SYNTHESIZING AUTONOMIC COMPOSITIONS IN GRID ENVIRONMENT

BY MANISH AGARWAL

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

and approved by

____

____

____

New Brunswick, New Jersey

October, 2003

ABSTRACT OF THE THESIS

Synthesizing Autonomic Compositions in Grid Environment

by Manish Agarwal

Thesis Director: Professor Manish Parashar

Dynamic composition of applications from components and services is becoming attractive, and requires advance composition models. Service composition can be simply defined as the process of taking existing services and combining them (based on definitions and constraints) to form new services. In existing composition model, the composer identifies relevant services, explicitly states their interactions and creates a composition script. A flow engine then invokes this composition. Unfortunately this relatively static composition approach is not very scalable and manual choreography of compositions and interactions is not very realistic. Furthermore the assumption that the composer has a priori knowledge about the composition goals, the participating services and their interaction patterns is not valid for pervasive computing environments and dynamic Grid applications, motivating the need for more flexible and dynamic composition models. In such a model, composition plans are created at runtime based on dynamically defined composition objectives, their semantic descriptions, constraints, and available services and resources.

In this work we propose a dynamic composition model based on relational algebra

and graph theory. Services are described using standard *Web Service Description Language* (WSDL)and extended with semantic metadata (keywords). Relational joins are then used to generate composition plans and choreograph ad-hoc interactions at runtime to satisfy the composers objectives and constraints. Alternate plans may be evaluated and ranked based on different factors. In this work, we presents the design and operation of *Accord Composition Engine* (ACE). ACE address dynamic service composition. The overall goal of ACE is to autonomically synthesize composition plans, when possible, from available pool of services based on dynamically defined objectives and constraints. ACE is a key component of the Accord composition framework in Project AutoMate. The overall objective of AutoMate is to investigate key technologies to enable the development of autonomic Grid applications that are context aware and are capable of self-configuring, self-composing, self-optimizing and self-adapting.

Acknowledgements

I would like to acknowledge my family in India who have been supportive of my efforts. I would like to thank my research advisor Dr. Manish Parashar for his invaluable guidance, support and encouragement during the course of this work. I wish to thank all my friends and members of the TASSL Laboratory, and especially Viraj Bhat for his helpful advice. To my close friends Krishna, Murali, Lalit and my relatives in United States for making my life easier in this foreign country. Finally my special thanks to CAIP and all its staff, who have always promptly helped me in resolving both administrative and systems problems.

Dedication

To my Parents and Harsh Uncle

Table of Contents

Abstract ii

Acknowledgements iv

Dedication v

List of Tables viii

List of Figures ix

1. Introduction 1

1.1. Motivation 1

1.1.1. Why We Need Composition? 1

1.1.2. What We Need In Grid and Autonomic Environment? 2

1.1.3. Why We Need Dynamic Service Composition? 2

1.2. Problem Statement 3

1.3. Contributions of the Thesis 3

1.4. Organization 4

2. Prior Work 6

2.1. Composition 6

2.2. Composition Languages 7

2.3. Composition in Grid and Commercial Environment 8

3. AutoMate: An Autonomic Component Framework 10

3.1. AutoMate Architecture 11

3.2. Autonomic Components in AutoMate 12

3.3. Autonomic Compositions in AutoMate 14

4. The Accord Autonomic Composition Model 17

5. Accord Composition Engine 22

5.1. Architecture Overview 22

5.1.1. ACE Translator 23

5.1.2. ACE Graph Generator 24

5.1.3. ACE Constraint Satisfaction Module 24

5.1.4. Dynamic Service Plan Generator and Evaluator 25

5.2. Operation 25

5.3. Prototype in Action 26

6. Conclusion and Future Work 31

6.1. Discussion - Advantages and Limitations 31

6.2. Conclusion 32

6.3. Future Work 32

Appendix A. Sample service interfaces of services 34

References 36

List of Tables

4.1. Service Pool for the example travel guide service. 20

4.2. Interaction table for the example travel guide service. 21

5.1. Sample Composition Scenarios 28

5.2. Participating services for composition scenarios 29

5.3. Interaction Links created by ACE Graph Generator 29

5.4. Source and Sink Services for Travel Guide Example 30

List of Figures

3.1. AutoMate Architecture Diagram 11
3.2. An AutoMate Component 13
3.3. Autonomic Compositions in AutoMate 15
4.1. Accord composition algorithm 17
4.2. Accord algorithm in action 18
5.1. Architectural overview of the Accord Composition Engine (ACE). . . . 22
5.2. Schemas for ACE service and message description tables. 23
5.3. Schema for ACE link description table. 24
5.4. Operation of the Accord Composition Engine. 25
5.5. Interface of services in Service Pool. 27
5.6. Composition Graph Instances. 30
A.1. WSDL Interface of Location Service. 34
A.2. WSDL Interface of Vehicle Dependent Driving Service. 35
A.3. WSDL Interface of Driving Direction Service. 35

Chapter 1
Introduction

In this research we developed mechanisms and supporting infrastructure to enable autonomic applications to be dynamically and opportunistically composed from autonomic components [1]. The composition is based on policies and constraints that are defined, deployed and executed at runtime, allowing the composition to be aware of the current state, requirement, capabilities, available resources and services. We present the design and operation of the *Accord Composition Engine* (ACE) [33] for the dynamic composition of Grid services [13]. ACE builds on the *Open Grid Service Architecture* (OGSA) [6] and autonomically synthesizes composition plans, when possible, from available pool of services based on dynamically defined objectives and constraints. The key innovation is a dynamic composition model based on the theory of relational algebra and graph theory.

1.1 Motivation

In this section, we address the motivation and advantages of this work.

1.1.1 Why We Need Composition?

With the advent of web services standards and a service - oriented Grid architecture [11], it is foreseeable that competing as well as complimenting services will proliferate. Several independent service providers will be providing related services. Consequently ability of taking existing services and combining and recombining them to solve new problems and create new services will become important. The main

advantage will be the reduction in cost, effort and development time to create and manage new services.

1.1.2 What We Need In Grid and Autonomic Environment?

The Grid [7] is rapidly emerging as the dominant paradigm for wide area distributed computing. Its goal is to provide a service-oriented infrastructure [11] that leverages standardized protocols and services to enable pervasive access to, and coordinated sharing of geographically distributed hardware, software, and information resources. The fundamental concept underlying the emerging service oriented Grid architecture is the virtualization of entities as services and the seamless interactions and integration of these services. The *Open Grid Service Architecture* (OGSA) specification [6] defines standard interfaces and mechanisms for describing, invoking and managing Grid services [6]. In OGSA, entities on the Grid are represented as services and new higher-level services and applications can be constructed from the available services. This motivates the need for flexible and scalable composition models.

Autonomic Computing [8] aims to transform tasks that require constant human interference and awareness, to be self-managing, self-healing, self optimizing, etc. Since composition creation might not be a one time effort and composition plans need to adapt to the changes in the environment and underlying resources, automating the process of composition creation and management and making it transparent to end user motivates the need for creating new tools and supporting services.

1.1.3 Why We Need Dynamic Service Composition?

In dynamic composition model [33, 27], composition plans are created at runtime based on dynamically defined composition objectives, their semantic descriptions, constraints, and available services and resources. Dynamic composition models are naturally suited for Grid environments where new services are constantly added and

existing services are extended, modified or retired, and Grid applications where composition and interaction requirements are only known at runtime. These models can support autonomic (i.e. with minimum human support) behavior and mutable interaction patterns where all service need not be envisioned at design time and can be created on-demand bases. We believe that dynamic service composition is required to address the challenges posed by complex applications, and to satisfy the need for low cost solutions with short turn around time.

1.2 Problem Statement

In this research, we investigate the issues and challenges in enabling dynamic service composition. We present the design and architecture of *Accord Composition Engine* (ACE). The overall goal of ACE is to autonomically synthesize composition plans, when possible, from available pool of services based on dynamically defined objectives and constraints.

1.3 Contributions of the Thesis

The thesis makes these key contributions:

- ***Autonomic Composition Model***: In our model, the composition plan(s) is described by a path in graph structure and the formalism is motivated by the theory of concurrent processes and relational algebra.

- ***Using user-defined constraints to model the behavior of composition***: ACE views constraints [22] as formalization of relationships which can hold between services participating in a composition instance. The composer is responsible for abstracting out different aspects of composition [23] and specifying the appropriate environment and constraints. ACE provides means to

initialize and invoke constraints. These simple conditions are used to create relationships between services. ACE is not a constraint solver and assumes that composer will give the valid set of constraints. Validity of constraint set implies that the set shows the property of confluence, termination and observable determinism.

- ***Design and implementation of ACE***: ACE enables the construction, deployment and evaluation of autonomic service composition plans. It is a part of the framework that dynamically integrates and composes Grid services and enables autonomic applications.

