system in the composition request management layer, described in Chapter 4, may transform the `composition request`, therefore searching for a take away place or a bistro in the same area. Similarly, if the `BookRestaurant` service fails due to the unavailablity of seats, an update of the state occurs and another restaurant may be selected.

The next chapter discusses the performance of this interleaved composition and execution process and how it provides an effective and useful failure management method.

5.3.2 Goal state monitoring

Certain events may cause changes in the environment that affect the ability to completely satisfy a `composition request`. Also, new opportunities may arise that the framework can take advantage of. It is necessary to monitor goal conditions and track unexpected events in the environment, in addition to monitoring and verifying service execution.

Event monitors are structured in the same way as service monitors and are invoked when relevant goal conditions are introduced or removed. They run in parallel with the executing service, and update the state of the `Composition Engine` and `Execution Engine`. At present service descriptions define service monitors and a system designer sets up event monitors. Deciding which events to monitor that are relevant to service composition raises an open research challenge.

In the context of this work events are considered to be goal states and unexpected side-effects of the service execution. Unforseen events are handled in the following way. If the main service effect has been unexpectedly satisfied, for example the user manually lowers the music volume, the `Execution Engine` 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.

If a required precondition is no longer true, as a side-effect of some other action, the system detects the relevant precondition and contacts the `Composition`

`Engine` to replan in an attempt to achieve it. For instance, if a user's location changes and the existing driving directions are no longer valid, the translation service should no longer proceed with the translation of the original request.

Goal state monitoring is an important and a large challenge to be addressed in the composite service execution. Whilst this issue is not dealt with in this dissertation, the framework has been designed to integrate the existing, ready-made, solution proposed by Haigh et al. [HV96]. This enables detection of execution failures and compensation for them, as well as responding to changes in the environment.

5.4 Summary

This chapter has focused on practical issues related to the implementation and deployment of the prototype framework for context aware service composition. Implementation details of the abstract service composition, architecture specific service composition and execution and monitoring layers were described. The internal architecture of platform components, introduced in Chapter 3, was presented, showing how they deliver the required functionality.

Firstly, the implementation of the abstract service composition layer was presented. Facilities provided by the `Abstract Service Repository`, which allow for independence of component technology and support semantic service annotation, were described. The applicability of existing planning systems to Web service composition problems was discussed and the use of the forward chaining planner TLPlan has been demonstrated.

Secondly, the architecture specific service composition layer was described, presenting how the `Plan Instantiator` is used to construct executable plans and to achieve independence of the run-time environment. The `Service Registry` has been developed as a UDDI based component to collect service advertisements and perform service lookups. Given an `abstract execution plan` specified in BPEL4WS, the `Plan Instantiator` communicates with the `Service Registry` to instantiate services and generate deployable service sequences.

Thirdly, the implementation of the execution and monitoring layer was discussed. The `Monitoring Engine` provides facilities to examine whether the preconditions and postconditions of an executable service are met and also to track changes in the environment.

The use of internal representation structures facilitates system extensibility, as identified in Section 1.3. The fault tolerance challenges set in Section 3.4 have been met by the use of monitoring procedures to provide adequate resilience to execution failures. The next chapter assesses, through experiments and discussion, the efficiency, scalability, and effectiveness of the design decisions made and the mechanisms employed.

Chapter 6

Evaluation

This chapter presents the evaluation of the prototype implementation of the framework for context aware service composition. Quantitative methods are employed to measure the framework's *performance* and *scalability*. Qualitative approaches are followed to determine its effectiveness in terms of reducing the *development effort* required for building context aware applications.

Firstly, the latency of the framework was considered. The time taken by the constituent phases of a composition process was measured for test cases with and without composition and execution failures. Secondly, the scalability of the framework was evaluated from three perspectives: (1) increasing size of the application domain, in terms of the number of available service instances in the `Service Registry`, (2) increasing size of a `composition request`, which is the number of individual goal conditions to be satisfied, and (3) increasing number of concurrent `composition requests`.

The development effort and efficiency of the proposed approach in building context aware applications is analysed and compared to the traditional application design approach. Finally, this chapter discusses how the requirements for a new application model, which have been identified in Section 1.3, are met through a combination of design decisions, mechanisms provided, and implementation choices followed.

6.1 Experimental setup

Framework deployment. Figure 6.1 shows the configuration of the environment in which the experiments were conducted. Two machines were used hosting different layers of the framework. The machine providing access to the abstract service composition layer, which contained TLPlan, was a dual processor Pentium

III 800 MHz with 2 GB RAM, located in Cambridge, UK. An IBM Thinkpad T41 with an Intel Pentium M 1700MHz processor and 1 GB RAM contained the composition request management, architecture specific service composition, and execution and monitoring layers. This machine was located in Berlin, Germany. The two machines were connected over the Internet via a link capable of supplying 380 kbps. The average throughput was measured using Test TCP (TTCP) [MS85].

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.

Figure 6.1 also shows the particular infrastructure implementing the presented framework. Abstract service composition was controlled by TLPlan. An application server Tomcat v5.5 contained the `Service Registry`, which was implemented using jUDDI v0.94 [Jud03], and the BPWS4J engine [IBM04], a platform for creating and executing BPEL4WS processes.

Sample domain. To conduct the experimental evaluation a sample context aware infotainment application domain was developed for describing the usage scenario, which was presented in Section 3.1. The `Goal Service` contained twenty different `composition requests`. The `Context Service` provided seven different context types, whose values triggered creation of context goal conditions. There were twenty sample abstract services in the `Abstract Service Repository`, such as `RestaurantFinder` and `DirectionsFinder`. The `domain description` consisted of 100 facts and twenty abstract services.

The `Service Registry` contained a number of instances of each abstract service available in the `Abstract Service Repository`. Service instances were assigned to several categories, which classified service capabilities and described their geographical coverage, in the `Service Registry`.

Table 6.1 shows the number of categories each service instance was assigned to. The `tModel` structure in UDDI stores the category information. The types of categories employed were two; geographical category, describing the geographical area in which service is applicable, and UNSPSC code categorisation. In addition, `tModels` were constructed to describe service preconditions and postconditions.

Figure 6.2 shows a sample UDDI description for a "Sample Swiss Restaurant Service 3", whose WSDL file is shown in Figure 5.11. This service is categorized as `Restaurant` and `Restaurant and Catering` service using UNSPCS codes. Additional `tModel` structures describe preconditions and postconditions of this service. For example, `restaurant_type` and `restaurant_found` are a precondition and a postcondition respectively for the sample service in Figure 6.2.

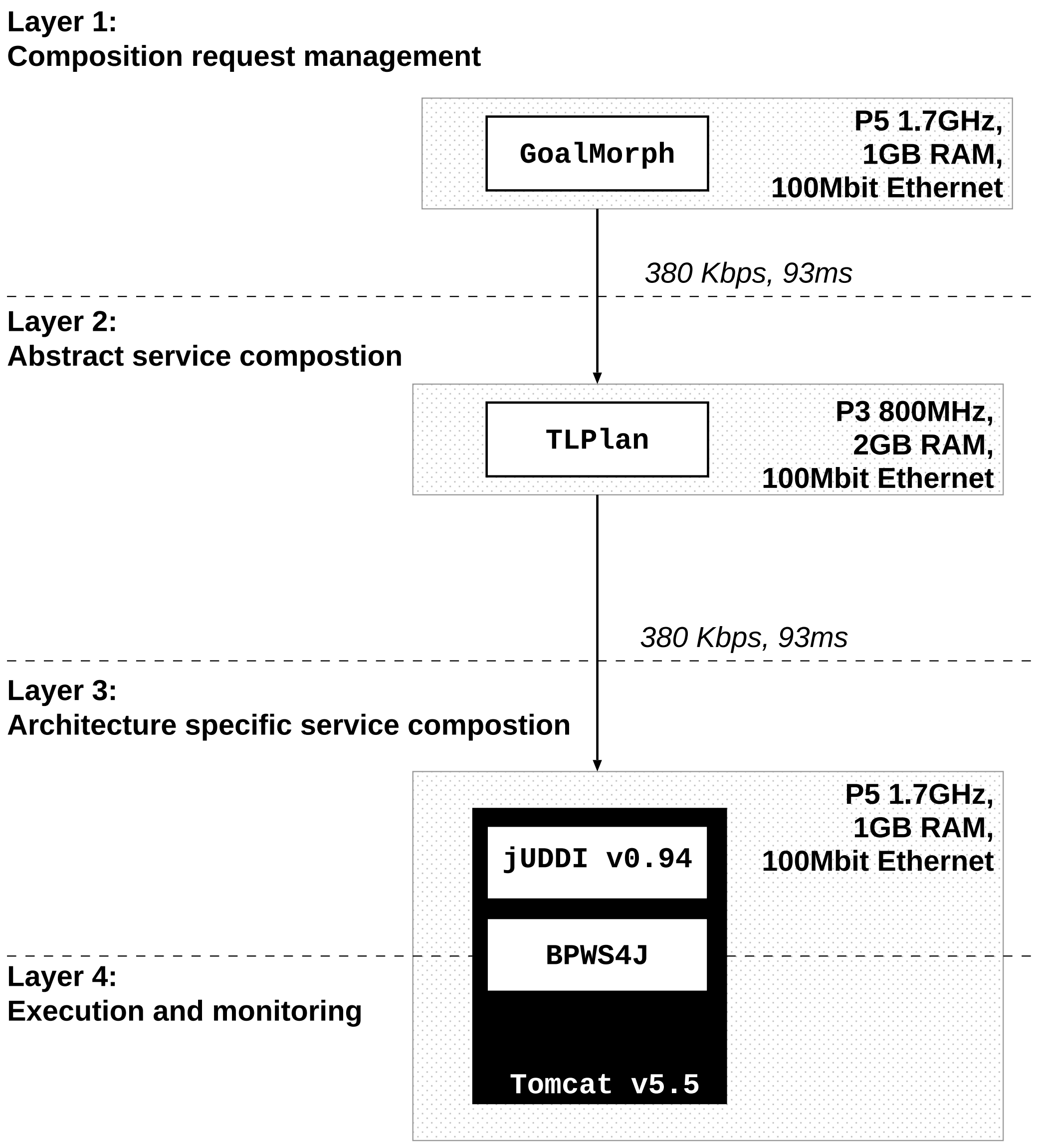

Layer 1:
Composition request management
GoalMorph
P5 1.7GHz, 1GB RAM, 100Mbit Ethernet
380 Kbps, 93ms
Layer 2:
Abstract service compostion
TLPlan
P3 800MHz, 2GB RAM, 100Mbit Ethernet
380 Kbps, 93ms
Layer 3:
Architecture specific service compostion
P5 1.7GHz, 1GB RAM, 100Mbit Ethernet
jUDDI v0.94
BPWS4J
Tomcat v5.5
Layer 4:
Execution and monitoring

Figure 6.1: Experimental setup

6.2 Performance

This section evaluates the overall process of service composition, which is divided into steps corresponding to the operation performed by each layer in the software architecture, as described in Section 3.3. The composition performance is analysed for the following four different variants of our usage scenario.

```
<?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <serviceDetail generic="2.0" operator="jUDDI.org"
    xmlns="urn:uddi-org:api_v2">
   <businessService businessKey="ADD69E70-8E5C-11DA-A96D-E3E1BE8701E7"
        serviceKey="8000A6C0-8E5D-11DA-A96D-F5A9FAE03222">
    <name>Sample Swiss Restaurant Service 3</name>
    <bindingTemplates>
     <bindingTemplate bindingKey="80053AA0-8E5D-11DA-A96D-A07AE3BE618C"
        serviceKey="8000A6C0-8E5D-11DA-A96D-F5A9FAE03222">
      <accessPoint URLType="AccessPoint.HTTP">
        localhost:8080/RestaurantLookup.wsdl</accessPoint>
      <tModelInstanceDetails/>
     </bindingTemplate>
    </bindingTemplates>
    <categoryBag>
     <keyedReference keyName="precondition"
        keyValue="restaurant_name"
        tModelKey="UUID:A035A07C-F362-44dd-8F95-E2B134BF43B4"/>
     <keyedReference keyName="precondition"
        keyValue="restaurant_type"
        tModelKey="UUID:A035A07C-F362-44dd-8F95-E2B134BF43B4"/>
     <keyedReference keyName="postcondition"
        keyValue="restaurant_found"
        tModelKey="UUID:A035A07C-F362-44dd-8F95-E2B134BF43B4"/>
     <keyedReference
        keyName="keyword" keyValue="restaurant-lookup-ch"
        tModelKey="UUID:A035A07C-F362-44dd-8F95-E2B134BF43B4"/>
     <keyedReference
        keyName="Restaurants" keyValue="90101501"
        tModelKey="UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384"/>
     <keyedReference
        keyName="Restaurants and catering" keyValue="90100000"
        tModelKey="UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384"/>
    </categoryBag>
   </businessService>
  </serviceDetail>
 </soapenv:Body>
</soapenv:Envelope>
```

Figure 6.2: Sample UDDI description for service instance shown in Figure 5.10

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 6.1: Sample service categorisation

Test cases

Case 1. The composition time was measured for the case in which the process of service composition occurred under ideal conditions, without any composition or execution failures.

