I/O-Automata based formal approach to Web Services Choreography

by

Saayan Mitra

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Ratnesh Kumar, Major Professor
Samik Basu
Manimaran Govindarasu
Robyn Lutz
Diane Rover

Iowa State University

Ames, Iowa

2009

DEDICATION

To my grandparents, who gave me so much and asked for so little.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I acknowledge the immense support I have received from my advisors Prof. Ratnesh Kumar and Prof. Samik Basu, whose patience, guidance has been with me in every step. It has been a pleasurable learning experience throughout. I am grateful to my Program of Study committee members Prof. Manimaran Govindarasu, Prof. Robyn Lutz, Prof. Diane Rover for their valuable suggestions and insights in this research as well as for the courses they taught. My teachers at Iowa State University, especially Prof. Thomas Daniels, Prof. Yong Guan, Prof. Doug Jacobson, Prof. Suraj Kothari, Prof. Leslie Miller, Prof. Giora Slutzki, Prof. Srikanta Tirthapura among others imparted value to the educational experience, without which I would not have been able to come this far.

I am indebted to my parents Subha Mitra and Anil Kumar Mitra for laying the foundations of all my knowledge, and my sister Sunayana Mitra for instilling creativity in me through her unique way of looking at the World. Their encouragement and the fruitful discussions I had with them had been invaluable. My interactions with my grandparents, Late Parul De, Late Pratap Chandra De, Late Tarulata Mitra and Late Pasupati Mitra during my formative years and later their stories shared by my parents had influenced me deep within. Much of who I am is because of Ms Sadhana Sharma who looked after me dearly during my childhood. My cousins Ms. Kamalika Mitra, Ms. Sreemoyee Chatterjee, Mr. Sayak Datta, Mr. Anirban Mitra, Ms. Sreyashi Dastidar, Dr. Kajari Biswas, Mr. Samidip Sarkar and Dr. Soma Sen and their parents had all inspired me with their wonderful accomplishments and affection. Dr. Saikat Guha had been a leading light with his splendid endeavors.

My teachers at Jadavpur University, Jodhpur Park Boys School and Nava Nalanda where I had most of my schooling had been instrumental in sowing the seeds of love for knowledge. I would especially thank Dr. Mohit Kumar Roy for supervising my undergraduate engineering project and at Nava Nalanda, Ms Jayasree Banerjee, Mr Sibaprasad Banerjee, Ms Tapati Basu, Ms Dipannita Bhosle,

Ms Ruchira Chakroborty, Ms Enakshi Chowdhury, Ms Aditi Datta, Ms Kalyani Das, Ms Nandini Dasgupta, Late Nirmalendu Gautam, Mr Jibon Kundu, Late Arya Mitra, Ms Bharati Mitra, Mr Uttam Mitra, Ms. Sipra Raha, Ms Sita Roy, Mr Subhrendu Sanyal, Ms Lila Sengupta, Ms Meera Tripathi, for the quality education and love they bestowed on me.

During my internship, I had the good fortune to work with Dr. Vishy Swaminathan, whose thorough approach towards solving problems was a positive influence on me. I also owe much to my erstwhile colleagues Mr. Anirban Chakroborty, Mr. Tapas Chakroborty, Mr. Nisith Dash, Mr. Saptarshi Das who taught me the nuances of programming in real world scenarios.

My friends Niraj Agarwal, Apala Guha had been extremely helpful and supportive bearing with all my questions when I was preparing to pursue higher education. I will always cherish the wonderful friendships I made along the way - Hemant Bhanot, Neevan Ramalingam, Naveen Kumar, Dinesh Kalyana, Chetan Hazaree, Supriyo Das, Shamik Dasgupta, Songyan Xu, Qin Wen, Jing Huang, Licheng Jin, Amine Kamel among many others. They made the journey enjoyable.

ABSTRACT

Web Services are heterogeneously developed software components invoked over the network viz. the Internet. Their main objective is to provide desired outputs in exchange of specified inputs. In the setting of service oriented architecture, Web services play a vital role by allowing computations to be carried out in a distributed fashion via communication between services over the network. This is commonly referred to as *Web service composition*. Service composition amounts to investigating whether (and how) various services can be utilized in tandem to develop new services desired by a client.

A wide range of problems needs to be addressed before service composition can be deployed in practice. These problems range from developing standard language representation for composite services to resolving semantic/vocabulary mismatch between services participating in a composition. In this dissertation we study the problem of synthesis of a mediator/choreographer in Web service composition for a given set of services and a goal. Services and goal are represented using i/o automata. The central theme of our technique relies on generating i/o automata representation of all possible choreographed behaviors of existing services (captured in form of universal service automaton, a concept introduced) and verifying that the goal can be *simulated* by the universal set of choreographed behaviors.

Such a technique is subject to state-space explosion. In light of this, we have developed a tabled-logic programming technique which generates and explores compositions in a goal-directed fashion to prove/disprove the existence of choreographer and to infer whether the desired functionality is realizable. We present a prototype implementation and show the practical applicability of our technique using composition problems with the corresponding computational savings in terms of number of states and transitions explored.

However, such a centralized choreography mechanism can involve communication/computation overhead that can be reduced through its decentralized realization. With this as motivation, we next study the problem of synthesizing a decentralized choreography strategy that will have an optimum overhead for service composition by developing a set of site-specific choreographers working concurrently to implement a desired goal service. Each communication/computation is quantified by a cost. We develop algorithms that takes as input the existing services, the goal service, the costs and produces as an output a set of site-specific choreographers that optimally realize the goal service using the existing services. The optimization would be different in cases of the goal automaton without loops (workflow) or with loops (certain operations can be repeated any number of times)

The contribution of this work lies in the automata-theoretic formal approach to the formulation and the systematic solution of the choreographer synthesis problem as well as formulation of the optimal decentralized choreographer synthesis problem and its solution. The contributions include a methodology for computing cost of automata (with or without cycles), given cost of its transitions, and a generalized solution of the optimized decentralization service composition problem.

CHAPTER 1. Web Services Overview: Architecture and Technologies

1.1 Introduction

Service-Oriented Architecture (SOA) is an application architecture in which all functions, or services, are defined using a description language and has interfaces that can be invoked to perform business processes. Each interaction is independent of each and every other interaction and the interconnect protocols of the communicating devices (i.e., the infrastructure components that determine the communication system do not affect the interfaces). A service in SOA must adhere to the two principles. The interface for interacting with the external world should be platform independent. Secondly, the service should be available for invocation by other services. In this fast growing market of software reuse, Web Services are a rapidly evolving technology having many facets. They are the torch bearer of SOA. Web Services use the three pronged principle of find, bind and publish as described in Figure 1.1. First a service is published, and clients can search or find the service they want and then dynamically or statically bind to the service. Next, we we introduce the infrastructure required for web services starting with the various standards.