- ***Service description enhancement with semantic information (keywords)***: The services are described by *Web service Description Language* (WSDL) [4]. ACE users complement the standard description with semantic and contextual metadata.

- ***Support for enabling ad hoc interactions***: The interaction links are created during the plan generation phase These ad-hoc interactions can have hierarchical or peer-to-peer relationship between them and are based on policies and constraints that are defined, deployed and executed during the composition plan generation phase.

- ***Support for finding and evaluating multiple composition plans***: Composition request might result in multiple alternative plans. ACE provides mechanisms to evaluate different plans according to cost, user defined constraints, environmental factors, etc and assist user in selecting the best plan.

1.4 Organization

The thesis is organized as follows. Chapter 1 gives the motivation of our work. It talks about the need of composition, need of applications in Grid environment and

specifically the need for dynamic service composition. Chapter 2 talks about different types of composition models and the existing languages to describe them. In this chapter we also give a brief overview of the approach used by different projects in academic and commercial domain. Chapter 3 presents project AutoMate, a framework to enable autonomic applications. Chapter 4 presents Accord Composition Model. Chapter 5 presents Accord Composition Engine. It also gives describes architecture, algorithm of operation and implementation of ACE . Chapter 6 gives conclusion and comments on the future work

Chapter 2
Prior Work

Composition models have received considerable interest in both academia and industry in recent years. In this chapter we summarize the recent efforts is this direction. Section 2.1 addresses the composition types and their characteristics. Section 2.2 specifies the languages used to describe compositions and workflows in scientific and commercial domain. Section 2.3 presents the relevant work in Grid and commercial environment, illustrating their key features and limitations.

2.1 Composition

Service composition can be simply defined as the process of taking existing services and combining them (based on definitions and constraints) to form new services. The composition model used by most existing application development frameworks assumes that the composer has a priori knowledge of the composition goals, the participating services and their interaction patterns. In this model, composer identifies relevant services, explicitly states their interactions and creates composition script. A flow engine then invokes this composition. Unfortunately this relative static composition approach is not very scalable. As the number of available services (resources, devices, applications, etc) increases exponentially, manual choreography of compositions and interactions is not realistic. Furthermore the assumption that the composer has a priori knowledge about the composition goals, the participating services and their interaction patterns is not valid in all scenarios. Thus a more dynamic composition model is required. In dynamic service composition model, composition plans

are created at runtime based on dynamically defined composition objectives, their semantic descriptions, constraints, and available services and resources. These models can support autonomic (i.e. with minimum human support) behavior and mutable interaction patterns where all services need not be envisioned at design time and can be created on-demand bases. However dynamic service compositions is extremely challenging and requires addressing a number of critical issues such as discovering and identifying relevant services, formulating and ranking (and selecting) composition plans using current context, goals, constraints and costs, and checking their validity. Interesting work on this subject has been done in recent years. Let us look at the languages for composition first.

2.2 Composition Languages

Many languages have been proposed by academic and industrial research groups for describing compositions and workflows. Industry efforts include *Web Services Flow Language*(WSFL) [9] from IBM an XML language for the description of Web Services compositions, *Business Process Modeling Language*(BPML) [21] which is a meta-language for modelling an abstracted execution model for collaborative and transactional business processes based on the concept of a transactional finite-state machine, *Web Services Choreography Interface* (WSCI) [15], XLANG [3] from Microsoft which specifies message exchange behavior among the participating web services for automation and composition of new business processes, *Jini Services flow Language* (JSFL) [9] an XML based notation for describing composite jobs made up of interacting services, *Web Service Description Language* (WSDL) [4] a standard language used to describe the syntactic aspects of services [23], *Business Process Execution Language for Web Services* (BPEL4WS) [21] combines the graph oriented process representation of WSFL and the structural construct based processes of XLANG into a unified standard for Web Service Composition and ebXML [14], a specification that

enables enterprizes to conduct business over the internet using an open XML-based infrastructure. Efforts in the Grid community include *Grid Services Flow Language* (GSFL) [32] an XML-based language that allows the specification of workflow descriptions for Grid services in the OGSA framework and HERMES [24], a specification language that can be used to express scripts for complex activities involving coordination and collaboration. In contrast to these standards, researchers are also developing unique Web service markup language called DAML-S which provides service providers with a core set of markup language constructs for describing the properties and capabilities of their services in unambiguous, computer-interpretable form.

2.3 Composition in Grid and Commercial Environment

Composition and workflow has been addressed by systems such as the Chimera Virtual Data System (GriPhyN) [17], Symphony [30], METEOR [19], COSMOS [29], Aurora [24], SWORD project [28], SELF-SERV [25] and DySCo [27]. Chimera Virtual Data System (GriPhyN) [17] considers compositions as graphs of services. Unfortunately the overall service graph is static and assumes a priori knowledge of participating services and their interaction pattern. Symphony [30] is a Java based composition and manipulation framework based on the Sun JavaBeans component architecture [12]. Its principle elements are a meta-program constructor and a back end execution environment for invocation. Symphony also supports only static compositions. METEOR [19] addresses runtime adaptability of a composed workflow. Its focus is primarily on runtime management rather than composition planning. COSMOS [29] and Aurora [24] are two similar examples of advanced architecture for e-service management. Once again, the main limitation of these systems is the rigidity in the interconnection and integration between services. SWORD [28] uses a rule-based expert system to find composition plans. In SWORD each service is represented as a logical rule that expresses the inputs and outputs associated with it.

The SWORD model only addresses interface matching informational services. SELF-SERV [25] main focus is on the peer to peer execution model for composite service execution and use of state charts to describe operations, components and services. DySCo [27] enables dynamic service composition and is based on the idea of functional incompleteness and multi-party orchestration. Semantics and characteristics of services are described using an ontology based approach. DySCo primarily address stateless e-services. Another interesting and related approach is the associative broadcast based coordination model [26], which integrates coordination with composition. The limitation is that associative naming and binding is defined at compile time to select an initial set of binding of components.

Efforts within the Grid computing community addressing composition in the context of workflows include Webflow [31], DAGMan [18], UNICORE [20] and XCAT [16]. Webflow [31] is one of the earlier workflow systems and support application compositions in Grid environments. DAGMan [18] is the meta-scheduler in Condor-G [18] and manages dependencies between jobs. XCAT Application Factories [16] address workflow related issues for Grid-based components within the Common Component Architecture (CCA) [16] framework. Different agent technologies are also developed that will use the markup exported by services to achieve end user's needs

Chapter 3
AutoMate: An Autonomic Component Framework

AutoMate [1] is a framework for enabling autonomic Grid applications. It allows the definition of autonomic components, development of autonomic applications as dynamic composition of autonomic components, and provides key enhancements to existing middleware and runtime services to support these applications on the Grid. It builds on three fundamental concepts:

- Separation of policy from mechanism distilling out the aspects of components and enabling them to orchestrate a repertoire of mechanisms for responding to the heterogeneity and dynamics, both of the applications and the Grid infrastructure. The policies that drive these mechanisms are specified separately. Examples of mechanisms are alternative numerical algorithms, domain decompositions, and communication protocols; an example of a policy is to select a latency-tolerant algorithm when network load is above certain thresholds.

- Context, constraint and aspect based composition techniques applied to applications and middleware as an alternative to the current ad-hoc processes for translating the application's dynamic requirements for functionality, performance, quality of service, into sets of components and Grid resource requirements.

- Dynamic, proactive, and reactive component management in order to optimize resource utilization and application performance in situations where computational and/or resource characteristics may change

Building on these fundamental concepts, AutoMate addresses fundamental issues and provide key solutions in the autonomic formulation, composition, and runtime management of applications on the Grid. AutoMate builds on the emerging Grid infrastructure and extends the Open Grid Service Architecture (OGSA).