Case 2. In this use case a composition failure was simulated by randomly removing service descriptions from the `domain description`, before a `composition request` was submitted. The failure injection process reduced the domain description size by 20% on average. As a result, this triggers the operation of the `GoalMorph` composition failure management system. The total composition time was measured, including the time for service failure recovery.

The following two cases illustrate scenarios in which the service fails during execution thereby triggering a service replacement process. Two different ways of handling such execution failures were compared.

Case 3a. Firstly, the traditional approach to service failure was applied. 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 is discovered, bound and scheduled for invocation.

Case 3b. A second variant of this case employed a proactive approach to service failure recovery. Several service replicas were deployed at the same time as the initial service instance. This reduced the overhead of the discovery process once the execution failure occurred.

Measurements

For each test case the times required by the following steps were measured:

1. *Layer 1: composition request management*
   (a) Retrieval of core goal conditions from the `Goal Service`.
   (b) Context retrieval from the `Context Service`.
   (c) Construction of context goal conditions by the `Context Proxy`.
   (d) Assembling of the `composition request`.
2. *Layer 2: abstract service composition*
   (a) Translation of the `composition request` to a `problem definition`.
   (b) Translation of abstract service descriptions in the `Abstract Service Repository` to generate a `domain description`.
   (c) Composition of an `abstract plan` by the planner using the `problem definition` and the `domain description`.
3. *Layer 3: architecture specific service composition*
   (a) Creation of an `abstract execution plan` in the BPEL4WS format.
   (b) Deployment of a BPEL4WS file on the BPWS4J engine.
   (c) Atomic service discovery and instantiation.
4. *Layer 4: execution and monitoring*
   (a) Service invocation, contacting a service at a specified binding.
   (b) Running the `Monitoring Engine`, which observes the environment and service execution to discover failures.
5. *Layer 1: composition failure recovery*
   (a) Transformation of the failed `composition request`, using the `GoalMorph` system, into ones that may be possible to be solved.
6. *Layers 3 and 4: service execution failure recovery*
   (a) Running the recovery mechanism after service execution failure to replace the failed service with another operating instance.

Step 2b in the list was performed only once at the beginning of the overall evaluation, as each test case uses the same `domain description`. Therefore the measurements do not include the time for this step. On average, loading the domain and translating it to the format supported by the `Composition Engine` takes approximately 4.3 seconds. Furthermore, steps 5 and 6 are only measured when a composition (Case 2) or a service execution failure occurs (Cases 3a and 3b). In addition, the experiments in Section 4.3.1 show the quality of the modified plans in comparison to the original service composition request.

The time needed for service discovery and instantiation includes the total time for all atomic service forming the composite service. Future work will consider interleaved service discovery and execution, in which each discovered service is immediately invoked, and the subsequent service is instantiated at the same time.

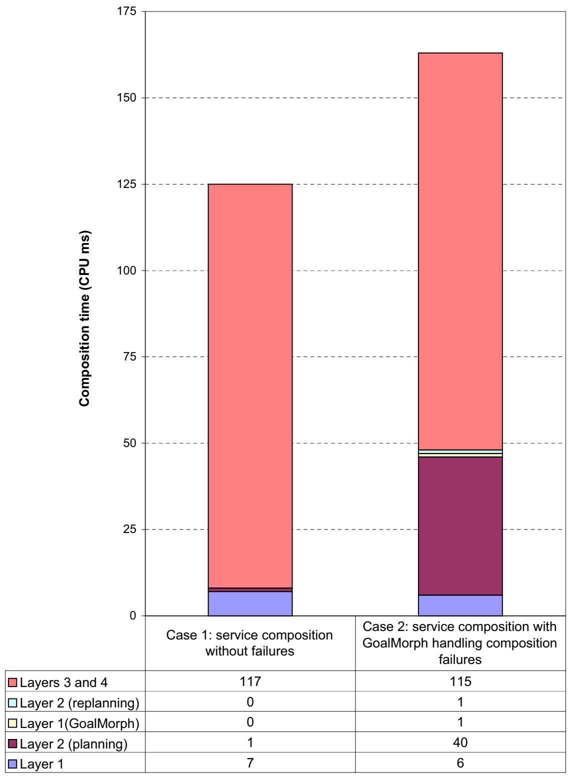
Measurement procedure The measurements were performed by obtaining snapshots of the total CPU time consumed by the system after each of the steps in the composition process as previously described. To obtain these figures I used JConfig [JCo02], a cross-platform library, which employs native OS-dependent system calls, and transfers data to Java through Java Native Interface (JNI). All the measurements were rounded to the nearest milisecond.

Composition timeline

Figures 6.3 and 6.4 show the composition timeline for each test case. All measurements were performed for a `composition request` containing ten goal conditions, out of which five are context goals. The resulting composite service consisted of 23 atomic services. The discovery process was performed using the `Service Registry`, which consisted of 160 service instances, each categorized in three to six classes. The measurements were repeated twenty times.

Figure 6.3 compares the average composition time for test cases with and without composition failure. When composition failure occurs in Case 2, it triggers the `GoalMorph` composition failure management system. This results in an overhead of 42 ms, out of which 40 ms is the time required for the planning process to fail, 1 ms to transform the goal and 1 ms to replan the request. It is important to note that the planning time can be bound, by setting the planner to timeout if the plan is not found within a certain time period.

Figure 6.4 shows the comparison of the composition timeline for the other two cases, in which an execution failure occurs. By contrast to Case 3a, in which each service is replaced as it fails, in Case 3b a service replica are fetched and cached, during the initial service discovery process. The proactive approach in



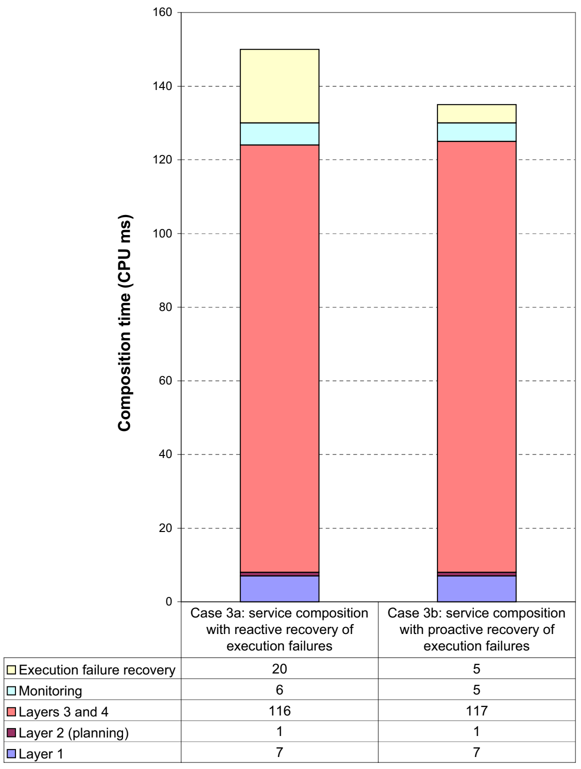
	Case 1: service composition without failures	Case 2: service composition with GoalMorph handling composition failures
🟥 Layers 3 and 4	117	115
⬜ Layer 2 (replanning)	0	1
⬜ Layer 1(GoalMorph)	0	1
🟪 Layer 2 (planning)	1	40
🟦 Layer 1	7	6

Figure 6.3: Composition timeline for Case 1 and Case 2

Case 3b reduces the failure recovery time from 25 ms to 10 ms.

On average the normal composition without any failures takes approximately 125 ms for a composite service consisting of 23 atomic services. The overhead introduced by handling composition failures (2 ms + planning time) and execution failures (10 ms) is acceptably small.

It is important to note that the reason why the architecture specific composition layer takes a disproportionate amount of time, compared to other steps, is because of the low performance of the WSDL parser [Wsd05] used for generating the BPEL4WS file.



	Case 3a: service composition with reactive recovery of execution failures	Case 3b: service composition with proactive recovery of execution failures
Execution failure recovery	20	5
Monitoring	6	5
Layers 3 and 4	116	117
Layer 2 (planning)	1	1
Layer 1	7	7

Figure 6.4: Composition timeline for Case 3a and Case 3b

6.3 Scalability

This section discusses scalability issues in the proposed framework design, which are raised by different types of participating components and operations carried out between them. The experimental evaluation considers the framework's ability to operate under increasing: (1) domain size, (2) size of `composition requests` and (3) number of concurrent `composition requests`. The experimental configuration used for performance described in Section 6.1 was also used to run the scalability tests. The complexity of the planning domain size was extended to contain 150 facts and 100 service types, to accommodate composite services consisting of up to 100 atomic services.

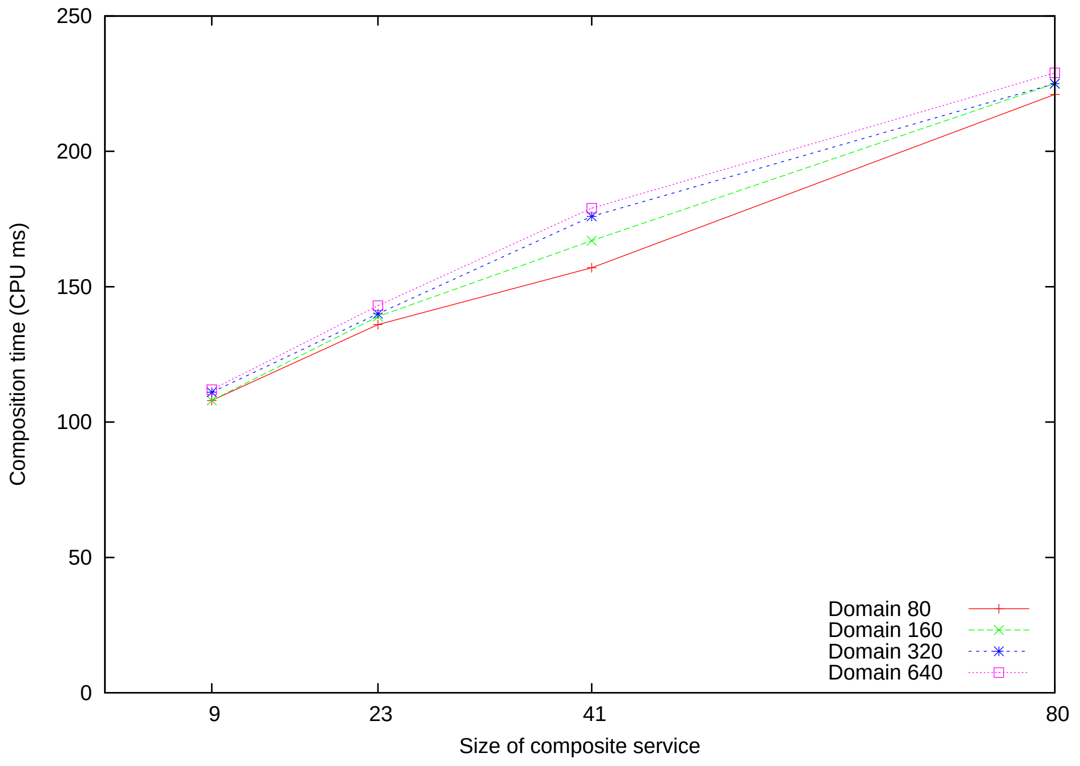


Figure 6.5: Scalability when increasing domain size

6.3.1 Scalability when increasing domain size

This experiment varied the number of service instances available in the `Service Registry` from 80 to 640. The parameters of the planning domain remained as described in Section 6.1, consisting of 100 facts and twenty abstract service types. For each stage of the domain 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. It is important to mention that the composite service may include more than one instance of an abstract service. Often in the composite service atomic services are executed more than once.

The composition process was invoked twenty times to measure the average CPU time. Figure 6.5 shows the results of this experiment, demonstrating that the framework can can scale to a realistic domain size (640 instances), while still requiring less than 229 ms of CPU time to compose complex services.

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

Table 6.2: Composition request size and composite service size

6.3.2 Scalability when increasing composition request size

This experiment measured the impact of `composition request` size on the composition time. The number of goals in the `composition request` was varied from 5 to 40, and the number of context goal conditions was varied from 1 to 10, respectively. Table 6.2 shows `composition request` sizes, the number of context goals in each and the average size of a resulting composite service used for testing.

This experiment measures the trend in composition time as the number of context goal conditions in the `composition request` was increased. Figure 6.6 shows the average composition time and provides a breakdown of performance for each layer of the framework.

While the overall composition time increases with the complexity of the `composition request` and consequently the resulting size of the composite service, the framework can compose and invoke a composite service of size 80, which corresponds to the `composition request` of size 40, in less than 0.224 seconds of CPU time.

6.3.3 Scalability when increasing number of composition requests

This experiment was conducted to measure how the system responds to an increasing number of concurrent `composition requests`. The number of concurrent composition sessions was varied from 1 to 100.