1.2 Web Service Standards

There are a lot of standards, some still emerging for different aspects such as security, reliability, management. Here we introduce the core of them driving traditional web services. As can be seen in Figure 1.1(b), at the transport layer established protocols can be used for transferring Web Services payload. Typically, HTTP(S) is used for internet based Web Services communication. For publishing purposes, UDDI detailed in the next section is used. Here, Web Services are registered using the following services and protocols.

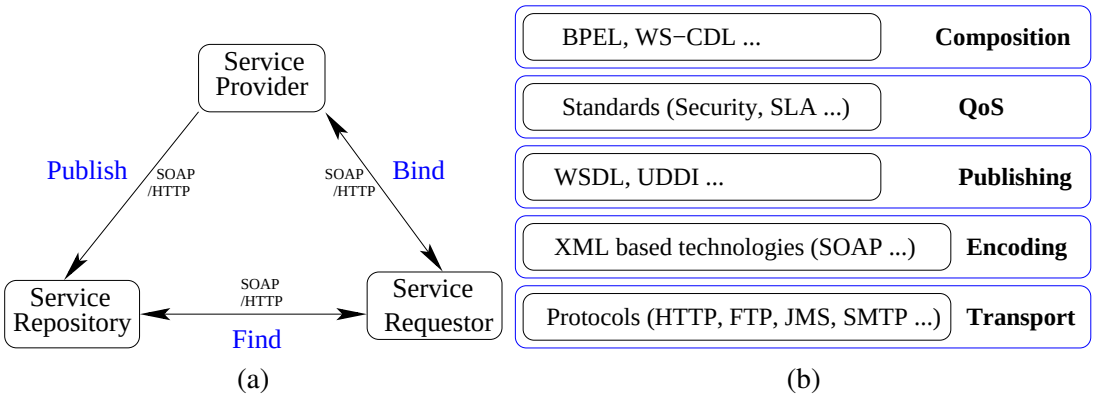


Figure 1.1: (a) Architecture of Web Services (b) Web Services Stack

1.2.1 UDDI[2]

Universal Description, Discovery, and Integration (UDDI) is a protocol for describing available Web services components. This standard allows businesses to register with an Internet directory that will help them advertise their services, so companies can find one another and conduct transactions over the Web. This registration and lookup task is done using XML and HTTP(S)-based mechanisms.

UDDI is a mechanism for clients to dynamically find other web services. Using a UDDI interface, businesses can connect to services provided by external business partners. An UDDI registry can be thought of as a CORBA trader, or a DNS lookup service for business applications. A UDDI registry has two kinds of clients: businesses that want to publish a service (and its usage interfaces), and clients who want to obtain certain services to fulfil their goals and bind programmatically to them. UDDI is layered over SOAP[5] and assumes that requests and responses are UDDI objects sent around as SOAP messages. A typical query is as follows.

Example 1 *Suppose CCNService is provided by a business CCN Corp.* `CCNService` *takes as input social security number (*`SSN`*) and produces as output the credit card number (*`CCN`*). Then the following query, when placed inside the body of the SOAP envelope, returns details on CCN Corp.*

```
<find_business generic="1.0" xmlns="urn:uddi-org:api"> <name>CCN
Corp</name> </find_business>
```

The result is a detailed listing of <businessInfo> elements currently registered for CCN Corp, which includes information about the UDDI service itself.

```
<businessList generic="1.0"
  operator="CCN Corporation"
  truncated="false"
  xmlns="urn:uddi-org:api">
  <businessInfos>
   <businessInfo
       businessKey="0076B468-EB27-42E5-AC09-9955CFF462A3">
    <name>CCN Corporation</name>
     <description xml:lang="en">
           Providing CCN query solutions
     </description>
      <serviceInfos>
       <serviceInfo
          businessKey="0076B468-EB27-42E5-AC09-9955CFF462A3"
          serviceKey="1FFE1F71-2AF3-45FB-B788-09AF7FF151A4">
        <name>Web services providing service 1</name>
       </serviceInfo>
      <serviceInfo
          businessKey="0076B468-EB27-42E5-AC09-9955CFF462A3"
          serviceKey="A8E4999A-21A3-47FA-802E-EE50A88B266F">
        <name>Web services providing service 2</name>
      </serviceInfo>
      </serviceInfos>
    </businessInfo>
  </businessInfos>
 </businessList>
```

1.2.2 SOAP[5]

Simple Object Access Protocol (SOAP) is the pervasive protocol for wrapping Web Services payloads for messages through transport layer. This standard is used for binding to clients as well as publishing to a UDDI Service. A soap message consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses. SOAP makes object access simple by allowing applications to invoke object methods or functions, residing on remote servers. A SOAP application creates a request block in XML, supplying the data needed by the remote method as well as the location of the remote object itself. We would require the mechanism to both wrap into and unwrap from SOAP messages at both participants of the binding. The *body* of the SOAP *envelope* would contain its payload, which can be interpreted with the XML schema associated with it.

Example 2 *A SOAP enveloped request to the CCNService service would be as follows:*

```
POST /CCNService HTTP/1.1 Host: www.ccnserviceserver.com
Content-Type: text/xml; charset="utf-8" Content-Length: n
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
    <m:GetCCN
     xmlns:m="Some-URI">
      <SSN>XXXXXX</SSN>
    </m:GetCCN>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A SOAP enveloped response to the CCNService service:

```
HTTP/1.1 200 OK Content-Type: text/xml; charset="utf-8"
Content-Length: n

<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
 <SOAP-ENV:Body>
  <m:GetCCNResponse
   xmlns:m="Some-URI">
   <CCN>YYYYYYY</CCN>
  </m:GetCCNResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

1.2.3 WSDL[3]

While publishing services, a cogent description is required to lend itself to discovery. For this purpose, Web Services Description Language (WSDL), is the accepted standard. It is a template for how services should be described and bound by clients. Once, the developer exposes a function through the services, WSDL is used to generate the interfacing points of the service. It may be a bunch of files (WSDL, WSDD etc), which together establish what the service does and how to access it, similar to IDL (Interface Definition Language) used for remote invocation procedures such as in CORBA.