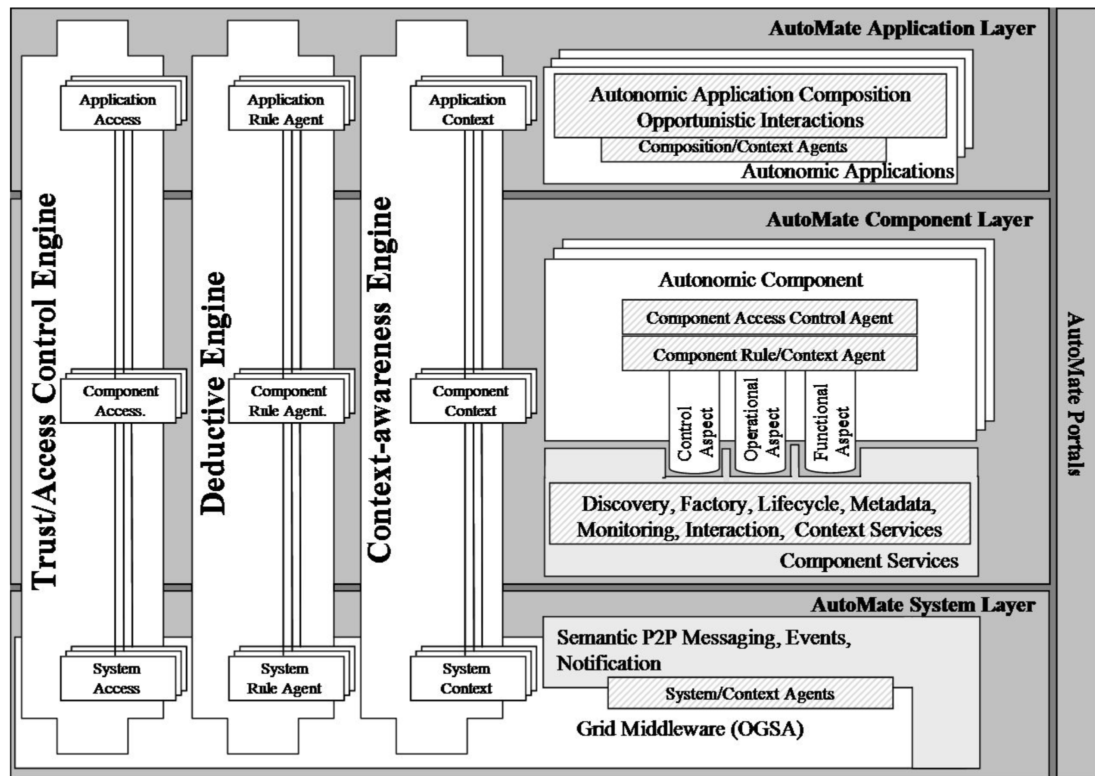


Figure 3.1: AutoMate Architecture Diagram

3.1 AutoMate Architecture

A schematic of the overall architecture is presented in Figure 3.1. AutoMate is composed of the following components:

A.) ***AutoMate System Layer***: The *AutoMate System Layer* builds on the Grid middleware and OGSA and extends core Grid services (security, information and resource management, data management) to support autonomic behavior. Furthermore, this layer provides specialized services such as peer-to-peer semantic messaging, events and notification.

B.) ***AutoMate Component Layer***: The *AutoMate Component Layer* addresses the definition, execution and runtime management of autonomic components. It consists of AutoMate components that are capable of self configuration, adaptation and optimization, and supporting services such as discovery, factory, lifecycle, context, etc. (which builds on core OGSA services).

C.) ***AutoMate Application Layer***: The *AutoMate application layer* builds on the component and system layers to support the autonomic composition and dynamic (opportunistic) interactions between components.

D.) ***AutoMate Engines***: The *AutoMate Engines* are decentralized (peer-to-peer) networks of agents in the system. The context-awareness engine is composed of context agents and services and provides context information at different levels to trigger autonomic behaviors. The deductive engine is composed of rule agents which are part of the applications, components, services and resources, and provides the collective decision making capability to enable autonomic behavior. Finally, the trust and access control engine is composed of access control agents and provides dynamic context-aware control to all interactions in the system.

In addition to these layers, AutoMate portals provide users with secure, pervasive (and collaborative) access to the different entities. Using these portals users can access resource, monitor, interact with, and steer components, compose and deploy applications, configure and deploy rules, etc.

3.2 Autonomic Components in AutoMate

Autonomic components in AutoMate export information and policies about their behavior, resource requirements, performance, interactivity and adaptability to system and application dynamics. In addition to the functional interfaces exported by traditional components, AutoMate components provide semantically enhanced profiles

or contracts that encapsulate their functional, operational, and control aspects. A conceptual overview of an AutoMate component is presented in Figure 3.2. The functional aspect specification abstracts component functionality, such as order of interpolation (linear, quadratic, etc.)

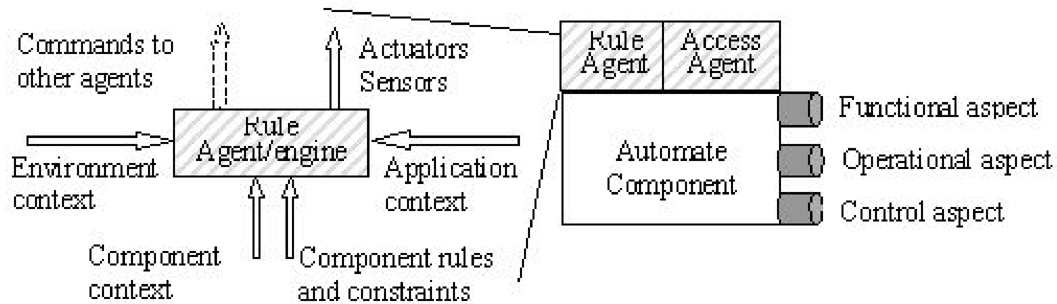


Figure 3.2: An AutoMate Component

This functional profile is then used by the compositional engine to select appropriate components based on application requirements. The operational aspect specification abstracts a component's operational behavior, including computational complexity, resource requirements, and performance (scalability). This profile is then used by the configuration and runtime engines to optimize component selection, mapping and adaptation. Finally, the control aspect describes the adaptability of the component and defines sensors/actuators and policies for management, interaction and control. AutoMate components also encapsulate access policies, rules, a rule agent, and an access agent that allow the components to consistently and securely configure, manage, adapt and optimize their execution based on rules and access policies. The access agent is a part of the AutoMate access control engine and the underling dynamic access control model, and manages access to the component based on its current context and state. The rule agent is the part of AutoMate deductive engine and manages local rule definition, evaluation and execution at the component level. Rules can be dynamically defined (and changed) in terms of the component's interfaces (based on

access policies) and system and environmental parameters. Execution of rules can change the state, context and behavior of a component, and can generate events to trigger other rule agents. The rule agent is also responsible for managing, resolving rule conflicts using rule priorities and a dynamic rule-lock mechanism. AutoMate components build on DIOS/DIOS++ [10] which provides mechanisms to directly enhance traditional computational objects/components with sensors, actuators, rules, a control network that connects and manages the distributed sensors and actuators, and enables external discovery, interrogation, monitoring and manipulation of these components at runtime, and a distributed rule-engine that enables the runtime definition and deployment for managing and adapting application components. Application components may be distributed (spanning many processors) and dynamic (be created, deleted, changed or migrated at runtime). Access to a component's sensors and actuators is governed by its local access control policies along with global application level policies. Rules can be dynamically composed using sensors and actuators exported by application components. These rules are automatically partitioned and deployed onto the appropriate components using the control network, and evaluated by the distributed deductive engine.

3.3 Autonomic Compositions in AutoMate

Applications are typically composed with well defined objectives. In case of autonomic applications, however, these objectives can dynamically change based on the state of the application and/or the system. As a result, we need to dynamically select components and compose them at runtime based on current objectives. Together, the profiles, policies, and rules allow autonomous components to consistently and securely manage and optimize their executions. Furthermore, they enable applications to be dynamically composed, configured and adapted. Dynamic application work-flows can