The BPWS4J engine v2.1, which is used in the implementation of the prototype, 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.

Figure 6.7 shows that there is a very modest increase in the composition time as the number of concurrent composition sessions grows. Specifically, the total composition time for one hundred simultaneous `composition requests` is less than twice the composition time for a single `composition request`. This demonstrates the ability of the system to scale gracefully to accommodate large numbers of concurrent `composition requests`.

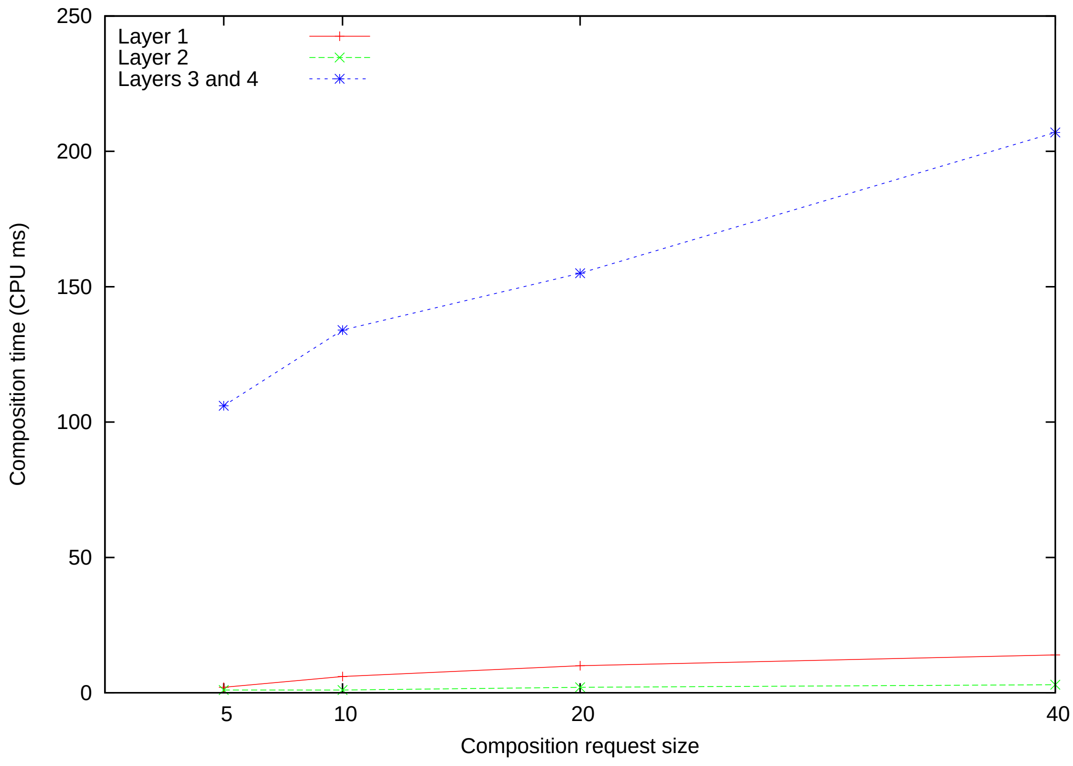


Figure 6.6: Scalability when increasing the composition request size, for domain size 160.

Table 6.3 shows the distribution of composition time across different layers and the corresponding composition steps. It is important to mention that 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. The bottleneck of the composition process is the generation of the BPEL4WS file, due to the low performance of the WSDL parser used [Wsd05]. Furthermore, the relatively small increase in the time taken for layers 3 and 4, shown in Table 6.3, results from the caching mechanism employed by the `Service Registry`, as the test data set contained composition requests consisting of overlapping service instance queries.

6.4 Qualitative evaluation

In addition to the previously described quantitative evaluation methods, two further qualitative directions of evaluation are undertaken. The first one analyses the effort involved in developing context aware applications using the traditional

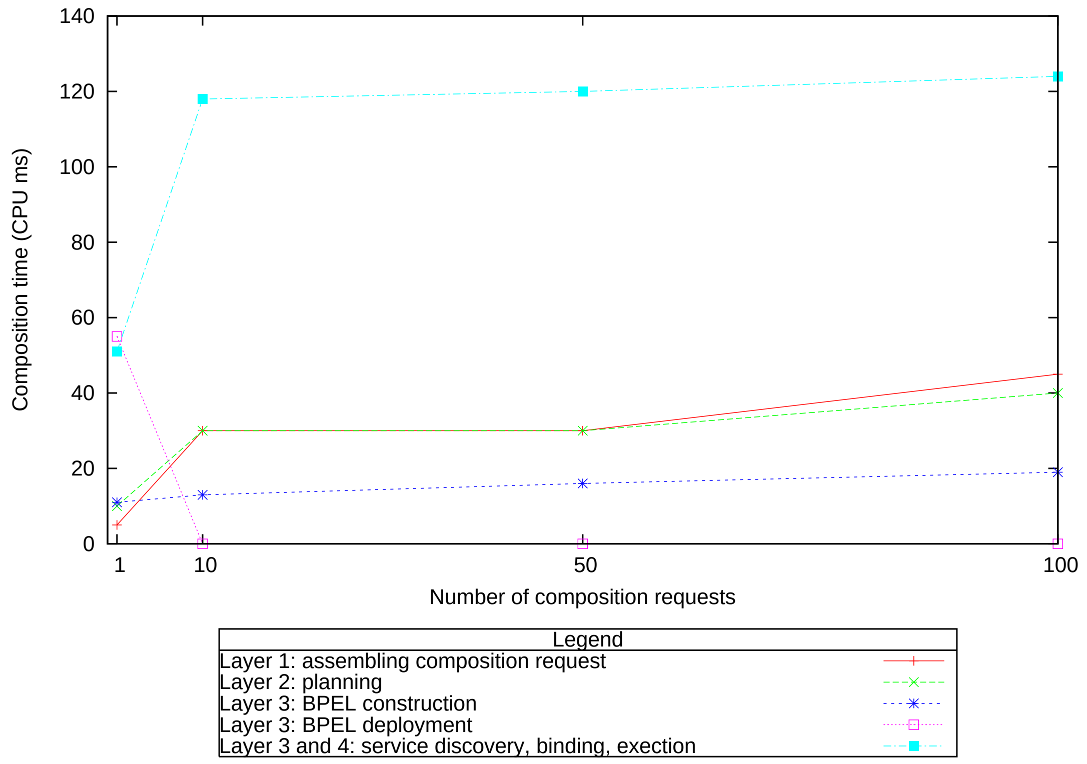


Figure 6.7: Scalability when increasing number of composition requests

application development methodology. It then compares it to the effort it takes to develop context aware applications using the proposed methodology, grounded in the context aware service composition framework. The second path of evaluation considers how the aims outlined in Section 1.3 have been met by the framework design.

6.4.1 Development effort

This dissertation presents a framework for context aware application development, grounded in the idea of planning-based service composition. The drive behind the design of the framework is to make context aware applications easier to build, in order to reduce their complexity and increase their extensibility.

At present there is no set of standard metrics for measuring system complexity. Furthermore, there is no formal specification of qualitative parameters to evaluate the design of the system.

Determining a set of metrics for evaluating autonomic, ubiquitous systems is a continuous research topic. Ranganathan *et al.* [RC03] propose measurements

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

based on task-structure complexity, unpredictability, size complexity, chaotic complexity and algorithmic complexity. McCann *et al.*[MH04] identify a set of metrics for autonomic systems based on AI, primarily dealing with the performance of the system, rather then its design. This set contains parameters such as Quality of Service, granularity and flexibility, failure avoidance, degree of autonomy and time to adapt.

The complexity of the 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 describes the steps involved in developing context aware applications using legacy application frameworks, which embed the contextual dependencies. It compares them to the ones in the approach proposed in this dissertation using the sample application scenario, described in Section 3.1. These two approaches are compared in terms of the design and development effort.

Developing applications using a legacy design model. Table 6.4 summarises the steps involved in developing applications based on traditional design models, using available application design toolkits and context middleware solu-

Step	**Step description**	**Development mode**	**Frequency**	**Difficulty**
Task specification				
1	Specify context	semi	repeated	easy
2	Register with context providers	auto	repeated	easy
3	Specify desired core goal	manual	repeated	moderate
4	Specify desired context behaviour	semi	repeated	*difficult*
5	Select desired task	semi	repeated	moderate
Application behaviour specification and configuration				
6	Develop core functionality	manual	one-off	moderate
7	Encode context aware behaviour	manual	repeated	*difficult*
Application execution				
8	(Platform-specific) deployment	semi	repeated	moderate
Unpredictability and failure recovery				
9	React to context changes	semi	repeated	*difficult*
10	React to task changes	auto	repeated	moderate
11	React to execution failures	semi	repeated	moderate

Table 6.4: Design process for building context aware applications using a legacy approach

tion, where appropriate.

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 follows *manual coding*, *semi-automated* with the assistance of toolkits and scripts, or *fully automated* with the assistance of frameworks or middleware.

The difficulty of the different phases depends on the availability of suitable frameworks, toolkits and scripts. For example context specification and registration with context providers are increasingly common feature of context middleware solutions. The key challenges in using this design method are how to specify context behaviour and how the developer can encode it. This legacy approach is scenario specific. There is very little reusability of essential phases in development of context aware applications in a different application domain.

The most critical steps are 4, 7, and 9 to 11, which deal with the specification of context behaviour and the systems's ability to handle unpredictability and react to failures. These tasks are most difficult as they require a high amount of effort to specify and encode context behaviour. For example, to incorporate a new context command, based on a new context type or value, a developer needs to perform several actions. Firstly, the developer has to specify desired context types to be retrieved from context middleware. Then the developer has to embed the

Step	**Step description**	**Development mode**	**Frequency**	**Difficulty**
Task specification				
1	Specify context	semi	repeated	easy
2	Register with context providers	auto	repeated	easy
3	Specify desired core task	semi	repeated	easy
4	Specify desired context behaviour	auto	repeated	easy
5	Select desired task	semi	repeated	easy
Application behaviour specification and configuration				
6	Semantic description of services	semi	one-off	difficult
7	Generate domain description	semi	one-off	moderate
8	Generate problem definition	auto	repeated	easy
9	Generate abstract plan	auto	repeated	easy
10	React to plan failures	auto	repeated	easy
11	Generate architecture specific plan	auto	repeated	easy
Application execution				
12	Service discovery and invocation	auto	repeated	easy
Unpredictability and failure recovery				
13	React to context changes	auto	repeated	easy
14	React to task changes	auto	repeated	easy
15	React to execution failures	auto	repeated	easy

Table 6.5: Design process for building context aware applications using the proposed approach, based on the concept of context aware service composition

new context functionality, which is a process that often requires reprogramming the existing application. Finally, the main limitation of this approach is that the set of features provided by the application is predefined.

Developing applications using context aware service composition. Table 6.5 summarises the design steps involved in building context aware applications using the approach proposed in this dissertation. Most of the phases are semi automated. This approach introduces two complex phases: semantic annotation of services (Step 6) and construction of the domain description (Step 7). Despite the fact that they require extensive (manual) design and development, both stages only occur once for each application domain.

Some of the phases shown in Table 6.5 can be automated and their development can be further simplified. For example, goals in the `Goal Service` can be automatically obtained by analysing the domain specification, and inferring the possible `composition requests` that can be fulfilled.

Finally, the most important advantage of this approach is its generality. It is independent of specific application scenario and embeds a general method for

Feature	**Legacy approach**	**Context aware service composition**
New context type	Automatically discovered using middleware	Automatically discovered using middleware
New context value	Context goal manually encoded	Context goal automatically generated
New context behaviour	Requires reprogramming the application behaviour	Automatically generated, may trigger recomposition

Table 6.6: Comparison of legacy and proposed application models in dealing with unanticipated context

specifying context behaviour.

Comparison: supporting new context behaviours Tables 6.4 and 6.5 show the steps involved in developing the sample application using the legacy approach and the new application model, respectively.

The goal of a context aware application designer is to develop applications that adapt their behaviour in response to context. The three most difficult and time consuming steps: execution, deployment and failure recovery are taken care of by the framework. The designer is expected to concentrate on the specification and performance of the context aware behaviours. Table 6.6 summarises how both of these approaches handle unanticipated context and evolving requirements.

The context aware service composition approach still requires certain amount of prediction and encoding of the set of possible constraints, and the impact they would have on the viability of a given plan (i.e., the domain description). There are three different ways how the framework new and unanticipated constraints could be dealt with by the framework, all within the scope of ContextMesh and GoalMorph system.

1. Context middleware provides context information in the form of a key-attribute pair, therefore cycling is provided it may be classified as context type activity. Unless the explicit representation of cycling is already present in GoalMorph (or provided by domain engineer- see third item), user may provide any user and application specific categorization and association of cycling (such as relation of cycling to other types of activities).