The repository where the services are published such as UDDI would list the corresponding WSDLs describing the services. It would have the information about how to find it, bind and invoke the service. In WSDL, custom abstract definitions of data, ports for binding pave the way for reuse. [3] summarizes the various aspects as follows.

- Types : A container for data type definitions using some type system (such as XSD).

- Message : An abstract, typed definition of the data being communicated.
- Operation : An abstract description of an action supported by the service.
- Port Type : An abstract set of operations supported by one or more endpoints.
- Binding : A concrete protocol and data format specification for a particular port type.
- Port : A single endpoint defined as a combination of a binding and a network address.
- Service : A collection of related endpoints.

Example 3 *For the example service, the WSDL would be as follows:*

```
<?xml version="1.0"?> <definitions name="CCN"
              targetNamespace="http://example.com/ccnservice.wsdl"
              xmlns:tns="http://example.com/ccnservice.wsdl"
              xmlns:xsd1="http://example.com/ccnservice.xsd"
              xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
              xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
   <schema targetNamespace="http://example.com/ccnservice.xsd"
           xmlns="http://www.w3.org/1999/XMLSchema">
    <element name="SSN">
     <complexType>
      <all>
       <element name="ssn" type="string"/>
      </all>
     </complexType>
    </element>
    <element name="CCN">
     <complexType>
      <all>
```

```
      <element name="ccn" type="string"/>
     </all>
    </complexType>
   </element>
  </schema>
 </types>

<message name="GetCCNInput">
  <part name="body" element="xsd1:SSN"/>
</message>
<message name="GetCCNOutput">
  <part name="body" element="xsd1:CCN"/>
</message>

<portType name="CCNServicePortType">
 <operation name="GetCCN">
  <input message="tns:GetCCNInput"/>
   <output message="tns:GetCCNOutput"/>
 </operation>
</portType>

<binding name="CCNServiceSoapBinding"
        type="tns:CCNServicePortType">
<soap:binding style="document"
 transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetCCN">
   <soap:operation
    soapAction="http://example.com/GetCCN"/>
```

```
      <input>
        <soap:body use="literal"
         namespace="http://example.com/ccn.xsd"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
     <output>
      <soap:body use="literal"
        namespace="http://example.com/ccn.xsd"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
     </output>
    </operation>
 </binding>

 <service name="CCNService">
  <documentation>provides CCN from SSN</documentation>
   <port name="CCNServicePort" binding="tns:CCNServiceBinding">
    <soap:address location="http://example.com/ccnservice"/>
   </port>
 </service>

</definitions>

 <binding name="CCNServiceBinding"
  type="CCNServiceType">
   <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getCCN">
     <soap:operation
```

```
        soapAction="http://www.getCCN.com/GetCCN"/>
         <input>
          <soap:body type="InMessageRequest"
           namespace="urn:ccn"
           encoding="http://schemas.xmlsoap.org/soap/encoding/"/>
         </input>
         <output>
          <soap:body type="OutMessageResponse"
           encoding="http://schemas.xmlsoap.org/soap/encoding/"/>
         </output>
       </operation>
     </binding>
     <service name="CCNService">
      <documentation>provides CCN from SSN
      </documentation>
       <port name="CCNServicePort"
        binding="tns:CCNServiceBinding">
        <soap:address location="http://example.com/ccnservice"/>
       </port>
      </service>
</definitions>
```

1.2.4 Technologies

Once the mechanism to publish, find and bind services falls in place, different layers can be introduced to weave a network of many such services in the real world. Based on the elements described earlier, the Web Services world has evolved to use complex technologies and new protocols for different aspects, such as security, quality of service as described in the technology stack in Figure 1.1(b). There are many IDEs to create web services in a semi-automated manner (generating the interface files

automatically from minimal user input) and once a service provider creates a web service, it can be published in a ebXML registry, like UDDI registry described above. The core architectural foundations of Web services are based on XML which are being developed in parallel by different vendors. Representational state transfer (REST) [81] can be used for a model dealing away with the complexity of SOAP. The web services conforming to REST are said to have a RESTful design. There are a number of mechanisms for constructing Web services. Microsoft prefers object-oriented language C# as the development language for Web services and .NET framework. The back end database such as a Microsoft SQL Server. Similarly, Sun Microsystems has its own set of technologies and tools for facilitating Web services development. Java Servlets, Java Server Pages (JSPs), Enterprise JavaBeans (EJB) architecture and other Java 2 Enterprise Edition (J2EE) technologies can be used for developing Web services, such as is developing Java APIs for XML-based remote procedure calls and for looking up services in XML registries JAX/RPC (Java API for XML Remote Procedure Calls) and JAXR (Java API for XML Registries). IBM have also pitched in with Web Services Toolkit (EETTK), WSDL Toolkit, and Web Services Development Environment (WSDE). Oracle had developed a plug-in for Business Process Execution Language (BPEL) which provides a layer facilitating composing Web Services manually. They provide IDEs for drag and drop of service interfaces and composing them into a workflow. There are a host of tools for composition which will be summarized in the next chapter.

1.3 Semantic Web Services

Though Web Services had a major impact on enterprise business since its inception, the technologies described so far provide more of a syntactic level description of their functionalities. To lend meaning to the attributes of communication, metadata has been extensively used as is the norm in XML based protocols. But this would limit the boundaries to knowing what a service does, and how to invoke it or who published it. But, to practically compose services without manual intervention in complex business scenarios the Web Services paradigm needs to satisfy a broader goal of having machine readable semantics. Otherwise inter-operability is largely limited. Thus, Semantic Web Services (SWS) have evolved by providing rich formal descriptions of their capabilities. This would help in automating various tasks in an open environment [68]. This is part of a broader aspect of research in

classifying data according to semantics which has led to the emergence of semantic web. In the next section, we briefly explore what a semantic web service would entail.