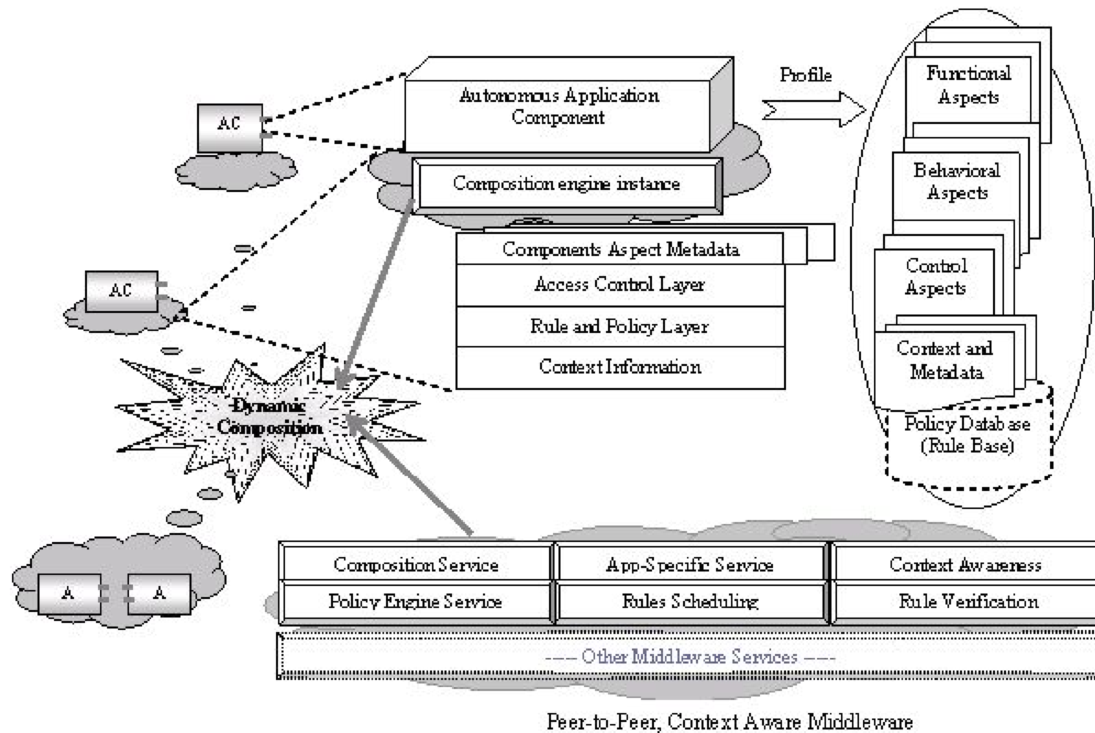


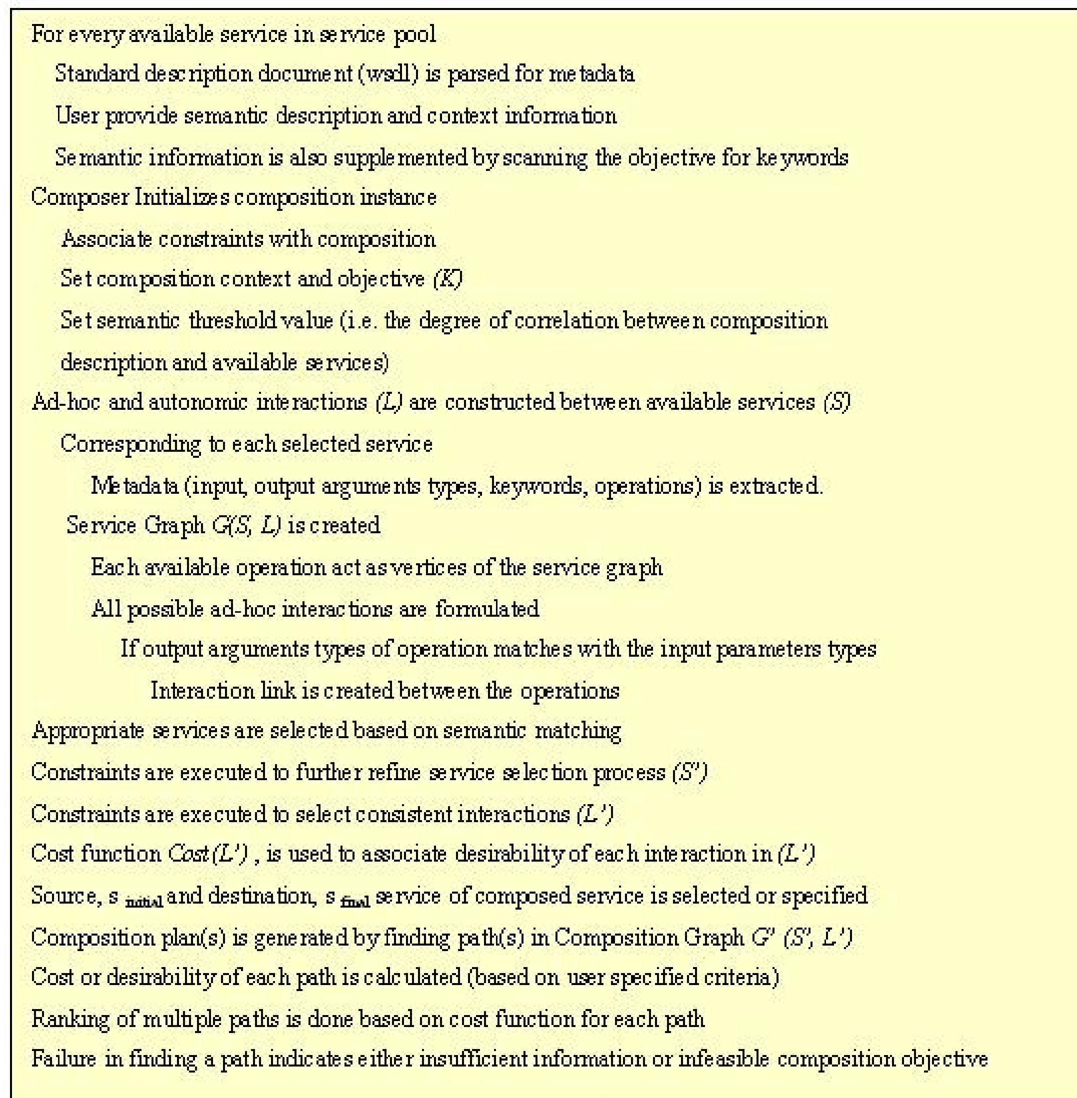
Figure 3.3: Autonomic Compositions in AutoMate

be defined to select the most appropriate components based on user/application constraints (highest performance, lowest cost, reservation, execution time upper bound, best accuracy), on the current applications requirements, to dynamically configure the component's algorithms and behavior based on available resources or system and/or applications state, and to adapt this behavior if necessary. The AutoMate dynamic composition model may be viewed as transforming a given composition or workflow into a new one by adding or modifying interactions and participating entities. Its primary goal is to enable dynamic (and opportunistic) choreography and interactions of components and services to react to the heterogeneity and dynamics of the application and underlying execution environment to produce the desired user objectives. The AutoMate dynamic composition model [33] is context aware and is based on policies and constraints that are defined, deployed and executed at runtime (see Figure 3.3). Composition policies and constraints are defined as simple rules and execute on the distributed deductive engine - i.e. there is no central authority that

manages the composition process. These rules are defined in terms of the interfaces and aspects exported by AutoMate components, the current context of the scenario and the overall objective of the application. Rules are simple and non-recursive, and can be composed and aggregated in a consistent way - based on logic and constraint based programming techniques [22]. Users can define and deploy rules at runtime provided they have the required privileges, and the rules inherit the priorities and privileges of their owners. Rules execute in a distributed fashion on a peer-to-peer deductive shell exported by the autonomic middleware as described below. Firing of rules causes the components to adapt, optimize, interact and compose. Composition metadata is defined locally at the component level or globally at the application or the middleware level using a standard representation.

Chapter 4

The Accord Autonomic Composition Model



For every available service in service pool
  Standard description document (wsdl) is parsed for metadata
  User provide semantic description and context information
  Semantic information is also supplemented by scanning the objective for keywords
Composer Initializes composition instance
  Associate constraints with composition
  Set composition context and objective *(K)*
  Set semantic threshold value (i.e. the degree of correlation between composition description and available services)
Ad-hoc and autonomic interactions *(L)* are constructed between available services *(S)*
  Corresponding to each selected service
    Metadata (input, output arguments types, keywords, operations) is extracted.
  Service Graph *G(S, L)* is created
    Each available operation act as vertices of the service graph
    All possible ad-hoc interactions are formulated
      If output arguments types of operation matches with the input parameters types
        Interaction link is created between the operations
Appropriate services are selected based on semantic matching
Constraints are executed to further refine service selection process *(S')*
Constraints are executed to select consistent interactions *(L')*
Cost function *Cost(L')* , is used to associate desirability of each interaction in *(L')*
Source, $s_{initial}$ and destination, s_{final} service of composed service is selected or specified
Composition plan(s) is generated by finding path(s) in Composition Graph *G' (S', L')*
Cost or desirability of each path is calculated (based on user specified criteria)
Ranking of multiple paths is done based on cost function for each path
Failure in finding a path indicates either insufficient information or infeasible composition objective

Figure 4.1: Accord composition algorithm

A key goal of the Grid is to provide ubiquitous resources and services availability. Furthermore, the Grid, by definition, is a dynamic and open environment where the

availability and state of these services and resources are constantly changing. The emerging Grid applications are similarly complex, dynamic and heterogeneous. As a result, the ability to compose services (and applications) on the fly based on current availability of services, current context and dynamically define objectives and goals is critical. While the existing systems listed do address many aspects of composition, they do not completely addresses the challenges of dynamic service composition. For example, the underlying composition approaches in these systems do not support dynamically defined objectives and constraints, or ad hoc definition of interactions and behaviors.