2. System handles new combinations of context. For example, a new device available in the market may be a combination of existing device capability values. Therefore there is no need to encode all the combinations in the sys-

tem, as the different context rules will be provided through ContextProxy and planner will reason over those.

3. Finally, when a completely new concept (e.g. cycling) is introduced, the assumption is that the domain engineer will provide some form of a tag associated with cycling, which will describe it in terms of its type and any context hierarchy membership and relation to other existing values, as well as initial utility value. This tag may also point to the actual ontology that contains the description of the concept.

In conclusion, despite the initial overhead in devising the new framework, it provides many advantages. It reduces the need for many steps in the design process that are replicated across individual applications and provides uniform support for these steps.

6.4.2 Effectiveness

The quantitative evaluation undertaken shows that the implementation of the framework provides an efficient and scalable service composition infrastructure. This section discusses in detail how each of the framework requirements and dissertation aims identified in Section 1.3 is met through system design principles that hold independently of the specific infrastructure implementing the framework.

1. *Composition failure recovery.* This dissertation introduces `GoalMorph`, a system for composition failure management. `GoalMorph` converts failed `composition requests` into ones that can be solved, by utilizing core and context goal transformation. The central component of `GoalMorph` is the `Context Mesh`, a multidimensional data structure for hierarchical organisation of context. It enables *context layering*, operation which controls the amount of context data considered when transforming the context goal.

2. *Automated context aware goal construction.* The framework presents the `Context Proxy`, a component that generates `cogotags`, `context goal tags`, for representing context goal conditions. The `Context Proxy` fetches the current context data from the `Context Service` and assembles context goal conditions. Context behaviour rules may be injected into the `Context Proxy` by the user, the `Context Service` or inferred from the past interaction of user with the system.

3. *Execution failure recovery.* The framework takes active steps to maintain the quality of composed services during their execution. It keeps a small number of service replicas for each service type. As a result, it can easily and quickly recover failures caused by service unavailability, by immediately replacing a failed service with a running one. The advantage of this approach is that it reduces the failure recovery overhead by avoiding invoking the discovery process to find a new service at execution time.

4. *Scalability.* The quantitative evaluation has shown that the framework efficiently copes with the increasing domain size and `composition request` size. The proposed framework can handle more than a hundred of concurrent `composition requests` without presenting performance bottlenecks. The system scales linearly as the complexity and the volume of requests grow, allowing an on demand addition of resources to cope with the increased demand.

5. *Independence of application domain.* The proposed framework supports any kind of application scenario without requiring compliance to any domain-specific code, aside from the `domain description`, which is extracted from the `Abstract Service Repository` and imported into the `Composition Engine`.

6. *Independence of component technology.* The framework does not assume or require a specific component technology, such as DCOM, CORBA or Web service. It accommodates plug-in modules that convert the specific component description format into an internal representation format. This enables the framework to accommodate any future emerging standards.

7. *Independence of composition methodology.* The framework is open to alternative composition methodologies. As described in Chapter 3, the framework employs an internal representation of the composite service. The `Translation Module` converts the `composition request` into the `problem definition` format supported by the composition method in use. The resulting `abstract plan`, representing composite service, is similarly stored in the internal representation format, to be converted to the implementation specific language. The benefit of such a design is twofold. Firstly, it results in an open and extensible framework, in which multiple different composition algorithms, such as model based and probabilistic planning, can coexist, providing specialised composition methodology tailored to the level of determinism and observability of the application domain. Secondly, it makes the `Composition Engine` independent of the execution environment.

8. *Independence of context middleware.* The framework employs internal representation for context types and their corresponding values. This has a number of important benefits. It enhances the extensibility of the framework and enables the acquisition of context from multiple context middleware solutions. Additionally, it supports switching between different context middleware solutions when the `Context Service` becomes unavailable.

6.5 Summary

The *efficiency* and *scalability* of the developed framework were demonstrated through experimental evaluation. The prototype system allows for composition and deployment of complex services, consisting of 80 components, in less than 0.225 seconds , when the `Service Registry` contains 160 service instances. It can scale to 100 concurrent composition requests without presenting performance bottlenecks.

The *development effort* required for building context aware applications using the proposed approach was also analysed. The proposed method was contrasted to the traditional application design approach. Whilst the proposed approach does require manual effort, e.g. Context Mesh encoding, the application programmers' effort saved by enabling applications to be dynamically composed and to evolve is greater. The framework automises 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. Finally, the chapter presented how each of the requirements for a new application model and dissertation aims, identified in Section 1.3, is met through the system design principles employed.

Chapter 7

Conclusion

The proposed context aware service composition framework represents a novel model for developing context aware applications in a structured and extensible way. Services are assembled based on the user's context, such as available resources, time constraints, and location. The core feature of this approach is the recomposition of the composite service during its execution, which may be triggered by changes in the context.

This is a challenging endeavor because of the failures that may arise during the composition process, the lack of automated ways for context aware goal construction, and the shortcomings of existing AI planning systems to fully handle the complexity of the service composition problem.

The framework provides a general-purpose, failure-tolerant solution for context aware service composition by combining a number of novel ideas. Firstly, it employs `GoalMorph`, a composition failure management system, which applies context aware goal transformations to failed `composition requests` to convert them into the ones that can be solved. At the same time, the layered framework design and the use of internal representations are instrumental in achieving independence of the framework from a specific component technology, semantic language, composition methodology and run-time environment. Finally, a new format for representing context aware goals, called `cogotags`, has been devised to enable automated construction of context aware composition requests.

A prototype implementation of the proposed framework has been developed and deployed. Evaluation results demonstrated that the framework provides a practical, efficient and scalable solution for realistic applications. The prototype system allows for composition and deployment of complex services, consisting of 80 components, in less than 0.22 seconds, when the `Service Registry` contains 160 service instances. It can scale to more than a hundred concurrent composition requests without causing performance bottlenecks.

`GoalMorph`, the framework extension to handle composition failures has been shown not to significantly impair the performance of the framework, adding only 0.001 seconds for a single goal transformation. `GoalMorph` transforms failed `composition requests`, consisting of more than 40 goal conditions in the example environment, into requests that can be solved and achieves more than 60% of the original `composition request` utility on average.

7.1 Contributions

This dissertation makes four principal contributions:

1. It reviews existing work in context aware computing and service composition and identifies the main research challenges that need to be tackled by a new application model to facilitate context awareness.
2. It evaluates the state of the art in existing AI planning technology and analyses its applicability to the Web service composition problem.
3. It describes the design and implementation of a novel proposed service composition framework for building context aware applications. It demonstrates that the mechanisms and techniques employed work efficiently and scalably.
4. It presents `GoalMorph`, a composition failure management system, which applies context aware goal transformations to failed `composition requests` to convert them into ones that can be solved by the `Composition Engine`. `GoalMorph` introduces the following:
   (a) A model for the representation of context aware goals.
   (b) A taxonomy of core and context goals and the corresponding transformations used to transform failed `composition requests`.
   (c) A utility based mechanism for trading off goal transformations and corresponding partial success in achieving one goal against partial success in achieving another.

7.2 Future work

To further address the goal transformation problem and allow for more flexible service composition, several related challenges need to be overcome. This section describes some of the potential directions for future work.

7.2.1 User driven goal transformation selection

`GoalMorph` allows for multiple methods to guide the selection of applicable goal transformations. At present the utility driven transformation mechanism uses planner success to update the originally randomly assigned transformation utilities. The next step is to extend `GoalMorph` to update utilities based on user feedback. For that, a model representing the user's satisfaction with a transformed request is needed.

Quantification of user preferences is a popular research topic. Poladian *et al.* [PSGS04] devise a mathematical model for optimising the dynamic configuration of resource-aware services. They use this approach to maximise configuration utility based on three input parameters: user preferences, application capability profiles, and resource availability. Poladian *et al.* express a user's utility by means of a user preference function that maps from a multidimensional configuration space to a one-dimensional utility space.

One idea would be to adapt this model to guide the utility-based transformation selection process in `GoalMorph`. Poladian *et al.* consider utility as a measure of a user happiness with respect to the possible outcomes. In `GoalMorph`, this would be a formal representation of how useful a transformed `composition request` is relative to the original request. Using a similar approach to Poladian *et al.* the goal transformation utility can be encoded as a number between zero and one, where zero utility corresponds to the transformed goal being unacceptable for the request, and one corresponds to complete user satisfaction. As a result, when the utility is one, increasing it may not improve the user's perception of usefulness for the specific task. Furthermore, based on the acquired utilities, learning methods can be applied to guide future goal transformations.

7.2.2 Composition request scheduling

A production deployment of the framework, with the `GoalMorph` extension will result in a number of `composition requests` competing for system resources. Given a number of independent requests and their possible transformations, a mechanism for their prioritisation is essential.

A method for scheduling of `composition requests` or transformations depending on high-level criteria, such as the importance of goals and the subscription class of the users submitting the requests, is envisaged. For instance, users paying the lowest subscription fee may get a lower probability of their goal utility requirements being met, whereas a high subscription user with the same requirements may get a higher probability and a faster service response.

7.2.3 Quality of Service for composite service execution

Current research approaches to providing Quality of Service-based service composition assume that service providers include Quality of Service specifications in the service description and that this will be available through service registries [ZBN$^+$04]. However, Fan *et al.* [FK05] report that only a very small number of real services are actually semantically annotated, at the time of writing. Furthermore there is a significant gap between the leading research activities and the reality of Web service applications.

As future work assembling and executing composite services based on real-time Quality of Service measurements, acquired through interaction with the services, is proposed. The system learns about service capabilities and records their properties such as failure rate, latency and generated traffic. The measured Quality of Service information is then used for selecting service instances. Users can specify their Quality of Service requirements together with a `composition request`. The system examines the Quality of Service requirements and existing services and their capabilities to determine the probability with which the user requirements will be met. It is important that the system does not give a yes/no admission control decision to the user, rather it informs the user about the probability of success and lets the user choose whether to proceed.

To improve the probability of meeting Quality of Service requirements, the system may opt to run redundant instances of the composite service (or specific services) to ensure that the overall execution will not be drastically impaired by delays caused by the failure of one or more atomic services.

7.2.4 Privacy, security and trust for composite services

User context, such as location, activity, and social setting, is enormously sensitive information. Mechanisms for managing privacy either through access control management or by ensuring anonymity to prevent tracking of users and their requests, are essential.

Furthermore, authentication and trust management in a federated environment, where services and composition framework are owned and administered by different organisations, is a challenging problem. A federated authentication, authorisation and accounting (AAA) infrastructure is required to enable secure service composition and interoperation.

Web Service Security Language (WS-Security) [ADLH$^+$02] is an extension of SOAP developed to provide message integrity, confidentiality, and authentication. The WS-Security model has been designed to aid the implementation of Web services in a platform independent and loosely coupled manner. It also provides mechanisms for establishing secure communications, defining policies for how services interact, and defining rules of trust between domains of services. The WS-Security model contains several components, which reflect the constraints and capabilities of Web services, such as Web Service Policy Language (WS-Policy) and Web Service Trust Language (WS-Trust).

When reasoning about service composition, the composition engine component needs to evaluate reliability, interoperability, availability, fault tolerance, and performability of loosely coupled and distributed services. This requires a more complex model of trustworthiness.

7.3 Summary

This dissertation has proposed a new approach for developing context aware applications based on the idea of context aware service composition. It has presented and evaluated a framework that uses this approach and embodies new system design principles, which hold independently of the particular infrastructure implementing the framework.

The framework successfully uses AI planning to control service composition based on the user's context. Contextual changes may trigger recomposition of services during execution, causing the application to evolve dynamically.

Also, the framework has introduced a comprehensive composition failure management system, `GoalMorph`, grounded in the idea of context aware goal transformations. By means of a sample application this work has demonstrated that context aware service composition can work in practice and provides an efficient, scalable and practical approach for building context aware applications.

By developing the prototype context aware service composition framework, this dissertation has contributed to enabling the development of extensible, fault-tolerant, and evolving context aware applications. The research presented within the scope of the proposed framework represents a step towards a new model for developing adaptive applications, which coordinate a varying set of software components to realise the computational task in highly dynamic and unpredictable conditions.

Appendix A

Interfaces

This appendix presents the interfaces that each component exports and the functionality that it delivers. The way the operations are performed in the prototype implementation of the framework is described in Chapters 4 and 5.

A.1 Composition request management layer

The interfaces of the components that participate in the process of composition request management are shown in Figure 4.2.

Goal Service

The `Goal Service` exports the following interfaces for managing user tasks and `composition requests`.

1. `list_application_domains()`
   This method lists all available application domains, which are supported by the system, such as infotainment portal, mail replication, and smart home automation.

2. `get_available_tasks(application_domain)`
   `application_domain:` Dimension of search, such as entertainment services.

   This method lists the supported tasks in the selected application domain.

3. `select_task(task_id)`
   `task_id:` Desired task to be composed.

   This method sets the task to be composed.

4. `get_goals(task_id)`
   `task_id`: An identifier of the task.

   This method returns a formal representation of the goal conditions for this the task, specified by `task_id`, which form the core part of the `composition request`.