1.3.1 OWL-S

OWL-S (previously DAML-S])[8] consists of a set of ontologies designed for describing and reasoning over service descriptions. It leverages the Resource Description Framework (RDF) . RDF is a data model describing relations between different resources. This is aided by the RDF schema which provides the associated vocabulary. The origin of this approach can be traced to AI, where it was used for describing Agent functionality in Multi-Agent Systems and in Planners for solving high level goals. OWL comes in different flavors of expressiveness such as OWL-Lite, OWL-DL, and OWL-Full. When adapted in the context of Web Services, it can be split into three ontologies the Profile, Process Model and Grounding. The Profile, which is used for finding services. The inputs, outputs, preconditions and effects of the function performed by the service along with some manual inputs such as its name, quality of service parameters are used to determine the service description. While inputs and outputs directly pertain to the service preconditions and effects pertain to the environment of execution. Preconditions are facts that should be asserted for the service to execute and effects are the facts that become asserted after the execution of the service. Profiles contribute to a universe of Profile Taxonomy from which other profiles can be derived. The process model is used for validating the composition of services. Processes can be of three types. Atomic processes are indivisible units having inputs and outputs. simple processes are the layer of abstractions detailing how to use the service. They are like a shell which have to be associated with atomic processes through discovery or dynamic binding to bind to a physical service. Composite processes can be hierarchically combined workflows of Atomic, Simple as well as other composite processes. Service grounding addresses the means to access a service by defining communication protocols, message formats etc.

1.3.2 WSMF

The Web Service Modeling Framework (WSMF) [9] provides a model for capturing the myriad aspects related to Web services. The model stresses loose coupling of the different components and a

mediator to enable communication among the components as needed. In our research we would use the term choreographer. Service mediation can be termed as Choreography meaning the view from outside how messages are exchanged between different components, or Orchestration is like a composer's view to compose services to realize a particular goal. The WSMF framework believes in scalability in supporting more data structures; adding business logics and message exchange protocols. The main elements of WSMF are ontologies standardizing the terminology used by the elements; goal repositories containing composition capabilities which should be realized by choreographing the services ; the description of the services themselves and mediators for realizing the goals. WSMF led to the projects, Semantic Web enabled Web Services (SWWS) [7]; and WSMO (Web Service Modeling Ontology) [6]. SWWS looks into providing a framework for description, discovery and mediation. WSMO is concerned with formalizing the service ontology defining goals, mediators and services. WSMO uses F-logic as the language for describing the various aspects and the inter-operability issues are handled in mediators to follow the philosophy of loose coupling.

1.4 Benefits of Web Services

Web Services as a form of SOA has manifold benefits. The most noticeable aspect in the SOA paradigm is the fact that custom made functions can be exposed to a network for discovery by others. Thus the functions become accessible via standard protocols such as HTTP. This opens up the idea of collaborating of services to make up newer services.

Heterogeneously developed functions in different languages can be participants in a collaborated Web Services development. The low coupling aspect allows any function that is needed for a composition to be invoked as on the uppermost layer Web Services would follow the same protocol. This makes the collaboration platform and technology independent. Of course, a collaboration would require matching in terms of the service level agreements as well as semantic matching.

Standardized protocols as described in the Web Services stack, for transport, messaging, description and discovery aid the development of Web Services. These make the job of developing new services easier by offering the layer of abstraction they provide. Internet being the primary method of transport allows for a cheaper alternative than proprietary B2B solutions. Similarly, flexibility is another key

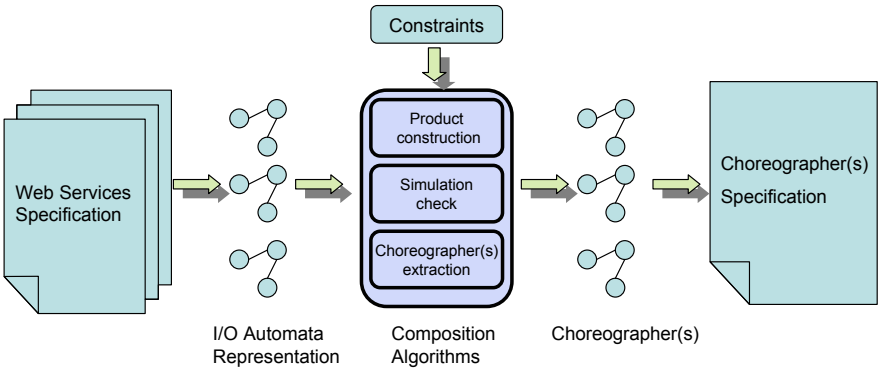


Figure 1.2: Automated composition

freedom offered to Web Services developers. One can use their own reliable transport mechanism for communication and their choice of languages and platforms for development. Web Services can be implemented from over ftp protocols to following a RESTful design as mentioned earlier. Similarly, Web Services discovery make it easier to find new opportunities in expanding business easier. Web Services being self-describing modules make development time short, by allowing loose coupled components to be made simultaneously.

Web Services have opened up the world of new business collaborations by making it easier to publish and discover services. Support of technologies such as EAI, EDI, B2B, Portals for distributed computing helps leveraging existing businesses to spread into other technologies. Thus, legacy applications can still be used fitted with Web Services interfaces, and at the same time those applications can be changed or modernized in the back end to ring in new technologies.

1.5 Research Goals

There has been number of aspects of research in Web Services. These range from theoretical research for modeling semantics of data exchange, services to efficient implementation of services using standard technologies. Given the volume of Services, a natural thing would be to compose new services from existing ones. Since, in essence, the paradigm involves data exchange in the core, match making and ensuring following of standards is an important issue. There has been a plethora

of specifications and they are undergoing constant changes for arriving at an international standard. The specifications are generally built upon XML. But a more important aspect is to formalize the process of composition of services. Thus, ongoing research is also focused on how to develop the core model for SOA, whatever the agreed specifications may be. Web Services choreography refers to the collaborative approach, where each party involved in the process describes the part they play in the interaction. Choreography tracks the sequence of messages that may involve multiple parties and multiple sources. It is associated with the public message exchanges that occur between multiple Web services. In Figure 1.2, Web Services are modeled as automata and composed for realizing newer services. The first task involves conversion from a high level language for describing services such as WSDL, BPEL, OWL to the model of choice. External constraints such as a communication cost model can serve as conditions to be met by the composition. After composing the services, the output model needs to be transformed into the language suited for discovery and use. And, inside the composition framework, the problem of the mediation of services is tackled - How can we know if a goal is realizable or not? If at all, what will be the model of the choreographer? If there are multiple choreographers possible, how can we minimize the communication and computation overhead to choose an optimal choreography scheme? These are the major questions addressed in this dissertation.