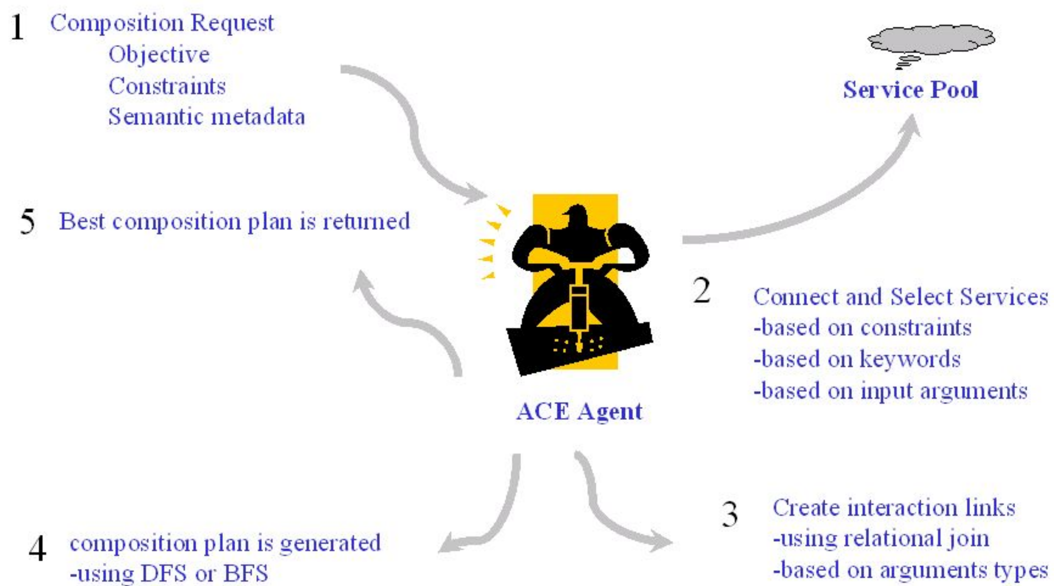


Figure 4.2: Accord algorithm in action

The primary focus of the Accord dynamic composition model presented in this thesis is to autonomically synthesize composition plans, when possible, from available pool of services based on dynamically defined objectives and constraints. Accord enhances standard (OGSA) service descriptions with semantic metadata and use this metadata along with, current context, dynamically defined composition objectives and constraints and relational algebra to choreograph ad-hoc interaction and composition plans at runtime. Alternate plans may be evaluated and ranked based on

different cost factors.

In Accord, the pool of currently available services (or current service pool) is represented as a graph where the nodes are the services in the pool and the links define the interactions and the composibility of the services. A composed service plan is then a path in this graph. The Accord dynamic composition model can be formally defined as follows:

A.) Composition is based on a service graph $G(S, L)$ where, S is a set of available services and L a set of possible interactions.

- Service set $S = \{s_i\}$ and each s_i is associated with an ordered set of keywords, $\{K(s_i)\}$.
- Interaction set $L = \{l_{i,j}\}$ such that $s_i, s_j \in S$. Each interaction $l_{i,j}$ has a cost value $Cost(l_{i,j})$ associated with it.

B.) In the service graph, $G(S, L)$, the available services are vertices and interaction are edges. The edges are created at runtime using a relational join operation, $l_{i,j} \in s_i \bowtie s_j (_{s_{i(OutputMsg.ArgTypes)} = s_{j(InputMsg.ArgType)}})$.

C.) The composer specifies composition description as initial service, $s_{initial}$, final service, s_{final}, an ordered set of keywords, $\{K_{composition}\}$ and a set of constraints, $\{c_k\}$.

D.) A subgraph of the service graph called composition graph $G^{'}(S^{'}, L^{'})$ is generated using these inputs as follows:

- $\forall\ i, s_i \in S^{'} \iff K(s_i) \subseteq \{K_{composition}\}$.
- $\forall\ i, j, l_{i,j} \in L^{'} \iff s_i \in S^{'}, s_j \in S^{'}$ and $Valid(l_{i,j}, \{c_k\}) = True$.

E.) Dynamic Service Composition can be defined as finding a path from $s_{initial}$ to s_{final} in $G^{'}(S^{'}, L^{'})$.

Service Name	Input Argument	Arguments Type	Output Arguments	Output	Keywords
Driving Direction (DDS)	SrcAddr, TgtAddr	String, String	Driving Direction	String	Driving Direction, MapQuest
Location Service (LS)	Location	String	Address	String	Address, Landscape
Location Service (LS)	firstname, lastname, city	String, String, String	Address	String	Address, Name, City
Vehicle Dependent Driving Service	SrcAddr, TgtAddr, Vehicle	String, String, String	Driving Direction	String	Vehicle, Driving Direction, Yahoo

Table 4.1: Service Pool for the example travel guide service.

The complexity of the plan generation algorithm is $O(S^{'}+L^{'})$. Note that the model defined about assumes that the composer will provide a proper set of constraints, and the set of constraints will satisfy properties of confluence, termination and observable determinism.

The Accord composition algorithm is presented in Figure 4.1. In the initialization and service selection step, the services in the current service pool are parsed to generate service set S. Then a relational join operation is used to construct the set of ad-hoc interactions, L, and the service graph G is created. The composer specifies a composition request as a set of constraints (C), keyword metadata ($\{K_{composition}\}$), input service ($s_{initial}$) and output services (s_{final}). The keyword set and constraint set are used to select the participating services, $S^{'}$, generate the set of associated interactions $L^{'}$, and the composition graph $G^{'}$. Cost associated with each $l^{'}_{i,j}$ is calculated. Candidate composition plans can now be generated as paths in $G^{'}$ between $s_{initial}$ and s_{final} using graph path algorithms(DFS,BFS). The composition plans can be ranked based on costs. These costs could reflect economic, operational environments and/or user defined factors. Constraints can belong to different categories and can control aspects of both services and compositions. Examples of constraint categories include security constraints, behavioral constraints and integrity constraints.

To illustrate the operation of the Accord composition model consider a scenario

Input	Source Service	Response Type	Target Service	Output	Cost
SrcAddr, TgtAddr	DDS	String	-	String	0
Landscape, TgtAddr	LS (landscape), TgtAddr	String, String	DDS	String	1
Landscape, Landscape	LS (landscape), LS (landscape)	String, String	DDS	String	1
Landscape, TgtAddr, Vehicle	LS (landscape), TgtAddr, Vehicle	String, String, String	VDDS	String	1
Landscape, Landscape, Vehicle	LS (landscape), LS (landscape), Vehicle	String, String, String	VDDS	String	1

Table 4.2: Interaction table for the example travel guide service.

where a user is looking for a travel guide service which gives the travel route between two locations. The set of available services includes a *Driving Directions Service (DDS)* that simply returns driving directions between two specified addresses, a *Vehicle-dependent Driving Direction Service (VDDS)* that returns directions as a function of the specified vehicle (e.g. car, train, boat, bicycle), and a *Location Service (LS)* that returns the exact address given an approximate location. The service pool and interaction table for this example is shown in Table 4.1and Table 4.2. In this scenario, if a service has exact endpoint addresses service DDS is directly invoked, if one or both of the endpoints are not exact, the composition of LS and DDS is required, and if vehicle information is included, then VDDS will be invoked instead of DDS. The decision to include or exclude any service is based on the service request and constraints. Since the participating services are not known in advance, interactions between them cannot be modelled a priori and must be generated at runtime based on the request and the set of available services.

Chapter 5
Accord Composition Engine

In this section we present the design and operation of Accord Composition Engine. The overall goal of ACE is to autonomically synthesize composition plans, when possible, from available pool of services based on dynamically defined objectives and constraints. The key innovation is a dynamic composition model based on relational algebra and graph theory. Services are described using standard Web Service Description Language (WSDL) and extended with semantic metadata (keywords). Relational joins are then used to generate composition plans and choreograph ad-hoc interactions at runtime to satisfy the composer's objectives and constraints. Alternate plans may be evaluated and ranked based on different factors. The key advantage of this approach is that services need not be envisioned at design time and can be created on on-demand bases.

5.1 Architecture Overview

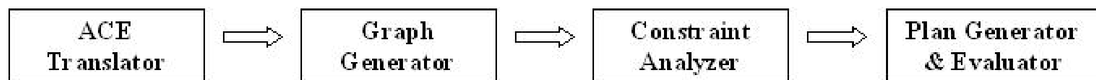


Figure 5.1: Architectural overview of the Accord Composition Engine (ACE).

An architectural overview of the *Accord Composition Engine* (ACE) is presented in Figure 5.1. ACE can be a part of composition services available on the Grid or composition agents within the Grid middleware. It builds on OGSA and the emerging Grid middleware. A service in ACE corresponds to a Grid service as specified in the

Grid Service Specification and is described using WSDL. The description field is used to add semantic information in the form of keywords describing the service. A service pool is the set of services that are available to a composer. The current service pool is defined by a Node Table, Message Table and Service Table which are constructed dynamically using existing OGSA discovery mechanisms such as SQUID, MDS or UDDI. The ACE architecture consists of four key models: ACE translator, Graph Generator, Constraint Analyzer, Plan Generator and Evaluator. These modules are described below.

5.1.1 ACE Translator

The ACE translator module parses the WSDL service description for each service in the current service pool and uses this information to update the relevant tables. It creates a row in the Node Table corresponding to each "operation" in the description, which contains the service name, operation name, ordered sequence of input parameters, input message name and output message name. For each message name, a separate entry is created in the Message Table with the message name as primary key. Each message entry also contains argument names and argument type attributes. The schema for these tables is presented in Figure 5.2.