5. `transform_core_goal(request_id, goal_id)`
   `request_id`: An identifier of the `composition request` being transformed.
   `goal_id`: Desired core goal condition to be transformed.

   This method transforms the core goal conditions specified by `goal_id`. The value of substitution goal depends on the actual transformation employed, such as goal specialisation.

Context Service

The `Context Service` is a component responsible for acquisition and management of context. It allows other system components both to query and subscribe for notifications about context change. The interfaces it exports are:

1. `list_available_context_providers()`
   This method lists all the available context providers and corresponding context types.

2. `register_with_context_provider(context_type, context_provider, time_interval)`
   `context_type`: Desired context type to be monitored.
   `context_provider`: An optional argument identifying the context provider.
   `time_interval`: An option argument specifying the time interval in which the values should be retrieved.

   This method enables the client to subscribe to the context provider, defined by the `context_provider`, the selected context type in callback mode. As a result the client is notified of a new value, in the given `time_interval`.

3. `get_context_value(context_type, context_provider)`
   `context_type`: The type of the context whose value is to be accessed.
   `context_provider`: An optional argument identifying the provider.

   This method returns the current value of the desired `context_type`.

Context Proxy

The `Context Proxy` creates, manages and stores the context goal conditions, which describe application behaviour, such as "if the user is driving, then display the directions in speech form". The interfaces it exports are:

1. `get_context_goal(request_id, context_type, context_value)`
   `request_id:` The `composition request` (task type) for which the goal should be retrieved.
   `context_type:` The type of context for which the rule should be retrieved.
   `context_value:` The value of the context that triggers the goal.

   This method returns any goals that are triggered by the given context value of this `context_type`.

2. `create_context_goal(request_id, context_type, context_value, effect_type, effect)`
   `request_id:` A `composition request` with which a context goal should be associated.
   `context_type:` The type of context for which this goal should be created.
   `context_value:` The value of the context that triggers the rule.
   `effect_type:` There are two types of effect: one that denotes that a condition holds, called additive, and one that represents that a condition does not hold, called subtractive.
   `effect:` The description of the effect that must or must not hold at the end of the execution of the composite service.

   This method generates context goal condition in the `cogotag` format.

3. `update_context_goal(request_id, context_goal_id, context_type, context_value, effect_type, effect)`
   `request_id:` An identifier of the `composition request`.
   `context_goal_id:` An identifier of the context goal to be updated.
   `context_type:` A type of context for which the goal should be updated.
   `context_value:` A value of the context that triggers this goal.
   `effect_type:` The effects are of two types: ones that denote that a condition must hold and ones that denote that a condition must not hold.
   `effect:` The description of the effect that must or must not hold at the end of the execution of the `composition request`.

   This method updates the existing context goal condition.

4. `list_context_goal_conditions()`
   This method lists all available context goal conditions in the `Context Proxy`.

5. `list_context_goal_conditions(request_id)`
   `request_id:` An identifier of the `composition request` for which the context goal conditions should be listed.

   This method lists all the available context goal conditions stored in the `Context Proxy` associated with a specific `composition request`.

6. `list_context_rules(context_type, domain_id)`
   `context_type:` The type of context for which the rules should be listed.
   `domain_in:` An optional argument providing an identifier of the domain.)

   This method lists all the available context goal conditions for a specified `context_type`. This is used by `Context Mesh` to retrieve goal conditions, for new context types and values, which result from transformations.

Goal Transformation Engine

This component coordinates transformation of failed `composition requests` into ones that can be solved by the `Composition Engine`. Transformation is not only applied when composition process fails, but also, potentially to optimise an existing `composition request` and provide a more useful solution, for example with a higher Quality of Service.

To select suitable transformations, the `Goal Transformation Engine` employs several different methods including a utility function, a random search, or a domain-knowledge dependent method. A utility based search uses numerical values associated with each transformation, derived from planner feedback as utilities to guide the transformation selection process. The interfaces exported by this component are:

1. `transform_composition_request(request_id)`
   `request_id:` A `composition request` to be transformed.

   This method triggers the overall process of the `composition request` transformation. For each core and context goal condition it passes control to the `Goal Service` and the `Context Mesh` respectively.

2. `list_transformation_selection_methods()`

   This method lists available methods for the selection of transformations.

3. `add_transformation_selection_method (method_id, method_description, plug_in)`
   `method_id`: An identifier of the new transformation selection method.
   `method_description`: A method description.
   `plug_in`: The URL of the component implementing this transformation selection method.

   The method for transformation selection can be guided by different mechanisms, such as utility driven and random search.

4. `select_transformation_selection_method (method_id)`
   `method_id`: An identifier of the selected method.

   This method sets the method that will be used to select transformations.

5. `update_transformation_utilities (method_id, request_id, utilities)`
   `method_id`: A transformation method for which utilities should be updated.
   `request_id`: An identifier of the `composition request` for which utilities should be updated.
   `utilities`: A set of numerical values associated with transformations for this request.

   This method is used to update the utilities of each single transformation applied for a specific `composition request`.

6. `track_transformation_success (method_id, request_id, new_request_id, success)`
   `method_id`: An identifier of the transformation method for which utilities should be updated.
   `request_id`: An identifier of the original `composition request`.
   `new_request_id`: An identifier of the transformed composition request.
   `success`: A flag keeping track whether the overall transformation of the request was successful or not.

   This method keeps track of the success of each transformation request. The two methods above are invoked following composite service execution.

Context Mesh

The `Context Mesh` transforms context goal conditions. It interacts with the `Context Service` to retrieve context data and with the `Context Proxy` to generate transformed context goal conditions. There are several ways a context goal can be transformed, and these are discussed in detail in the Chapter 4.

1. `transform_context_goal(request_id, goal_id)`
   `request_id:` An identifier of the `composition request` being transformed.
   `goal_id:` Desired context goal condition to be transformed.

   This method transforms the context goal condition, specified by `goal_id`. It either returns a new context goal condition, which is generated by the `Context Proxy`, or completely removes the original goal condition from the `composition request`.

A.2 Abstract service composition layer

This layer is responsible for composition on the abstract level. It passes control to the composition request management layer if the request cannot be satisfied, and to the architecture specific service composition layer if the `abstract plan` was successfully generated.

Translation Module

The `Translation Module` converts the internal representation of the `composition request` into one supported by the composition methodology in use.

1. `generate_problem_definition(request_id)`
   `request_id:` An identifier of the `composition request` whose formal representation is to be converted.

   This method converts the `composition request`, specified by `request_id`, from an internal representation to the one supported by the `Composition Engine`.

2. `generate_domain_definition(abstract_services)`
   `abstract_services:` A list of available abstract service in the domain.

   This method converts abstract services in the domain from an internal representation to the one supported by the `Composition Engine`.

3. `list_conversion_methods()`

   This method lists supported composition methods and their respective plugins for conversion of representations.

4. `get_conversion_method(method_id)`
   `method_id:` A method whose conversion schema is to be retrieved.

   This returns the method for the conversion of the `composition request` and the domain for the composition technology in use.

5. `add_conversion_method(method_id, plug_in)`
   `method_id:` A composition method for which the plug_in is being added.
   `plug_in:` The URL of the component implementing this conversion method.

   This method facilitates integration of new conversion methods. As a result it system is extensible and open towards new composition approaches.

Abstract Service Repository

This component stores information about and provides retrieval of abstract services.

1. `list_available_service_types()`
   This lists available abstract service types in the domain.

2. `get_domain_description()`
   This returns the domain description encompassing abstract services in the representation format supported by the `Composition Engine`.

Composition Engine

This component locates, selects and composes abstract services that meet the constraints of a `composition request` to construct an `abstract plan`.

1. `compose_abstract_plan(problem_definition, domain_description)`
   `problem_definition:` A formal description of initial and goal states.
   `domain_description:` A formal description of available abstract service.

   This method performs the composition on an abstract level. It returns an `abstract plan` in the language of the composition method. Chapter 5 describes in more detail TLPlan representation of a `problem_definition`, `domain description` and `abstract plan`.

A.3 Architecture specific service composition layer

This layer converts an `abstract plan` to the one that can be deployed and executed, by instantiating abstract services.

Plan Translator

This component converts an `abstract plan` into an `abstract execution plan`.

1. `generate_abstract_execution_plan(abstract_plan, format)`
   `abstract_plan:` An abstract representation of the composite service generated by the `Composition Engine`.
   `format:` An identifier of the representation format used for translation.

   This method returns the `abstract execution plan` in the representation language supported by the execution environment, by converting the `abstract plan` generated by the `Composition Engine`.

Plan Instantiator

This component generates an executable and deployable service, based on the given `abstract plan`.

1. `instantiate(abstract_service, search_criteria)`
   `abstract_service:` A description of an abstract service, which forms the main criteria for finding matching instances.
   `search_criteria:` Additional criteria that must be satisfied by selected service, such as Quality of Service parameters.

   This method is used to contact the `Service Registry` in order to instantiate an abstract service.

2. `generate_deployable_service(service_description,`
   `service_binding)`
   `service_description:` Information on preconditions and postconditions that must hold before and after the execution of the service.
   `service_binding:` Information on how to access this service.

   This returns the service instance to be deployed and executed. In addition to the service binding information, required service preconditions and postconditions are also passed to the execution and monitoring layer.

Service Registry

A `Service Registry` is a network-based directory that contains information about available services. It is an entity that stores contracts from service providers and provides those contracts to interested service consumers.

1. `publish_service(service_id, service_data)`
   `service_id:` An identifier of the service to be published.
   `service_data:` Functional and binding information about this service.

   Service providers use this method to publish their services to the `Service Registry`, making them available for use by other service requestors.

2. `find_service(search_criteria, max_rows)`
   `service_description:` A set of service capabilities.
   `max_rows:` An optional argument specifying the maximum number of matching service instances to be returned.

   This lists all service instances that match the requested properties.

3. `get_service(service_id)`
   `service_id:` A unique identifier of the service to be accessed.

   This returns the binding information for the selected service.

A.4 Execution and monitoring layer

This layer invokes service instances and continuously monitors their execution.

Execution Engine

This component is an off-the-self run-time environment.

1. `execute_service(service_id)`
   `service_id:` An identifier of the service instance to be invoked.

   This is provided by the execution platform and is used to execute the service instance, specified by `service_id`.

Monitoring Engine

This provides monitoring capabilities to manage service execution.

1. `set_service_monitoring_procedure(service_id, monitoring_procedure_id)`
   `service_id:` An identifier of the service to be monitored.
   `monitoring_procedure_id:` An identifier of the monitoring procedure.

2. `set_event_monitoring_procedure(event_id, monitoring_procedure_id)`
   `event_id:` An identifier of the event to be monitored.
   `monitoring_procedure_id:` An identifier of the monitoring procedure to be applied.

   These two methods define services and events that the `Monitoring Engine` should observe. Monitoring procedures are described in detail in Section 5.3

3. `fire_service_failure(service_instance, failure_type, environment)`
   `service_id:` An identifier of service to be monitored.
   `failure_type:` There are two types of failure, a service failure, such as when service stops responding, or service unavailability, for example, if service cannot be initially invoked.
   `environment:` Describes the current state of the execution environment.

   This is activated when a service failure occurs and triggers the fault recovery process. Information about the environment is used to update the state of the `Execution Engine`.

4. `fire_event_failure(event_id, environment)`
   `event_id:` An identifier of the event.
   `environment:` Describes the current state of the execution environment.

   This activates when an unexpected event occurs. A description of the current environment is passed to the `Composition Engine` and `Execution Engine`.

Appendix B

System extensibility

Different application domains and composition problems may require the applied composition methodology to support different features, such as varying degree of observability. Furthermore, new component technologies and composition methodologies are routinely becoming available. For the framework to be considered general-purpose, no single service component technology and composition methodology can be assumed. The service composition framework therefore needs to be designed in an open way so that new features can be added easily, without requiring significant changes to the framework.

To facilitate extensibility, the system employs internal representations of the `composition request` and `abstract plan`. Translation modules are employed for each new composition method or component technology, which is introduced in the system, to represent the internal structure of the `composition request` and the `abstract plan`.

Independence of composition methodology

The `composition request` is a construct that specifies a user's computational task. The data structure used to represent it contains the following:

1. `request_id`:
   A unique identifier for the formalised `composition request`.

2. `task_id`:
   A unique identifier of the high-level task this request is associated with.

3. `application_domain_id`:
   A unique identifier of the application domain to which this `composition request` belongs to.

4. `core_goals`:
   A list of the core goal conditions.

5. `context_goals`:
   A list of the context goal conditions.

6. `priority_level`:
   The level of importance of this goal, used when several goals require system resources at the same time.

The central part of the `composition request` is goal conditions, whose description in the current implementation of the framework follows the formalism used in state transition systems [FN71]. Each goal condition consists of a type and a set of values. Values include primitives, such as integers and strings, structure values containing named components, and list values containing components. Every goal condition has an XML representation and every XML document in the correct format can be viewed as a goal condition value.