CHAPTER 2. Composing Web Services through Choreography: Overview and Prior Work

2.1 Centralized Choreography

Centralized Choreography of services refer to the cooperative interaction of services with a single mediator. The central mediator participates in all the messages exchanges between a service and another service or the end user. The main research goal of Composing services is to find the existence and extraction of the mediator referred to as the Choreographer. A number of approaches has been developed in the recent past to detect and/or synthesize choreography based composition of services. The techniques range from manual development of choreography to more rigorous automated procedures that rely on formal methods. The techniques applying formal methods to service composition are typically based on automata theory, dynamic logic and AI planning. In the following we discuss some of the representative works in this domain.

In [21] the authors describe services as automata extended with queue and allow exchange of messages between services in an asynchronous fashion. A global watcher is developed to keep track of messages being exchanged in the composition to detect whether a specified goal service is realizable. Subsequently in [22] the authors use Spin model checker [54] to verify whether a composition correctly replicates the required goal. Also in [48], they have dealt with verification of service conversations. The message passing framework is extended in [19] which uses satisfiability of propositional dynamic logic to detect the existence of a choreographer. The work is further generalized in [20] where non-determinism in the service behavior is considered. A similar approach on service composition is presented in [56], where the main objective is to create a choreographer which makes the composition behavior bisimulation equivalent to the goal. [34] introduces the concept of lookahead for choreographers to plan ahead in delegating activities, based on the approach in [33].

Several works investigated the applicability of AI planning techniques for service composition. These works apply techniques ranging from rule based planning [38], Situation Calculus [28, 45], query planning [43] and theorem proving [31], model checking [27, 42, 26]. In essence, the techniques reduce the problem of composition to that of planning a desired execution of workflows. The underlying basis in the planning domain also rely on state transition systems with states, actions and observations. The services communicate through messaging which again emphasizes input-output behavior of services.

2.1.1 Tools

There are a number of tools/frameworks for facilitating composition of services. Some of these provide a new IDE or a plug-in to existing tools to orchestrate or choreograph the services manually. Others address the issue of automated composition through tools, based on various representations of services. In [69], the authors provide a tool where the control flow logic in mediated composite services specified in BPEL is verified against the design specified using Message Sequence Charts (MSC) and finite state processes. In [52], finite automata are augmented with XML messages and XPath expressions as the basis for verifying temporal properties of the conversations of composite web services. SWORD [70] provides a composition model that can instantiate and execute generated plans using AI techniques. The execution model includes a filter mechanism for user interactions and any other necessary intermediate computation. Among Planning based tools in the domain, MBP [27] provides synthesis, validation and simulation. Subsequently, other tools [25, 66, 51, 58] which use hierarchical planning, PDDL were also developed. [13, 10, 11] provide a summary of the research based tools which have been developed till now.

Logic based solution to address service composition problems is not new. In [28], the authors apply the golog language based on Situation Calculus for composing services. [60] uses structural synthesis of program for composition whereas [59, 31] uses theorem provers. [12] uses Linear Logic to realize a semantic composition. [74] proposes a methodology to build web services.

Recently, [50] provides extension of the work in [51] to compare on-the-fly and once-for-all process level composition. Here, the authors refer to on-the-fly as composing available services to serve one-shot user request on an “as needed” basis, whereas, once-for-all composition builds a composition

which can serve multiple user requests. Research on formalism for composition based on process algebra approach has been the focus in [71],[72], [73]. In [65], the authors describe a tool which builds on the METEOR-S Web Service composition framework by adding the abstract process designer, constraint analyzer, optimizer and binder module. In [64], the authors provide several visual facilities to ease the definition of a business process such as multiple views of a process, syntactic and semantic awareness, filtering, logical zooming capabilities and hierarchical representations. [67] introduces a lightweight tool for specification, composition and execution. In [61], the authors provide a tool for composing on the fly in a pervasive computing environment. In [62], authors provide a tool for analysis of BPEL composed services.

2.2 Decentralized Choreography

The centralized mechanism of a single choreographer entails invoking services and relaying messages for all the required computation or communication transactions. However, in most practical scenarios, there is a cost associated with each computation and communication and as such, it is important to identify a choreography mechanism which incurs the minimum overall cost. The cost can be defined in terms of network bandwidth, congestion, or pure monetary value for using some network or service. In one of the subsequent chapters, we address the problem of minimizing the overall cost of choreography by synthesizing an optimum decentralized choreography strategy. Decentralization amounts to generating multiple choreographers; one for each service in the composition. The objective is to eliminate the requirement of a single choreographer's (typically located at the client site) involvement in every computation or communication transaction.

So far, a lot of work had undergone to realize automatic composition of web services. Service composition referred to as orchestration or choreography in the literature normally follows the approach of peer to peer communication or a centralized mediator. Automated composition has been addressed in representative work such as [51, 47, 19, 56, 13]. Decentralization deals between these two extremes. Peer to peer web service communication is the version without any intermediary and it has been pursued in works such as [47]. The merits of distributing the data storage, execution structure is discussed in [78]. Workflow partitioning of state chart models have been dealt earlier in [76]. Thus in essence,

decentralization techniques are applied to models which are nothing but graph structures. Such schemes based on partitioning a Program Dependence Graph has been shown in [75]. Following the same path, the decentralizing the execution of web services has been dealt in [40]. In the recent past, they have introduced a composition tool called Synthy. [39] proposes a complete decentralization scheme with service invocation triggers to route traffic efficiently. More recently, in [79] a formal model is proposed which handles synchronization and concurrency constraints in decentralizing service compositions. Decentralization entails security constraints and such issues are addressed in recent works [77].

In contrast to the program partitioning approach, our work is based on optimizing cost in an automata theoretic setting and not directly partitioning executable code. The approach will yield the optimal decentralization scheme, which can then be implemented in the language chosen. Unlike the program partitioning schemes our approach can be applied at the modeling level of service composition.

Decentralization of computation mainly aims to move away from the simple mediator based approach to a more sophisticated scheme. [80] gives an overview of recent and ongoing approaches, such as BondFlow, Symphony, OSIRIS. Decentralization of composition has been researched in authors in [40] based on program dependence and graph partitioning. The industry standard workflow models are moving from simple centralized scheme towards a more loosely coupled approach by distributing the workflow, more suited to lightweight mobile technologies [83, 84, 87, 88]. [85] talks about late binding of services for communication optimization. To this effect, [86] proposes a hybrid model specifying centralized control-flow and distributed data-flow. [89] introduces a peer-to-peer approach of service communication, from a centralized specification. In contrast our approach guarantees the minimization of the overall cost, the valuation of which can be generic and context based. No manual intervention for obtaining the optimal solution is necessary as our approach is automated.