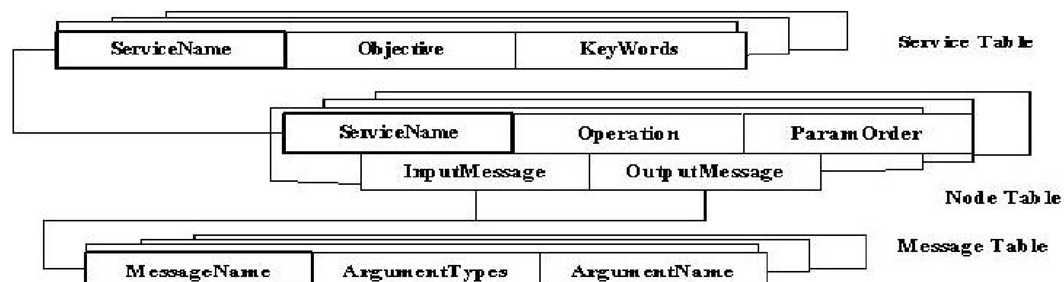


Figure 5.2: Schemas for ACE service and message description tables.

5.1.2 ACE Graph Generator

The ACE Graph Generator module is responsible for defining the interaction links between services in the service pool using relational joins. This is done based on the message description in the Message Table. If the arguments and attribute types associated with the output message of a source operation is a superset of the arguments associated with the input message of a target operation, then a directed edge exists from source operation to target operation. Corresponding to each such link, an entry is created in the Link Table. The attributes of Link Table are the source operation name, source service name, source message name, destination operation name, destination service name, destination message name, cost of the link (defined by the context), level of composition (in case where composition span across multiple service pools), and a valid flag that is true if the current link is active.

5.1.3 ACE Constraint Satisfaction Module

The Constraint Satisfaction Module is responsible for evaluating and executing the constraints associated with individual services and service composition requests. In ACE, constraints are represented by simple SQL expressions that modify the validity of interaction links. Thus the ACE constraint satisfaction module operates on Link Table and enables or disables link entries in the table. The schema for the Link Table is shown in Figure 5.3.

SourceOperation	SourceService	SourceMessageName
TargetOperation	TargetService	TargetMessageName
CostOfLink	Valid	Level

Figure 5.3: Schema for ACE link description table.

5.1.4 Dynamic Service Plan Generator and Evaluator

The dynamic service plan generator and evaluator module is responsible for generating composition plans in response to a composition request. It works in conjunction with Constraint Satisfier Module and operates on the Link Table. A plan is an ordered set of services and their interactions that can satisfy the request. Service and link costs are used to rank plans when multiple plans exist.

5.2 Operation

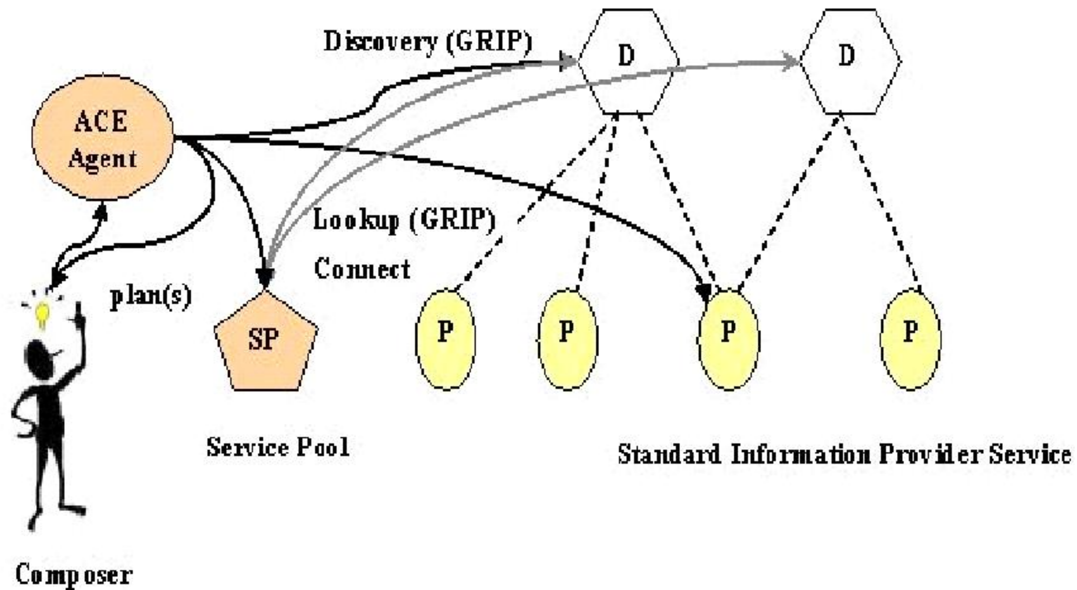


Figure 5.4: Operation of the Accord Composition Engine.

The end to end operation of ACE is illustrated in Figure 5.4. For example let us reconsider our travel guide composite service. The dynamic service composition is triggered when a composer makes a request to the ACE agent. Composer can be a user or a service. A request will contain the semantic description of the composition along with the input parameters and constraints. Say a request is made to find the driving directions between source and destination address. The composer likes Yahoo Maps and is interested in finding the shortest route. ACE agent on getting a composition request will connect to the service pool (refer Table 4.1). Services present

in the service pool can belong to different directories. We assume that providers have already registered their services using standard Grid registration protocols such as GRRP. ACE agent will first select the services based on keywords and request parameters. For example "Yahoo" and "Shortest route" will result in the selection of the VDDS service (refer Table 4.1). The location service will only be selected if either source or destination addresses are not properly defined. Finally composition plan(s) is generated by the agent and returned to the composer. In cases where multiple plans are possible, the cost is calculated for each plan and appropriate plan is returned. The composition request fails if (1) a plan does not exist, (2) the composition request is insufficient, or (3) the constraints are invalid. The first case occurs when the composite service can be expressed using ACE plan but no sequence of services exists that can satisfy the request. This situation can be handled by increasing the number of available services in service pool and lowering the semantic correlation threshold used to select services. In the second case, the composer can be asked for additional specifications for the composition. For the final case, ACE assumes that the constraints specified by the composer are valid, i.e. they exhibit the property of confluence (same effect irrespective of constraints execution order), observable determinism (actions are same) and termination (cascaded constraints execution not allowed). If the specified constraint set does not satisfy these properties, ACE will fail to generate a valid plan.

5.3 Prototype in Action

In this section, we will walk through our travel guide composite service example to illustrate the working of ACE algorithm and demonstrate prototype in action. In our model a service is choreographed on the bases of service request that specifies the desired functionality. The plan of composition will consist of set of participating services and requisite interactions between them. Suppose that composer wish to

create a service that looks up the driving direction between two addresses. The step by step guide to the service composition process is illustrated below.

```
//Service returns driving direction between two addresses
public interface IDrivingDirectionService {
   String drivingDirection(String sourceAddress, String targetAddress);
}

// Service returns driving direction as a function of specified vehicle
public interface IVehicleDependentDrivingService {
   String drivingDirection(String sourceAddress, String targetAddress, String vehicle);
}

// Service returns exact address given the approximate location
// People lookup service, returns address of person given his name and city
public interface ILocationService {
   String getAddress(String relatedLocation);
   String getAddress(String firstname, String lastname, String city);
}
```

Figure 5.5: Interface of services in Service Pool.

- ***Step 1***:ACE agent will contact the service pool (ref Figure 5.4). Services present in the service pool can belong to different directories. We assume that different service providers have already registered their service interfaces with the directories using standard Grid registration protocols such as Grid Registration Protocol (GRRP) [5]. Since a provider can export their service interfaces in any native programming languages, a uniform standard (WSDL or GSDL) is required to describe services. The service interfaces for our example are given in figure 5.5. We are using Glue tool "java2wsdl" to get the standard service description from java class files. The sample standard descriptions (wsdl files) of our interfaces are given in appendix (A.1 , A.2, A.3). The ACE agent is responsible for transforming and extracting the metadata from standard service interfaces and storing it in a relational (tabular) form. The resulting format for our travel guide example is presented in Table 4.1.

- ***Step 2***:The ACE translator will parse the wsdl service description and will transform it into a new schema (ref figure 5.2) and annotate it with semantic metadata by parsing the service description. Additional semantic metadata can also be associated by the user to further refine search and support advance querying in later stages. Sample snapshot for our example is given in Table 4.1

Scenario	Service Request	Invocation parameters	Description
A	**Name-to-Driving-Direction-Service**	**[First name, Last name, City], [First name, Last name, City]**	**Looks up driving directions between two persons homes given their name and cities**
B	**Vehicle-Dependent-Direction-Service**	**Landscape, Landscape, Vehicle**	**Gives directions between two addresses as a function of available vehicle**
C	**Driving-Direction-Service**	**Landscape, Landscape, Keywords**	**Returns driving directions between locations given constraints such as shortest path, avoiding highways, etc**

Table 5.1: Sample Composition Scenarios

- ***Step 3***:The composer (user or service) makes a request to the ACE agent. Suppose the composer wants to create a "Name to Driving Direction Service", where user provides the name and city of two individuals and service is expected to return driving directions between their homes. In another variation of the same service, user also specifies the vehicle as a part of service request. In yet another scenario, user might have certain constraints such as he does not know the name of the individual but knows the approximate location of his house or he likes Yahoo Maps service and is interested in finding the shortest route avoiding all highways. Some possible composition scenarios for our example are given in Table 5.1.