The `problem definition` is a set of goal conditions represented in XML form. The `domain description` is a list of semantically enriched abstract services, mapped to the abstract planning actions. The `Abstract Service Repository` is designed to cooperate with third-party mediators, supporting heterogeneous ontologies that different service providers may use to describe their capabilities.

The architecture is also independent of the component technology used, and in addition facilitates the use of multiple component technologies provided that the service description can be translated into the internal representation format. The `abstract plan` makes the system open to any composition methodology and further allows the coexistence of multiple composition methodologies, as different composition methodologies may be more applicable to the problem at hand.

Independence of execution environment

The `Plan Translator` transforms an `abstract plan` into an `abstract execution plan`. For each abstract service in the `abstract execution plan` The `Plan Instantiator` creates a `deployable service representation`, which contains:

1. a service binding in the implementation specific representation

2. service preconditions and postcondition

3. redundant services for substitution in case of failure

4. a monitoring procedure including recovery method

The `Monitoring Engine` is designed to wrap around the `Execution Engine` provided by the run-time environment. There are no architectural barriers to supporting multiple execution environments at the same time, assuming they can be accessed through interfaces defined in the architecture specific service composition layer.

In summary, the open and extensible design of the system ensures a low maintenance cost in terms of effort, because existing execution environments and available composition methods are supported and can easily be integrated, as the mechanisms for easy composition management, execution and monitoring are provided by the proposed framework.

Bibliography

[ABH^{+}02] Anupriya Ankolenkar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne, and Katia Sycara. DAML-S: Web Service Description for the Semantic Web. In *Proceedings of the First International Semantic Web Conference (ISWC)*, Sardinia, Italy, June 2002.

[ACD^{+}05] Vikas Agarwal, Girish B. Chafle, Koustuv Dasgupta, Sumit Mittal, and Biplav Srivastava. Evaluating Planning Based Approaches for End to End Composition and Execution of Web Services. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05). Workshop on Exploring Planning and Scheduling for Web Services, Grid and Autonomic Computing*, Pittsburgh, Pennsylvania, USA, 2005.

[ACK94] Abhaya Asthana, Mark Cravatts, and Paul Krzyzanowski. An Indoor Wireless System for Personalized Shopping Assistance. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, Santa Cruz, CA, USA, December 1994.

[ADLH^{+}02] Bob Atkinson, Giovanni Della-Libera, Satoshi Hada, Maryann Hondo, Phillip Hallam-Baker, Johannes Klein, Brian LaMacchia, Paul Leach, John Manferdelli, Hiroshi Maruyama, Anthony Nadalin, Nataraj Nagaratnam, Hemma Prafullchandra, John Shewchuk, and Dan Simon. Specification: Web Services Security (WS-Security). Available at `http://www-106.ibm.com/developerworks/webservices/library/ws-secure/`, April 2002. (Last accessed 1st March 2006).

[AJLS97] Mike Addlesee, Alan Jones, Finnbar Livesey, and Ferdinando

Samaria. ORL Active Floor. *IEEE Personal Communications*, 4(5):35–41, 1997.

[AVG$^+$04] Rama Akkiraju, Kunal Verma, Richard Goodwin, Prashant Doshi, and Juhnyoung Lee. 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, British Columbia, Canada, June 2004.

[BA01] Fahiem Bacchus and Michael Ady. Planning with Resources and Concurrency: A Forward Chaining Approach. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 417–424, Seattle, Washington, USA, August 2001.

[BBC$^+$01] Steve Benford, John Bowers, Paul Chandler, Luigina Ciolfi, Martin Flintham, Mike Fraser, Chris Greenhalgh, Tony Hall, Sten Olof Hellstrm, Shahram Izadi, Tom Rodden, Holger Schnädelbach, and Ian Taylor. Unearthing Virtual History: Using Diverse Interfaces To Reveal Hidden Virtual Worlds. In *Proceedings of the International Conference on Ubiquitous Computing (UbiComp) 2001*, pages 1–6. ACM, November 2001.

[BBG$^+$00] Guruduth Banavar, James Beck, Eugene Gluzberg, Jonathan Munson, Jeremy Sussman, and Deborra Zukowski. Challenges: An Application Model for Pervasive Computing. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MOBICOM)*, pages 266–274, Boston, Massachusetts, United States, 2000. ACM Press.

[BCGM03] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, and Massimo Mecella. Reasoning about Actions for e-Service Composition. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling. Workshop on Planning for Web Services*, Trento, Italy, June 2003.

[BCLP03] Piergiorgio Bertoli, Alessandro Cimatti, Ugo Dal Lago, and Marco Pistore. Extending PDDL to Nondeterminism, Limited Sensing and Iterative Conditional. In *Proceedings of The International Conference on Automated Planning and Scheduling (ICAPS). Work-*

shop on Planning Domain Description Language (PDDL), Trento, Italy, June 2003.

[BCP$^+$01] Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. MBP: a Model Based Planner. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI). Workshop on Planning under Uncertainty and Incomplete Information*, Seattle, Washington, USA, August 2001.

[Bei00] Michael Beigl. MemoClip: A Location-Based Remembrance Appliance. *Personal and Ubiquitous Computing*, 4(4):230–233, 2000.

[Ber05] Daniela Berardi. *Automatic Service Composition. Models, Techniques, Tools.* PhD thesis, University of Rome "La Sapienza", Rome, Italy, 2005.

[BF95] Avrim Blum and Merrick Furst. Fast Planning Through Planning Graph Analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, Montréal, Québec, Canada, 1995.

[BG98] Blai Bonet and Hector Geffner. HSP: Heuristic Search Planner. Entry at the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98), Planning Competition. Available at `http://www.ldc.usb.ve/~bonet/reports/aips98-competition.ps`, August 1998. (Last accessed 1st March 2006).

[BHB97] John Bates, David Halls, and Jean Bacon. Middleware Support for Mobile Multimedia Applications. *ICL Systems Journal*, 12(2):289–314, November 1997.

[BK95] Fahiem Bacchus and Froduald Kabanza. 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, Florida, USA, 1995.

[BRH94] Frazer Bennett, Tristan Richardson, and Andy Harter. Teleporting - Making Applications Mobile. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, USA, December 1994.

[Bro96] Peter J. Brown. The Stick-e Document: A Framework for Creating Context-Aware Applications. In *Electronic Publishing*, pages 259–272, Laxenburg, Austria, September 1996.

[BRSM03] Daniela Berardi, Fabio De Rosa, Luca De Santis, and Massimo Mecella. Finite State Automata as Conceptual Model for e-Services. In *Proceedings of the Seventh World Conference on Integrated Design and Process Technology (IDPT). Modeling and Developing Process-Centric Virtual Enterprises with Web-Services (VIEWS'03)*, Austin, Texas, USA, 2003.

[CAD$^+$05] Francisco Curbera, Tony Andrews, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. 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`, 2005. (Last accessed 1st March 2006).

[CBC$^+$04] Norman H. Cohen, James Black, Paul Castro, Maria Ebling, Barry Leiba, Archan Misra, and Wolfgang Segmuller. Building Context-Aware Applications with Context Weaver. Research Report RC 23388, IBM, October 2004.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Specification available at `http://www.w3.org/TR/wsdl`, 2001. (Last accessed 1st March 2006).

[Ccp99] Composite Capabilities/Preference Profiles. Website available at `http://www.w3.org/Mobile/CCPP/`, 1999. (Last accessed 1st March 2006).

[CFJ03] Harry Chen, Tim Finin, and Anupam Joshi. An Ontology for Context-Aware Pervasive Computing Environments. *Knowledge Engineering Review*, 18(3):197–207, 2003.

[CIJ$^+$00] Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and Dynamic Service Composition in eFlow. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 13–31, Stockholm, Sweden, June 2000.

[CJ01] Dipanjan Chakraborty and Anupam Joshi. Dynamic Service Composition: State-of-the-Art and Research Directions. Technical Report TR-CS-01-19, Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, Maryland, USA, 2001.

[CK00] Guanling Chen and David Kotz. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Department of Computer Science, Darmouth College, Hanover, New Hampshire, USA, November 2000.

[CKJH02] Ivica Crnković, Zeynep Kiziltan, Totte Jonsson, and Brahim Hnich. Specification of Software Components. In I. Crnković and M. Larsson, editors, *Building Reliable Component-Based Systems*, chapter 1, pages 5–22. Artech House, 2002.

[CMD99] Keith Cheverst, Keith Mitchell, and Nigel Davies. Design of an Object Model for a Context Sensitive Tourist GUIDE. *Computers and Graphics*, 23(6):883–891, 1999.

[Cor95] Microsoft Corporation. The Component Object Model Specification, Version 0.9. Available at `http://www.microsoft.com/com/resources/comdocs.asp`, October 1995. (Last accessed 1st March 2006).

[CPJ+02] Dipanjan Chakraborty, Filip Perich, Anupam Joshi, Timothy Finin, and Yelena Yesha. A Reactive Service Composition Architecture for Pervasive Computing Environment. In *Proceedings of the Seventh Personal Wireless Communications Conference (PWC 2002)*, Singapore, 2002.

[CV98] Michael T. Cox and Manuela M. Veloso. Goal Transformations in Continuous Planning. In *Proceedings of the American Association for Artificial Intelligence (AAAI) Fall Symposium on Distributed Continual Planning*, Orlando, Florida, USA, October 1998. AAAI Press.

[CvHH+01] Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. DAML+OIL Reference Description. Project Website available at `http://www.w3.org/TR/daml+oil-reference`, 2001. (Last accessed 1st March 2006).

[CZ04] Michael T. Cox and Chen Zhang. Planning as a Mixed-Initiative Goal Manipulation Process. Technical Report WSU-CS-04-02, Wright State University, Department of Computer Science and Engineering, Dayton, Ohio, USA, 2004.

[DA99] Anind K. Dey and Gregory D. Abowd. Towards a Better Understanding of Context and Context-Awareness. Technical Report GIT-GVU-99-22, Georgia Institute of Technology, College of Computing, Atlanta, Georgia, USA, June 1999.

[DA00] Anind K. Dey and Gregory D. Abowd. CybreMinder: A Context-Aware System for Supporting Reminders. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing (HUC2K)*, pages 172–186, Bristol, UK, 2000.

[DAS01] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction (HCI) Journal*, 16:97–166, 2001.

[DAW98] Anind K. Dey, Gregory D. Abowd, and Andrew Wood. CyberDesk: a Framework for Providing Self-Integrating Context-Aware Services. In *Proceedings of the Third International Conference on Intelligent User Interfaces*, pages 47–54, San Francisco, California, USA, 1998. ACM Press.

[DC02] Johnatan Dale and Luigi Ceccaroni. Pizza and a Movie: A Case Study in Advanced Web-Services. In *Proceedings of the International Conference on Autonomous Agents and Multiagents (AAMAS). Agentcities: Workshop on Challenges in Open Agent Systems*, Bologna, Italy, 2002.

[Der99] Michael L. Dertouzos. The Future of Computing. *Scientific American*, 281(2):52–63, August 1999.

[Dey00] Anind K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, Georgia Institute of Technology, Atlanta, Georgia, USA, 2000.

[DGAV04] Prashant Doshi, Richard Goodwin, Rama Akkiraju, and Kunal Verma. Dynamic Workflow Composition using Markov Decision Processes. In *Proceedings of the IEEE International Conference* *on Web Services (ICWS), Industrial Track*, San Diego, California, USA, July 2004.

[DK75] Frank DeRemer and Hans Kron. Programming-in-the-large versus Programming-in-the-small. *ACM Special Interest Group on Programming Languages Notices (ACM SIGPLAN Notices)*, 10(6):114–121, 1975.

[EHAB99] Mike Esler, Jeffrey Hightower, Tom Anderson, and Gaetano Borriello. Next Century Challenges: Data-centric Networking for Invisible Computing: the Portolano Project at the University of Washington. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 256–262, Seattle, Washington, United States, August 1999. ACM Press.

[EHL01] Maria R. Ebling, Guerney D. H. Hunt, and Hui Lei. Issues for Context Services for Pervasive Computing. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, November 2001.

[FB02] Dieter Fensel and Christoph Bussler. The Web Service Modeling Framework (WSMF). *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[FFK$^+$02] Margaret Fleck, Marcos Frid, Tim Kindberg, Eamonn O'Brien-Strain, Rakhi Rajani, and Mirjana Spasojevic. From Informing to Remembering: Ubiquitous Systems in Interactive Museums. *Pervasive Computing*, 1(2):12–21, 2002.

[FGC$^+$97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *Symposium on Operating Systems Principles*, pages 78–91, 1997.

[FK05] Jianchun Fan and Subbarao Kambhampati. A Snapshot of Public Web Services. *ACM Special Interest Group on Management Of Data (ACM SIGMOD) Record*, 34(1):24–32, 2005.

[FL03] Maria Fox and Derek Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[Fla04] David Flanagan. *Java in a Nutshell.* O'Reilly Media, Inc., Sebastopol, California, USA, 5th edition, 2004.