Similar to the existing work, we use transition system based representation of services, more specifically, we use i/o automaton to capture the input output interfaces of each service. In fact, as noted in [49] treating a Web service as an automaton comes naturally, as it is equipped with i/o capabilities and from the point of view of composition, we are interested in the i/o functionality of the automaton. One of the distinguishing aspect of our technique is that the proposed automata theoretic approach provides

valuable insights to the composition problem with respect to the *capabilities* of the choreographer. We provide a uniform solution methodology to the choreographer existence problem for centralized as well as decentralized settings. We provide algorithms to realize goals which are loop-free and are like workflows and also to realize goals which have loops.

CHAPTER 3. Centralized Choreography: Formulation, Existence and Synthesis

Driving Problem. In this chapter, we investigate automata-theoretic approach to address the problem: *given a set of services and a goal, is it possible to generate a choreographer which can communicate with the services to realize the goal?* In our technique, we consider existence and synthesis of two types of choreographers: simple choreographer and transducing choreographer. Simple choreographer is only capable of relaying output from one service to input of another, i.e., choreographer does not buffer any output for later use. On the other hand, transducing choreographer is capable of storing already seen inputs and outputs and using them to provide inputs to services at a later stage. The phenomenon is referred to as *inducing* as the choreographer can induce a service to respond to a stored input. Furthermore, the transducing choreographer can consume or absorb outputs from services that are not necessary to realize the goal. This is referred to as *hiding*. In short, the transducing choreographers can store, use and ignore inputs and outputs to and from services as and when necessary.

Our Solution. Given a set of services, our solution relies on identifying all possible behaviors that can be realized from the services using a simple or transducing choreographer. If the goal behavior is simulated by all possible composed behavior, we infer that synthesizing a choreographer is possible for the said goal; otherwise choreographer does not exist.

3.1 Illustrative Example

Consider the sequence diagrams in Figure 3.1; (a) and (b) which show the input-output behavior of two existing services. The sequence of activities can be obtained by top-down scanning of the corresponding diagram. S_1 takes as input social security number (`SSN`) and produces as output the credit card number (`CCN`); while S_2 takes as input `CCN` twice and outputs credit approval (`Appr`) and

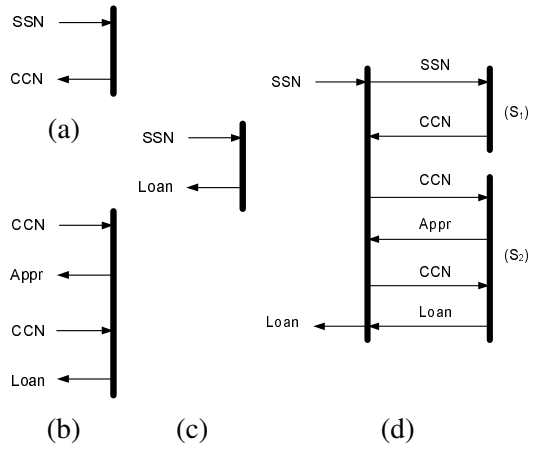


Figure 3.1: (a) Service S_1, (b) Service S_2, (c) Required Service S_G and (d) Composed Services.

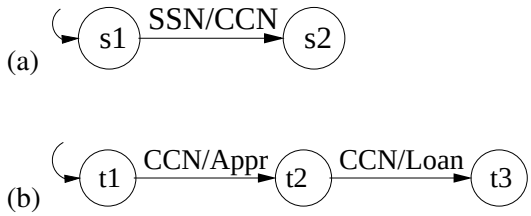


Figure 3.2: Automata: Services (a) A_1, (b) A_2.

loan approval (`Loan`). The inputs to the services (when functioning on their own) come from their environment and outputs are provided to the environment. By environment, we mean the client or user who is using the services.

Assume that the developer wants to create a new service (referred to as goal service) S_G which takes as input `SSN` from the client and outputs `Loan` (Figure 3.1(c)). The existing services S_1 and S_2 do not provide the required S_G. However, an intermediary or a choreographer can be synthesized which relays and controls information between S_1 and S_2 and replicates the input-output behavior of S_G. Figure 3.1(d) shows the choreographed S_1 and S_2. Note that the choreographer acts as the environment of the services; it relays the `SSN` input from the client to S_1 and stores the `CCN` output from S_1. It does not relay `CCN` to the client (it hides this output), instead the choreographer provides `CCN` as input to S_2 to obtain the outputs `Appr` and `Loan`. Finally, the choreographer sends the `Loan` to the client. The

objective is to identify the conditions under which such a choreographer can be synthesized given a set of existing services and a goal.

3.2 Preliminaries

3.2.1 Services as Automata

We describe the (goal and component) services as input/output automata whose states represent the configuration of the services and interstate transitions define the way in which the services evolve from one configuration to another. Each transition is enabled in the presence of a specific input and results in a specific output. Formally, input/output automaton is defined as follows:

Definition 1 (I/O Automaton) *An i/o automaton A is defined by a tuple (X, X^0, I, O, T), where, X is the set of states, $X^0 \subseteq X$ is the set of initial states, I is the set of inputs, O is the set of outputs and $T \subseteq X \times I \times O \times X$ is the set of transitions. An element of T represented by $t = (x, i, o, x')$ is such that $x \in X$ is the origin state of the transition, $i \in I$ is the input to the transition, $o \in O$ is the output of the transition and $x' \in X$ is the destination state of the transition. We will often use $x \xrightarrow{i/o} x'$ to denote $(x, i, o, x') \in T$.*

Example 4 *Consider the automata representation of services (Figure 3.1(a,b)) in Figure 3.2. The input-labels of each transition in Figure 3.2 are the inputs to the services from their corresponding environment, while the output-labels are the outputs provided by the service following an input. The sequencing in the service automata can be obtained from the sequence of input-output in its sequence diagram. WLOG we assume that the input and output always alternate*[1].

3.2.2 Role of a Choreographer

Recall that the problem of service composition is to identify whether a set of existing services can be used to realize a specified goal (a new service). The goal can be represented by an i/o automaton and an existing service is said to realize the goal if all possible behaviors of the goal is also present in the service.

[1] *A sequence of inputs followed by outputs can be easily represented if transition labels are sequence-based.*

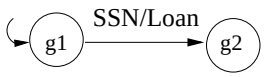


Figure 3.3: Automaton: Goal A_G.