- ***Step 4***: Dynamic service composition is triggered when a composer makes a request of service to the ACE agent. The ACE agent on getting the composition request will select the appropriate services and create interaction links between them based on request parameters, their specified order and semantic information (keywords). The selected services for different service composition

Scenario	Service Request	Services Selected
A	Name-to-Driving-Direction-Service	Location Service, Location Service, Driving Direction Service
B	Vehicle-Dependent-Direction-Service	Location Service, Location Service, Vehicle Dependent Driving Service
C	Driving-Direction-Service	Location Service, Location Service, Driving Direction Service

Table 5.2: Participating services for composition scenarios

scenarios are described in table 5.2.

Source Operation	Target Operation	Comment
Location Service (Landscape)	Driving Direction Service	Location Service, provides address to Driving Direction Service
Location Service (Landscape)	Vehicle Dependent Direction Service	Location Service provides address to Vehicle Direction Service
Location Service (Name, Name, City)	Vehicle Dependent Direction Service	Name-Location Service provides address to Vehicle Direction Service
Location Service (Name, Name, City)	Driving Direction Service	Name-Location Service provides address to Driving Direction Service

Table 5.3: Interaction Links created by ACE Graph Generator

- ***Step 5***:Then a relational join operation is used to construct the set of ad-hoc interactions and the service graph is created. In our sample example (ref Figure 4.1), twelve ad-hoc interactions are created (since self loops are not allowed). The valid interactions created by ACE Graph Generator and ACE Constraint Satisfaction Modules are given in table 5.3. Based on composition request as a set of constraints (C), keyword metadata ($\{K_{composition}\}$), input parameters and their order, start service($s_{initial}$) and sink services (s_{final}) are selected. Cost of each interaction is evaluated. The candidate composition plans can now be generated as paths. Some simple scenarios for our composition requests (5.1) are illustrated in figure 5.6

Possible Sink Services or Operations	**Driving Direction Service, Vehicle Dependent Driving Service**
Possible Source Services or Operations	**Location Service(Landscape), Location Service (First name, Last name, City)**

Table 5.4: Source and Sink Services for Travel Guide Example

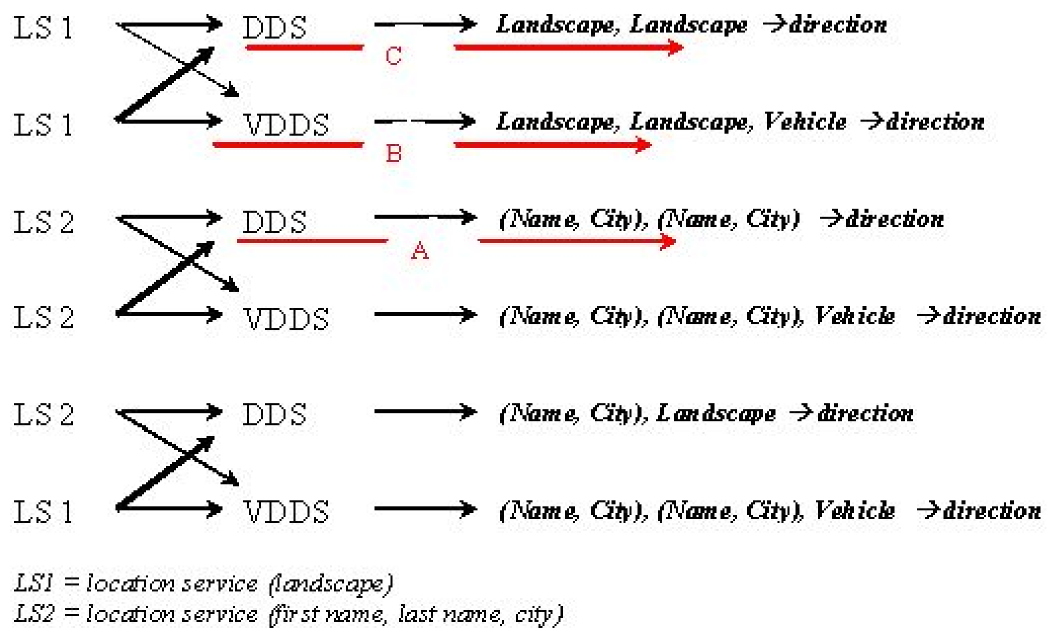


Figure 5.6: Composition Graph Instances.

Chapter 6
Conclusion and Future Work

6.1 Discussion - Advantages and Limitations

In Grid environment, composite service creation is not necessarily a one-time effort. Composition may need to adapt to the changes in the environment and underlying resources. Moreover as the services become more ubiquitous, it is not possible to consider all the permutations manually. Thus involving end users in service composition is unacceptable, creating a need for systems such as ACE that enable the construction of autonomic service composition plans. ACE also provides the mechanism to rank different plans and select the most appropriate one. An additional advantage of generating multiple plans is redundancy and fault tolerance. If one plan fails, an alternate plan can be invoked, or multiple plans can be used simultaneously for reliability or QoS. We found that dynamic service composition is extremely challenging and requires addressing a number of critical issues such as guaranteed correctness, scalability, performance analysis, and constraints analysis. In traditional service environments, response time depends primarily on resource latencies and network loads. With dynamic service composition, planning time can become an additional overhead. As a result composition planning mechanisms must be very efficient. Another important challenge is in ensuring guaranteed correctness. In many cases, it may not be possible to find any guaranteed correct plan for a composition request. ACE specifically provides no such guarantee and is based on the notion that "uncertain plan" is better than no plan. In static composition, the process is bound with the service at design time and designer can evaluate the performance metrics associated

with it. However, in dynamic composition the binding is not possible until the plans are found and invoked. In ACE, the ranking of different plans is done based on costs rather than performance data. Other challenges that need to be addressed include missing or no inputs and outputs, multiple service responses or multiple responses types, etc.

6.2 Conclusion

This thesis addressed issues and challenges in enabling dynamic service composition on the Grid and presented design and prototype implementation of the Accord Composition Engine (ACE). The ACE composition model enables autonomic generation of composition plans, when possible, from available pool of services based on dynamically defined objectives and constraints. It enhances the standard (OGSA) service descriptions with semantic metadata and uses this metadata along with the current context, dynamically defined composition objectives and constraints and relational algebra to choreograph ad-hoc interactions and composition plans at runtime. Alternate plans may be evaluated and ranked based on different cost factors. The key advantage of this approach is that services need not be envisioned at design time and can be created on on-demand bases.

6.3 Future Work

ACE addresses pre invocation planning activities of dynamic service composition. However, there are still several issues that need to be addresses such as:

- **Security**: The composition description can be associated with some form of authentication mechanism (i.e. digital signature). The motivation is to establish a trusted association between service description and implementation and between participating services.

- **Ontology-based description**: The semantic and characteristics of the composition (i.e. cost, availability, response time, available options) are currently described by simple keywords. Different service attributes need to be described with an ontology based approach.

- **Autonomic Middleware Services**: The design, development and deployment of key services on top of Grid Middleware infrastructure to support autonomic applications and dynamic compositions. The composition engine should be able to use the context information provided by the infrastructure layer to analyze, execute, plan and monitor the composition components.

- **Performance Evaluation**: For static compositions, as services are bound at design time, and designer can search for services that have metrics (such as cost, time, space, availability, etc) satisfying the quality of service (QoS) requirement of the problem being solved. Unfortunately in dynamic composed service, efficiency of the process cannot be determined until the service is invoked. Since the performance of a single participating service has the potential to affect the performance of entire composition, it is imperative to evaluate plans and services beforehand. Thus advance performance evaluation methods, models and tools are required.