[FN71] Richard Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artifical Intelligence*, 2(3–4):189–208, 1971.

[For82] Charles Forgy. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[GBK99] Hans-W. Gellersen, Michael Beigl, and Holger Krull. The MediaCup: Awareness Technology Embedded in an Everyday Object. In *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing (HUC)*, pages 308–310, Karlsruhe, Germany, 1999. Springer-Verlag.

[GC92] David Gelernter and Nicholas Carriero. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, 1992.

[GGKS02] Karl Gottschalk, Stephen Graham, Heather Kreger, and James Snell. Introduction to Web Services Architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[GHK$^+$98] Malik Ghallab, Adele Howe, Craig A. Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wikins. PDDL—The Planning Domain Definition Language. The International Conference on Artificial Intelligence Planning Systems (AIPS-98) Planning Competition Language Specifications. Specification available at `ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz`, 1998. (Last accessed 1st March 2006).

[GL05] Alfonso Gerevini and Derek Long. Plan Constraints and Preferences in PDDL3. Technical Report R.T. 2005-08-47, Department of Electronics for Automation, University of Brescia, Brescia, Italy, August 2005.

[GLL00] Giuseppe De Giacomo, Yves Lesperance, and Hector J. Levesque. Congolog, a Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning, Theory and Practice.* Elsevier, 2004.

[GNY04] Xiaohui Gu, Klara Nahrstedt, and Bin Yu. SpiderNet: an Integrated Peer-to-Peer Service Composition Framework. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 110–119, Honolulu, Hawaii, USA, June 2004.

[Gou04] Jonathan Gough. XPDDL: The eXtensible Planning Domain Definition Language. Schema available at `http://www.cis.strath.ac.uk/~jg/XPDDL/`, 2004. (Last accessed 1st March 2006).

[GSSS02] David Garlan, Daniel P. Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.

[HBS02] Albert Held, Sven Buchholz, and Alexander Schill. Modeling of Context Information for Pervasive Computing Applications. In *Proceedings of the Sixth World Multiconference on Systemics, Cybernetics and Informatics (SCI2002)*, Orlando, Florida, USA, 2002.

[HC02] Robert Headon and Rupert Curwen. Movement Awareness for Ubiquitous Game Control. *Personal and Ubiquitous Computing*, 6(5-6):407–415, 2002.

[HH93] Peter Haddawy and Steve Hanks. Utility Models for Goal-Directed Decision Theoretic Planners. Technical Report TR-93-06-04, University of Washington, Department of Computer Science and Engineering, Seattle, Washington, USA, 1993.

[HHS$^+$99] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The Anatomy of a Context-Aware Application. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 59–68, Seattle, Washington, United States, August 1999.

[HK97] Markus Horstmann and Mary Kirtland. DCOM Architecture. Technical article, Microsoft Corporation, Available at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomarch.asp`, July 1997. (Last accessed 1st March 2006).

[HL01] Jason I. Hong and James A. Landay. An Infrastructure Approach to Context-Aware Computing. *Human-Computer Interaction (HCI) Journal*, 16:287–303, 2001.

[Hol05] David Hollingsworth. The Workflow Management Coalition Specification. Specifications available at `http://www.wfmc.org/standards/docs/tc003v11.pdf`, 2005. (Last accessed 1st March 2006).

[HRC02] Christopher K. Hess, Manuel Román, and Roy H. Campbell. Building Applications for Ubiquitous Computing Environments. In *Proceedings of the First International Conference on Pervasive Computing*, Lecture Notes in Computer Science, pages 16–29, Zurich, August 2002. Springer-Verlag.

[HS94] Peter Haddawy and Meliani Suwandi. Decision-Theoretic Refinement Planning using Inheritance Abstraction. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 266–271, Chicago, Illinois, June 1994. AAAI Press.

[HV96] Karen Zita Haigh and Manuela Veloso. 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*, pages 35–44, Stanford University in Palo Alto, California, USA, March 1996. AAAI Press, Menlo Park, California.

[HZB$^+$06] Duncan Hull, Evgeny Zolin, Andrey Bovykin, Ian Horrocks, Ulrike Sattler, and Robert Stevens. Deciding semantic matching of stateless services. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)*, 2006. To appear.

[IBM04] IBM. The IBM Business Process Execution Language for Web Services Java^{TM} Run Time (BPWS4J). Software available at `http://www.alphaworks.ibm.com/tech/bpws4j`, 2004. (Last accessed 1st March 2006).

[Ing78] Daniel H. H. Ingalls. The Smalltalk-76 Programming System Design and Implementation. In *Proceedings of the Fifth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 9–16, Tucson, Arizona, USA, 1978. ACM Press.

[IRRH03] Jadwiga Indulska, Ricky Robinson, Andry Rakotonirainy, and Karen Henricksen. Experiences in using CC/PP in Context-Aware Systems. In M. S. Chen and P.K. Chrysanthis and M. Sloman and A. Zaslavsky, editor, *Proceedings of the Fourth International Conference on Mobile Data Management (MDM2003)*, pages 247–261, Melbourne, Australia, 2003. Lecture Notes in Computer Science (LNCS), Springer.

[JCo02] JConfig. Software available from `http://tolstoy.com/samizdat/jconfig.html`, 2002. (Last accessed 19th July 2006).

[Jon02] Martin Jonsson. Context Shadow: A Person-Centric Infrastructure for Context Aware Computing. In *Proceedings of the Artificial Intelligence in Mobile Systems Workshop in Conjuction with European Conference on Artificial Intelligence (ECAI)*, Lyon, France, July 2002.

[Jud03] jUDDI v.0.94rc: Java Implementation of the Universal Description Discovery, and Integration (UDDI) Specification for Web Services. Apache Web Services Project. Software available at `http://ws.apache.org/juddi/`, 2003. (Last accessed 1st March 2006).

[KBSD97] Froduald Kabanza, M. Barbeau, and Richard St-Denis. Planning Control Rules for Reactive Agents. *Artificial Intelligence*, 95(1):67–113, 1997.

[KN04] Ugur Kuter and Dana S. Nau. Forward-Chaining Planning in Nondeterministic Domains. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, pages 513–518, San Jose, California, USA, June 2004.

[Koe99] Jana Koehler. Handling of Conditional Effects and Negative Goals in IPP. Technical Report 128/99, University of Freiburg, Freiburg, Germany, 1999.

[KS03] Jana Koehler and Biplav Srivastava. Web Service Composition: Current Solutions and Open Problems. In *The International Conference on Automated Planning and Scheduling (ICAPS). Workshop on Planning for Web Services*, pages 28–35, Trento, Italy, 2003.

[KV05] Evangelos Kotsovinos and Maja Vuković. su-chef: Adaptive Coordination of Intelligent Home Environments. In *Proceedings of the Joint International Conference on Autonomic and Autonomous Systems 2005 / International Conference on Networking and Services 2005 (ICAS/ICNS 2005)*, Papeete, Tahiti, October 2005. IEEE Computer Society.

[LF94] Mik Lamming and Mike Flynn. Forget-me-not: Intimate Computing in Support of Human Memory. In *Proceedings of the FRIEND21 Symposium on Next Generation Human Interfaces*, Tokyo, Japan, 1994.

[LKAA96] Sue Long, Rob Kooper, Gregory D. Abowd, and Christopher G. Atkeson. Rapid Prototyping of Mobile Context-Aware Applications: The Cyberguide Case Study. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 97–107, White Plains, New York, USA, 1996. ACM Press.

[LPT02] Ugo Dal Lago, Marco Pistore, and Paolo Traverso. Planning with a Language for Extended Goals. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 447–454, Edmonton, Alberta, Canada, 2002. AAAI.

[LRL$^+$97] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[LSD$^+$02] Hui Lei, Daby M. Sow, John S. Davis II, Guruduth Banavar, and Maria R. Ebling. The Design and Applications of a Context Service. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):45–55, 2002.

[Maa97] Henning Maass. Location-Aware Mobile Applications Based on Directory Services. In *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 23–33, Budapest, Hungary, 1997. ACM Press.

[MBH$^+$04] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia

Sycara. OWL-S: Semantic Markup for Web Services. White paper. Available at `http://www.w3.org/Submission/OWL-S`, 2004. (Last accessed 1st March 2006).

[MBW$^+$98] Elizabeth D. Mynatt, Maribeth Back, Roy Want, Michael Baer, and Jason B. Ellis. Designing Audio Aura. In *Proceeding of the Conference on Human Factors in Computing Systems (CHI 98)*, pages 566–573, Los Angeles, California, USA, April 1998.

[MF02] Sheila A. McIlraith and Ronald Fadel. Planning with Complex Actions. In Salem Benferhat and Enrico Giunchiglia, editors, *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR 2002)*, pages 356–364, Toulouse, France, April 2002.

[MH69] John McCarthy and Patrick J. Hayes. Some Philosophical Problems From the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[MH04] Julie A. McCann and Markus C. Huebscher. Evaluation Issues in Autonomic Computing. In Hai Jin, Yi Pan, and Nong Xiao, editors, *Proceedings of the Third International Conference on Grid and Cooperative Computing Workshops (GCC)*, volume 3252 of *Lecture Notes in Computer Science*, pages 597–608, Wuhan, China, 2004. Springer.

[MKB01] Z. Morley Mao, Randy H. Katz, and Eric A. Brewer. Fault-tolerant, Scalable, Wide-Area Internet Service Composition. Technical Report UCB/CSD-01-1129, EECS Department, University of California, Berkeley, California, USA, 2001.

[MM03] Daniel J. Mandell and Sheila A. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *Proceedings of the Second International Semantic Web Conference*, pages 227–241, Sanibel Island, Florida, USA, October 2003.

[MPL03] Ryusuke Masuoka, Bijan Parsia, and Yannis Labrou. Task Computing - The Semantic Web Meets Pervasive Computing. In Dieter

Fensel, Katia P. Sycara, and John Mylopoulos, editors, *Proceedings of the Second International Semantic Web Conference*, pages 866–881, Sanibel Island, Florida, USA, October 2003.

[MR91] David A. McAllester and David Rosenblitt. Systematic Nonlinear Planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 634–639, Anaheim, California, USA, 1991.

[MS85] Mike Muuss and Terry Slattery. ttcp: Test TCP. US Army Ballistics Research Lab (BRL). Enhanced version by Silicon Graphics Incorporated, October 1991. Software available from `ftp://ftp.sgi.com/sgi/src/ttcp`, November 1985. (Last accessed 1st March 2006).

[MS00] Natalia Marmasse and Chris Schmandt. Location-Aware Information Delivery with ComMotion. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing (HUC2K)*, pages 157–171, Bristol, UK, 2000.

[MS02] Sheila McIlraith and Tran Cao Son. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002)*, Toulouse, France, April 2002.

[Nin97] The Ninja Project Enabling Internet-scale Services from Arbitrarily Small Devices. Project Website available at `http://ninja.cs.berkeley.edu/`, 1997. (Last accessed 1st March 2006).

[NM02] Srini Narayanan and Sheila McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proceedings of the 11th International World Wide Web Conference (WWW 2002)*, Honolulu, Hawaii, USA, May 2002.

[NMAC$^+$01] Dana S. Nau, Héctor Muñoz-Avila, Yue Cao, Amnon Lotem, and Steven Mitchell. Total-Order Planning with Partially Ordered Subtasks. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 425–430, Seattle, Washington, USA, August 2001.

[NSBW00] Joern Nilsson, Tomas Sokoler, Thomas Binder, and Nina Wetcke. Beyond the Control Room: Mobile Devices for Spatially Distributed Interaction on Industrial Process Plants. In *Proceedings of* *the Second International Symposium on Handheld and Ubiquitous Computing (HUC2K)*, pages 30–45, Bristol, UK, 2000.

[Obj91] Object Management Group and X/Open. The Common Object Request Broker: Architecture and Specification (CORBA). Technical Report 91.12.1, Object Management Group (OMG), 1991. Available from `http://www.omg.org`.

[Ols91] Dan R. Jr Olsen. *User Interface Management Systems: Models and Algorithms*. The Morgan Kaufmann, 1991.

[OS00] Reinhard Oppermann and Marcus Specht. A Context-Sensitive Nomadic Information System as an Exhibition Guide. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing (HUC2K)*, pages 127–142, Bristol, UK, 2000.

[Par72] David L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Pas97] Jason Pascoe. The Stick-e Note Architecture: Extending the Interface Beyond the User. In Johanna Moore, Ernest Edmonds, and Angel Puerta, editors, *Proceedings of the International Conference on Intelligent User Interfaces*, pages 261–264, Orlando, Florida, USA, January 1997. ACM.

[PBB$^+$04] Marco Pistore, Fabio Barbon, Piergiorgio Bertoli, D. Shaparau, and Paolo Traverso. Planning and Monitoring Web Service Composition. In *Proceedings of the Artificial Intelligence: Methodology, Systems, and Applications, 11th International Conference (AIMSA)*, pages 106–115, Varna, Bulgaria, September 2004.