Typically however, a goal cannot be realized from one service. For example, consider the goal automaton in Figure 3.3. It cannot be realized from any of the services in Figure 3.2(a,b). In such cases, a choreographer is required to control the services such that their composition with the choreographer realizes the specified goal. In our setting there are two important features of such a composition: (a) a service does not communicate with the client or other services directly and (b) the choreographer cannot produce any input on its own; instead it relies on services or the client for the inputs. For example, the service automaton A_1 in Figure 3.2(a) requires an input `SSN` and provides an output `CCN`. A choreographer can transfer the `SSN` input from the client or another service to A_1 and can transfer the `CCN` from A_1 to the client or another service can consume it.

The main idea behind choreographer synthesis is to a realize each goal-transition by executing certain service-transitions in a certain sequence, and relaying the inputs/outputs of the service-transitions in accordance to that sequence. The input to the first service-transition in the sequence is the same as the goal-transition input, the output from the last service-transition in the sequence is the same as goal-transition output, and the output from any other service-transition is the same as the input to the next service-transition in the sequence. Such a goal-transition is said to belong to the closure of the sequence of service-transitions. We can also project such a sequence of service-transitions on individual services to obtain a set of sequences, each one of which belongs to a single service. We then say that the goal-transition is realizable from that set of sequences. The set of all possible service-transition sequences are first obtained by taking the interleaving product of the service-automata. A "closure" operation is then performed over the sequences of service-transitions (obtained through interleaving) to identify the goal-transitions that a certain sequence of service-transitions is able to realize. In the following we formalize these notions.

Before proceeding to formally define a choreographer, we present the notational convention and background definitions. We will use α, β, μ, ν and their subscripted versions to define sequences.

Concatenation will be denoted by "." (dot). For any two strings α and β, $\alpha \preceq \beta$ denotes that α is a prefix of β.

Definition 2 (Interleaving product of sequences) *The interleaving product of sequences* ($\|$) *is defined inductively as:* $\epsilon \parallel \alpha = \alpha \parallel \epsilon = \alpha$; $(i/o.\mu) \parallel (i'/o'.\nu) = i/o.(\mu \| i'/o'.\nu) + i'/o'(i/o.\mu \parallel \nu)$.

Definition 3 (Closure) *The closure of a string* $\alpha = i_1/o_1.i_2/o_2 \ldots i_n/o_n \in (I \times O)^*$, *denoted by* $\mathcal{CL}(\alpha)$, *is equal to*

$$\left\{ \begin{array}{r} i_1/o_1 \ldots i_{m-1}/o_{m-1}.i_m/o_k.i_{k+1}/o_{k+1} \ldots i_n/o_n \mid \\ \exists m < k : \forall j, m \leq j < k : i_{j+1} = o_j \end{array} \right\}$$

For example,

$$\mathcal{CL}(a/b.b/c.c/d) = \{a/b.b/c.c/d,\ a/c.c/d,\ a/d,\ a/b.b/d\}$$

Definition 4 (Realizability) *A given* $\alpha \in (I \times O)^*$ *is said to be realizable from* $\{\beta_n \mid \beta_n \in (I_n \times O_n)^*\}$, *where* $I \subseteq \bigcup_n I_n$ *and* $O \subseteq \bigcup_n O_n$, *if and only if* $\exists \mu \in \|_{n \leq N}\ \beta_n : \alpha \in \mathcal{CL}(\mu)$, *where* $\|$ *denotes interleaving product.*

Let $\alpha = a/d$, $\beta_1 = a/b.c/d$ and $\beta_2 = b/c$. Then $\beta_1 \parallel \beta_2 = \{a/b.c/d.b/c,\ a/b.b/c.c/d,\ b/c.a/b.c/d\}$ and there exists a sequence $\mu = a/b.b/c.c/d$ in the above set, the closure of which includes a/d (see example after Definition 3). As $\alpha \in \mathcal{CL}(\mu)$, α is said to be realizable from $\{\beta_1, \beta_2\}$.

Using the definition of closure and realizability, we next define the notion of a choreographer, which maps a sequence belonging to a goal to a set of sequences, one for each available service. The mapping must be causal (i.e., prefix-preserving) for it to be implementable and also must satisfy the requirement of realizability.

Definition 5 (Choreographer) *Given a goal automaton* $G = (X_g, X_g^0, I_g, O_g, T_g)$ *and* N *service automata of the form* $A_n = (X_n, X_n^0, I_n, O_n, T_n)$ *for* $n \leq N$ *such that* $I_g \subseteq \bigcup_n I_n$ *and* $O_g \subseteq \bigcup_n O_n$, *a choreographer is function of the form* $C : (I_g \times O_g)^* \rightarrow \prod_{n \leq N} (I_n \times O_n)^*$ *such that*

1. $\forall \mu, \nu \in (I_g \times O_g)^* : \mu \preceq \nu \Rightarrow \forall n \leq N : C_n(\mu) \preceq C_n(\nu)$

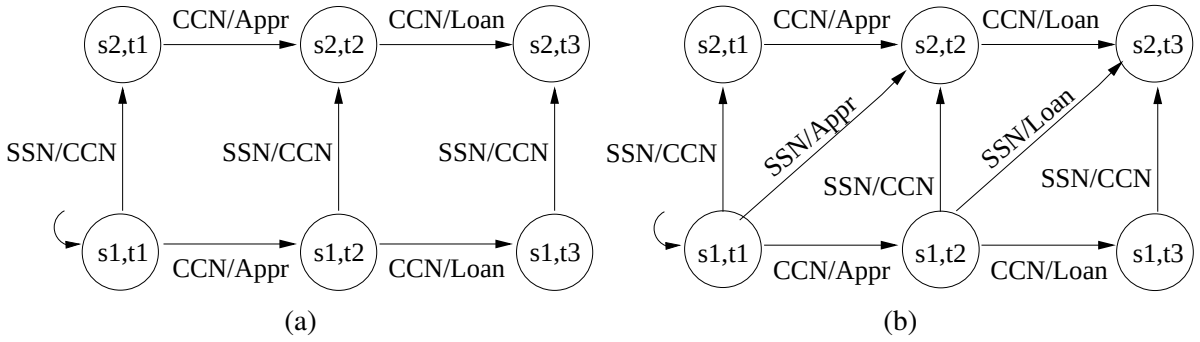


Figure 3.4: (a) $A_1 \| A_2$-automaton and (b) U-automaton.

2. $\forall \mu \in (I_g \times O_g)^* : \mu$ *is realizable from* $\{C_n(\mu)\}$

We present below the automata-theoretic approach to verify the existence of a choreographer for a given set of the services and a specified goal.