Appendix A

Sample service interfaces of services

```
<definitions name='LocationService'
      -------
<message name='getAddress0SoapIn'>    <part name='relatedLocation' type='xsd:string'/> </message>
<message name='getAddress0SoapOut'> <part name='Result' type='xsd:string'/> </message>
 <message name='getAddress1SoapIn'>
       <part name='firstname' type='xsd:string'/>
       <part name='lastname' type='xsd:string'/>
       <part name='city' type='xsd:string'/>
 </message>
 <message name='getAddress1SoapOut'> <part name='Result' type='xsd:string'/> </message>

<portType name='LocationServiceSoap'>
   <operation name='getAddress' parameterOrder='relatedLocation'>
        <input name='getAddress0SoapIn' message='tns:getAddress0SoapIn'/>
        <output name='getAddress0SoapOut' message='tns:getAddress0SoapOut'/>
   </operation>
   <operation name='getAddress' parameterOrder='firstname lastname city'>
       <input name='getAddress1SoapIn' message='tns:getAddress1SoapIn'/>
       <output name='getAddress1SoapOut' message='tns:getAddress1SoapOut'/>
   </operation>
 </portType>

<binding name='LocationServiceSoap' type='tns:LocationServiceSoap'>
   <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http'/>
       ------
</binding>
</definitions>
```

Figure A.1: WSDL Interface of Location Service.

```
<definitions name='VehicleDependentDrivingService'
  ------
  <message name='drivingDirection0SoapIn'>
   <part name='targetAddress' type='xsd:string'/>
   <part name='sourceAddress' type='xsd:string'/>
   <part name='vehicle' type='xsd:string'/>
  </message>
  <message name='drivingDirection0SoapOut'>
   <part name='Result' type='xsd:string'/>
  </message>

 <portType name='VehicleDependentDrivingServiceSoap'>
   <operation name='drivingDirection' parameterOrder='targetAddress sourceAddress vehicle'>
       <input name='drivingDirection0SoapIn' message='tns:drivingDirection0SoapIn'/>
       <output name='drivingDirection0SoapOut' message='tns:drivingDirection0SoapOut'/>
   </operation>
 </portType>

 <binding name='VehicleDependentDrivingServiceSoap' type='tns:VehicleDependentDrivingServiceSoap'>
   <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http'/>
 </binding>
</definitions>
```

Figure A.2: WSDL Interface of Vehicle Dependent Driving Service.

```
<definitions name='DrivingDirectionService' >
       ------
  <message name='drivingDirection0SoapIn'>
   <part name='sourceAddress' type='xsd:string'/>
   <part name='targetAddress' type='xsd:string'/>
  </message>
  <message name='drivingDirection0SoapOut'>
   <part name='Result' type='xsd:string'/>
  </message>

  <portType name='DrivingDirectionServiceSoap'>
   <operation name='drivingDirection' parameterOrder='sourceAddress targetAddress'>
    <input name='drivingDirection0SoapIn' message='tns:drivingDirection0SoapIn'/>
    <output name='drivingDirection0SoapOut' message='tns:drivingDirection0SoapOut'/>
   </operation>
  </portType>

  <binding name='DrivingDirectionServiceSoap' type='tns:DrivingDirectionServiceSoap'>
       --------
  </binding>
</definitions>
```

Figure A.3: WSDL Interface of Driving Direction Service.

References

[1] M. Agarwal, V. Bhat, Z. Li, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, M. Parashar, B. Khargharia, and S. Hariri. AutoMate: Enabling Autonomic Applications on the Grid. In *Proc. of Autonomic Computing Workshop, 5th Annual International Workshop on Active Middleware Services(AMS 2003)*, Seattle, WA, IEEE Computer Society Press, pp 48-57, June 2003.

[2] T. Bellwood. UDDI (Universal Description Discovery and Integration) Version 2.04 API Specification. http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm, July 19, 2002.

[3] Satish Thatte XLANG: Web Services for Business Process Design (XLANG). http://www.gotdotnet.com/team/xml_wsspecs/xlang-c, December, 2001.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 15, 2001.

[5] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proc. of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10) IEEE Press*, pages 181–194, San Francisco, CA, August 7-9 2001.

[6] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid:An Open Grid Services Architecture for Distributed Systems Integration. In *Proc. of the Open Grid Service Infrastructure WG, Global Grid Forum*, June 22 2002.

[7] I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure.* Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[8] P. Horn. Autonomic Computing:IBM's perspective on the State of Information Technology. http://www.research.ibm.com/autonomic/, Oct 2001. IBM Corp.

[9] F. Leymann. Web Services Flow Language (WSFL) 1.0. http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf/, May,2001. IBM Academy of Technology, IBM Software Group.

[10] R. Muralidhar and M. Parashar. An Interactive Object Infrastructure for Computational Steering of Distributed Simulations. In *Proc. of the Ninth International Symposium on High Performance Distributed Computing (HPDC 2000), IEEE Computer Society Press*, pages 304–305, Pittsburgh, PA, August 2000.

[11] M. Champion, C. Ferris, E. Newcomer, and E. Newcomer. Web Services Architecture. http://www.w3.org/TR/ws-arch/, November 14, 2002.

[12] Sun Microsystems, Inc. The JavaBeans Component Architecture. July, 2002.

[13] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham and C. Kesselman. Grid service specification. February, 2002.

[14] ebXML Team. ebXML Requirements Specification, Version 1.06. http://www.ebxml.org/specs/ebREQ.pdf, May 8, 2001.

[15] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic and S. Zimek. Web Service Choreography Interface (WSCI) 1.0. http://www.w3.org/TR/wsci/, August 2002.

[16] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon and R. Bramley. XCAT 2.0: A Component-Based Programming Model for Grid Web Services. Dept. of C.S., Indiana Univ, Technical Report-TR562, June 2002.

[17] I. T. Foster, J. Vckler, M. Wilde and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *SSDBM 2002*, pages 37-46, CA, August 2001.

[18] J. Frey, T. Tannenbaum, I. Foster, M. Livny and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proc. of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, pages 7-9, CA, August 2000.

[19] J. Miller, D. Palaniswami, A. Sheth, K. Kochut and H. Singh. WebWork: METEOR's Wen-based Workflow Management System. In *Journal of Intellegent Information Systems*, Vol 10, number 2, pages 185-215, 1998.

[20] M. Romberg. The UNICORE Grid Infrastructure. Scientific Programming, Special Issue on Grid Computing, Vol 10, Number 2, pages 149-157, CA, 2002.

[21] Intalio and BPMI.org. Business Process Modeling Language. http://www.bpmi.org/bpmi-downloads/BPML-SPEC-1.0.zip, Retrieved June 24th, 2002.

[22] K. Marriott and P.J. Stuckey. Programming with Constraints: an Introduction. MIT Press, 1999.

[23] A. Popovici, T. Gross and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. Proc. 1st Intl Conf. Aspect-Oriented Software Development, April 2002.

[24] M. Marazakis, D. Papadakis and C. Nikolaou. Aurora: An Architecture for Dynamic and Adaptive Work Sessions in Open Environments. In *Proc. of the International Conference on Database and Expert System a Applications (DEXA'98)*, Springer-Verlag LNCS Series, pages 7-9, CA, August 1998.

[25] B. Benatallah, B. Medjahed, A. Bouguettaya, A. Elmagarmid, and J. Beard. Composing and maintaining web based virtual enterprises. In *Proc. of the 1st VLDB workshop on Technologies for E-Services*, Cairo, Egypt, September 2000.

[26] B. Bayerdorffer. Distributed Programming with Associative Broadcast. In *Proc. of the 27th Annual Hawaii International Conference on System Sciences* ,Volume 2: Software Technology (HICSS94-2), Wailea, HW, USA, pp.353-362, 1994.

[27] G. Piccinelli and L. Mokrushin. Dynamic e-service composition in DySCo. In *Proc. of 21st International Conference on Distributed Computing Systems Workshops (ICDCSW '01)*, Mesa, Arizona, April 2001.

[28] S. R. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. In *11th World Wide Web Conference (Web Engineering Track)*, Honolulu, Hawaii, May 2002.

[29] F. Griffel, M. Boger, H. Weinreich, W. Lamersdorf and M. Merz. Electronic Contracting with COSMOS - How to Establish, Negotiate and Execute Electronic Contracts on the Internet. In *2nd Int. Enterprise Distributed Object Computing Workshop (EDOC '98)*, April 1998.

[30] M. Lorch and D. Kafura. Symphony : A Java-based Composition and Manipulation Framework for Computational Grids. In *Proc. of 2nd IEEE/ACM Int. Symp. on Cluster Computing and the Grid*, pages 136-143, Berlin, Germany, 2002.

[31] D. Bhatia, V. Burzevski, M. Camuseva, G. Fox, W. Furmanski and G. Prem-Chandran. WebFlow : A Visual Programming Paradigm for Web/Java Based Coarse Grain Distributed Computing. In *Proc. of 21st International Conference on Distributed Computing Systems Workshops (ICDCSW '01)*, Concurrency: Practice and Experience, Vol 9, number 6, pages 555-577, 1997.

[32] Patrick Wagstrom, Sriram Krishnan and Gregor von Laszewski. GSFL: A Workflow Framework for Grid Services. In *Supercomputing Conference (SC 2002)*, Concurrency: Practice and Experience, Vol 9, number 6, pages 11-16, November 2002.

[33] Manish Agarwal and Manish Parashar. Enabling Autonomic Compositions in Grid Environments. Submitted in *the Proc. of the 4th International Workshop on Grid Computing (Grid 2003)*, Phoenix, Arizona November 2003.