[Ped94] Edwin P. D. Pednault. ADL and the State-Transition Model of Action. *Journal of Logic and Computation*, 4(5):467–512, 1994.

[PF02] Shankar R. Ponnekanti and Armando Fox. SWORD: A Developer Toolkit for Web Service Composition. In *Proceedings of the 11th World Wide Web Conference (Web Engineering Track)*, Honolulu, Hawaii, USA, May 2002.

[PKPS02] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Importing the Semantic Web in UDDI. In *Proceedings* *of the Workshop on Web Services, E-Business, and the Semantic Web (WES)*, pages 225–236, Toronto, Ontario, Canada, May 2002.

[PLF$^+$01] Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *Proceedings of the Third International Conference on Ubiquitous Computing (UbiComp)*, pages 56–75, Atlanta, Georgia, USA, 2001. Springer-Verlag.

[PSGS04] Vahe Poladian, Joao Pedro Sousa, David Garlan, and Mary Shaw. Dynamic Configuration of Resource-Aware Services. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 604–613, Washington, DC, USA, 2004. IEEE Computer Society.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., 1994.

[PV04] Joachim Peer and Maja Vuković. A Proposal for a Semantic Web Service Description Format. In Liang-Jie Zhang, editor, *Proceedings of the European Conference On Web Services (ECOWS)*, volume 3250 of *Lecture Notes in Computer Science*, pages 285–299, Erfurt, Germany, 2004. Springer.

[PW92] J. Scott Penberthy and Daniel S. Weld. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114, Cambridge, Massachusetts, USA, 1992.

[RAC$^+$02] Bhaskaran Raman, Sharad Agarwal, Yan Chen, Matthew Casar, Weidong Cui, Per Johansson, Kevin Lai, Tal Lavian, Sridhar Machiraju, Z. Morley Mao, Lakshimanrayanan Subramanian, Takashi Suzuki, Shelley Zhuang, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. The SAHARA Model for Service Composition across Multiple Providers. In *Proceedings of the First International Conference on Pervasive Computing*, volume 2414 of *Lecture Notes in Computer Science*, pages 1–14, Zurich, Switzerland, August 2002. Springer-Verlag. Invited paper.

[RC03] Anand Ranganathan and Roy H. Campbell. What is the Complexity of a Distributed System? Technical Report UIUCDCS-R-2005-2568, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, USA, April 2003.

[RHC+02] Manuel Roman, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: a Middleware Platform for Active Spaces. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):65–67, 2002.

[RKL+05] Dumitru Roman, Uwe Keller, Holger Lausen, Rubén Lara Jos de Bruijn, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

[RN95] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.

[RPM98] Nick S. Ryan, Jason Pascoe, and David Morse. Enhanced Reality Fieldwork: the Context-aware Archaeological Assistant. In V. Gaffney, M. van Leusen, and S. Exxon, editors, *Proceedings of the 25th Computer Applications in Archaeology 1997*, British Archaeological Reports, Oxford,UK, October 1998.

[Rya99] Nick Ryan. ConteXtML: Exchanging Contextual Information between a Mobile Client and the FieldNote Server. Language specifications available at `http://www.cs.kent.ac.uk/projects/mobicomp/fnc/ConteXtML.html`, 1999. (Last accessed 1st March 2006).

[SAB02] Ted Selker, Ernesto Arroyo, and Win Burleson. Chameleon Tables: Using Context Information in Everyday Objects. In *Extended Abstracts on Human Factors in Computer Systems (CHI)*, pages 580–581, Minneapolis, Minnesota, USA, 2002. ACM Press.

[SAT+99] Albrecht Schmidt, Kofi Asante Aidoo, Antti Takaluoma, Urpo Tuomela, Kristof Van Laerhoven, and Walter Van de Velde. Advanced Interaction in Context. In *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing (HUC)*, pages 89–101, Karlsruhe, Germany, 1999. Springer-Verlag.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-Aware Computing Applications. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, USA, 1994.

[Sch95] Bill N. Schilit. *System Architecture for Context-Aware Mobile Computing.* PhD thesis, Columbia University, New York, USA, 1995.

[SD91] Bill Schilit and Dan Duchamp. Adaptive Remote Paging for Mobile Computers. Technical Report CUCS-004-91, Columbia University Computer Science Department, New York, USA, February 1991.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, Inc., 1996.

[SLP04] Thomas Strang and Claudia Linnhoff-Popien. A Context Modeling Survey. In *Proceedings of the Workshop on Advanced Context Modelling, Reasoning and Management as Part of The Sixth International Conference on Ubiquitous Computing (UbiComp)*, Nottingham, UK, September 2004.

[Soa00] SOAP Specifications. Website available at `http://www.w3.org/TR/soap/`, 2000. (Last accessed 1st March 2006).

[SP97] Clemens Szyperski and Cuno Pfister. Summary of Workshop on Component-Oriented Programming (WCOP96). In M. Mühlhaeuser, editor, *Special Issues in Object-Oriented Programming. Prooceedings of the European Conference on Object-Oriented Programming (ECOOP). Workshop Reader*, pages 127–130, Linz, Austria, 1997. dpunkt Verlag.

[SPH05] Evren Sirin, Bijan Parsian, and Jim Hendler. Template-based Composition of Semantic Web Services. In *Proceedings of the 2005 AAAI Fall Symposium Series on Agents and the Semantic Web*, Menlo Park, CA, 2005. AAAI Press / The MIT Press.

[SPP$^+$03] Umar Saif, Hubert Pham, Justin Mazzola Paluska, Jason Waterman, Chris Terman, and Steve Ward. A Case for Goal-oriented Programming Semantics. In *Proceedings of the Fifth Annual Conference on Ubiquitous Computing, Workshop on System Support for Ubiquitous Computing (UbiSys)*, Seattle, Washington, USA, 2003.

[Sri04] Biplav Srivastava. A software framework for applying planning techniques. Research Report RI 04001, IBM, March 2004.

[SRvS03] Marta Sabou, Debbie Richards, and Sander van Splunter. An Experience Report on Using DAML-S. In *Proceedings of the Twelfth International World Wide Web Conference, Workshop on E-Services and the Semantic Web (ESSW'03)*, Budapest, Hungary, May 20-24 2003.

[ST94] Bill Schilit and M. Theimer. Disseminating Active Map Information to Mobile Hosts. *IEEE Network*, 8(5):22–32, 1994.

[STM00] Albrecht Schmidt, Antti Takaluoma, and Jani Mantyjarvi. Context-Aware Telephony Over WAP. *Personal Ubiquitous Comput.*, 4(4):225–229, 2000.

[SW95] Bill Schilit and Roy Want. The Xerox PARCTAB. Project Website available at `http://www.ubiq.com/parctab/`, 1995. (Last accessed 1st March 2006).

[Szy00] Clemens Szyperski. *Component Software and The Way Ahead*, chapter 1, pages 1–20. Cambridge University Press, 2000.

[TKA03] Snehal Thakkar, Craig A. Knoblock, and José Luis Ambite. A View Integration Approach to Dynamic Composition of Web Services. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling. Workshop on Planning for Web Services*, Trento, Italy, June 2003.

[Tom06] Apache Tomcat v5.5 Application Server. Software available at `http://tomcat.apache.org/`, 2006. (Last accessed 1st March 2006).

[TPC$^+$05] Michele Trainotti, Marco Pistore, Gaetano Calabrese, Gabriele Zacco, Gigi Lucchese, Fabio Barbon, Piergiorgio Bertoli, and Paolo Traverso. ASTRO: Supporting Composition and Execution of Web Services. In *Proceedings of the International Conference on Automated and Planning Sheduling (ICAPS). Demo*, Monterey, California, USA, June 2005.

[TYB$^+$01] John C. Tang, Nicole Yankelovich, James Begole, Max Van Kleek, Francis Li, and Janak Bhalodia. Connexus To Awarenex: Extending Awareness to Mobile Users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 221–228, Seattle, Washington, United States, 2001. ACM Press.

[Udd00] Universal Description, Discovery Integration(UDDI). Project Website available at `http//www.uddi.org`, 2000. (Last accessed 1st March 2006).

[Uns98] The United Nations Standard Products and Services Code UNSPSC. Website available at `http://www.unspsc.org/`, 1998. (Last accessed 1st March 2006).

[VCP+95] Manuela Veloso, Jaime Carbonell, Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating Planning and Learning: The PRODIGY Architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[vdADtH03] Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Web Service Composition Languages: Old Wine in New Bottles? In *Proceedings of the 29th EUROMICRO Conference 2003, New Waves in System Architecture*, pages 298–307, Belek-Antalya, Turkey, September 2003. IEEE Computer Society.

[vdBNDK04] Menkes van den Briel, Romeo Sanchez Nigenda, Minh Binh Do, and Subbarao Kambhampati. Effective Approaches for Partial Satisfaction (Over-Subscription) Planning. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, pages 562–569, San Jose, California, USA, July 2004.

[VR04a] Maja Vuković and Peter Robinson. Adaptive, Planning Based, Web Service Composition for Context Awareness. In Gabriele Kotsis, editor, *Advances in Pervasive Computing. A Collection of Contributions Presented at PERVASIVE 2004*, volume 176, pages 247–252. Oesterreichische Computer Gesellschaft (Hrsg.), April 2004.

[VR04b] Maja Vuković and Peter Robinson. Application Modeling for Context Awareness. Building and Evaluating Ubiquitous System Software. (Work in Progress Section). *IEEE Pervasive Computing Magazine*, 3(3):Page 59, July-October 2004.

[VR05a] Maja Vuković and Peter Robinson. Context Aware Service Composition. In *Proceedings of the Third UK UbiNet Workshop*, Bath, UK, 2005.

[VR05b] Maja Vuković and Peter Robinson. GoalMorph: Partial Goal Satisfaction for Flexible Service Composition. *International Journal of Web Services Practices*, 1(1–2):40–56, December 2005.

[VR05c] Maja Vuković and Peter Robinson. SHOP2 and TLPlan for Proactive Service Composition. In *Proceedings of the UK-Russia Workshop on Proactive Computing*, Nizhniy Novgorod, Russia, February 2005.

[Wei91] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, January 1991.

[WG00] Zhenyu Wang and David Garlan. Task-Driven Computing. Technical Report CMU-CS-00-154, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, May 2000.

[WHFG92] Roy Want, Andy Hopper, Veronica Falco, and Jonathan Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems (TOIS)*, 10(1):91–102, 1992.

[Wil94] Mike Williamson. Optimal Planning With a Goal-Directed Utility Model. In Kristian J. Hammond, editor, *Proceedings of the Second International Conference on AI Planning Systems*, pages 176–181, Chicago, Illinois, USA, 1994.

[Win01] Terry Winograd. Architectures for Context. *Human-Computer Interaction (HCI) Journal*, 16:401–419, 2001.

[WJH97] Andy Ward, Alan Jones, and Andy Hopper. A New Location Technique for the Active Office. *IEEE Personal Communications*, 4(5):42–47, October 1997.

[WMLF98] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel A. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. In *Proceedings of the Second Conference on Object-Oriented Technologies (COOTS)*, pages 219–231, Toronto, Ontario, Canada, June 1996. USENIX.

[WSA$^+$95] Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis, and Mark Weiser. The ParcTab Ubiquitous Computing Experiment. Technical Report CSL-95-1, Xerox Palo Alto Research Center, Palo Alto, California, USA, March 1995.

[Wsc02] Web Service Choreography Interface (WSCI) 1.0. Website available at `http://www.w3.org/TR/wsci/`, 2002. (Last accessed 1st March 2006).

[Wsd05] Web Services Description Language for Java. Software available at `http://sourceforge.net/projects/wsdl4j`, 2005. (Last accessed 1st March 2006).

[WSH$^+$03] Dan Wu, Evren Sirin, James Hendler, Dana Nau, and Bijan Parsia. 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, June 2003.

[Wsm04] Web Service Modelling Ontology. Website available at `http://www.wsmo.org/`, 2004. (Last accessed 1st March 2006).

[YL04] Hakan L. S. Younes and Michael L. Littman. PPDDL1.0: An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects. Technical Report CMU-CS-04-167, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, October 2004.

[YYDW99] Jie Yang, Weiyi Yang, Matthias Denecke, and Alex Waibel. Smart Sight: A Tourist Assistant System. In *Proceedings of the Third International Symposium on Wearable Computers (ISWC)*, Pittsburgh, Pennsylvania, USA, October 1999.

[ZBL$^+$03] Liangzhao Zeng, Boualem Benatallah, Hui Lei, Anne H. H. Ngu, David Flaxer, and Henry Chang. Flexible Composition of Enterprise Web Services. *Electronic Markets*, 13(2), 2003.

[ZBN$^+$04] Liangzhao Zeng, Boualem Benatallah, Anne H. H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.

[ZZ99] Tim Zagat and Nina Zagat. ZagatSurvey: Restaurant, Nightlife, Hotels, Attractions. Website available at `http://www.zagat.com`, 1999. (Last accessed 1st March 2006).