3.3 Existence of Choreographer

We first consider the case where the choreographer is capable of only relaying output from one service to input of another. It does not store and/or reuse any inputs from the client or outputs from the existing services. The functionality is denoted by the closure defined in Definition 3.

To identify the existence of such a choreographer, we first define an interleaving product automaton from the existing service automata. This automaton captures all possible interleaved behavior of the participating service automata.

Definition 6 (Interleaving Product) *Given a set of service automata* $A_1, A_2, \ldots, A_N$ *where* $A_n = (X_n, X_n^0, I_n, O_n, T_n)$ *for* $1 \leq n \leq N$*, the interleaving product of* $\{A_n\}$ *is defined as an automaton* $\|_n A_n = (X, X^0, I, O, T)$*, where,* $X \subseteq X_1 \times X_2 \times \ldots X_N$ *is the set of states,* $X^0 = X_1^0 \times X_2^0 \times \ldots X_N^0$ *is the set of initial states,* $I = \bigcup_{1 \leq n \leq N} I_n$ *is the set of inputs and* $O = \bigcup_{1 \leq n \leq N} O_n$ *is the set of outputs. Finally,* $T \subseteq X \times I \times O \times X$ *is the set of transitions such that*

$$(x_1, \ldots, x_N) \xrightarrow{i/o} (x'_1, \ldots, x'_N) \in T \;\Leftrightarrow$$
$$\exists i \leq N : x_i \xrightarrow{i/o} x'_i \in T_i \;\wedge\; \forall j \leq N, j \neq i : x'_j = x_j$$

Example 5 *Figure 3.4(a) shows the automaton $\|_n A_n$ obtained from service automata A_1 and A_2 in Figures 3.2(a) and (b). Each state in the $\|_n A_n$ is annotated with the tuple (i, j) where i represents the state of A_1 and j represents the state of A_2.*

The $\|_n A_n$ represents the set of behaviors that can be realizable from the interleaving of services A_ns. To identify the sequences that can be obtained through some choreography, we define the operation closure of an automaton, which follows from the Definition 3.

Definition 7 (Closure of I/O Automaton) *Given an i/o automaton $A = (X, X^0, I, O, T)$, the closure of A, denoted by A^*, is defined as (X, X^0, I, O, T^*) where*

$$x \xrightarrow{i_1/o_k} x' \in T^* \Leftrightarrow \exists k : \left[\begin{array}{l} x \xrightarrow{i_1/o_1} x_2 \in T \wedge \\ x_2 \xrightarrow{i_2/o_2} x_3 \in T \wedge \\ \ldots \\ x_k \xrightarrow{i_k/o_k} x' \in T \wedge \\ \forall 1 < j \leq k : i_j = o_{j-1} \end{array}\right]$$

The universal service automaton, denoted `U`*, is defined as the closure of the service-product automaton, i.e.,* $\texttt{U} := (\|_n A_n)^*$.

Note the above definition of closure ensures that for all $\mu \in A$, $\mathcal{CL}(\mu) \subseteq A^*$.

Example 6 *Going back to the example services A_1 and A_2 in Figures 3.2(a) and (b), the corresponding* `U`*-automaton obtained from $\|_n A_n$ of Figure 3.4(a) is shown in Figure 3.4(b). Observe that the* `U`*-automaton also includes transitions that can be realized when a choreographer relays output of one service to the input of the other. For example, the transition $(1,1) \xrightarrow{\texttt{SSN/Appr}} (2,2)$ is generated from the transitions $(1,1) \xrightarrow{\texttt{SSN/CCN}} (2,1)$ and $(2,1) \xrightarrow{\texttt{CCN/Appr}} (2,2)$. It captures the situation when a choreographer sends the* `SSN` *input (from client) to A_1 at state 1 and relays the* `CCN` *output from A_1 to the input of A_2 at its state 1. As a result, both A_1 and A_2 move from their corresponding states 1 to 2.*

`U`-automaton contains all possible choreographed behavior of the services. Proceeding further, the goal automaton A_G is realizable from the services using some choreographer if and only if all possible behavior of A_G is simulated by `U`-automaton.

Definition 8 (Simulation [55]) *Given an i/o automaton $A = (X, X^0, I, O, T)$, for all x_1 and x_2 in X, x_1 is simulated by x_2 if they are related by the largest simulation relation denoted by $x_1 \sqsubseteq x_2$ and defined as: $x_1 \sqsubseteq x_2 \Leftrightarrow [\forall x_1' : x_1 \xrightarrow{i/o} x_1' \Rightarrow (\exists x_2' : x_2 \xrightarrow{i/o} x_2' \wedge x_1' \sqsubseteq x_2')]$*

An i/o automaton $A_1 = (X_1, X_1^0, I_1, O_1, T_1)$ is said to be simulated by $A_2 = (X_2, X_2^0, I_2, O_2, T_2)$, denoted by $A_1 \sqsubseteq A_2$, if all states in X_1^0 is simulated by some state in X_2^0.

Theorem 1 *Given a goal A_G and a set of services $A_1, A_2, \ldots, A_N$, the goal can be realizable from the composition of A_ns with a choreographer if and only if $A_G \sqsubseteq \mho$ where $\mho$ is the closure of the $\|_n A_n$ obtained from $\{A_n\}$.*

Proof: Let $A_G = (X_g, X_g^0, I_g, O_g, T_g)$ and $\mho = (X, X^0, I, O, T^*)$.

$$
\begin{aligned}
A_G \sqsubseteq \mho \quad &\Leftrightarrow \quad \forall x_g^0 \in X_g^0 : \exists x^0 \in X^0 : x_g^0 \sqsubseteq x^0 \\
&\Leftrightarrow \quad \left(\begin{array}{l} \forall x_g^1 : x_g^0 \xrightarrow{i_1/o_1} x_g^1 \in T_g \Rightarrow \\ \exists x^1 : x^0 \xrightarrow{i_1/o_1} x^1 \in T^* \wedge x_g^1 \sqsubseteq x^1 \end{array} \right)
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
A_G \sqsubseteq \mho \quad \Leftrightarrow \quad & \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m \\
& \text{s.t. } \forall 1 \leq l \leq m : x_g^{l-1} \xrightarrow{i_l/o_l} x_g^l \in T_g \\
& \exists \nu = \mu \text{ s.t. } \forall 1 \leq l \leq m : x^{l-1} \xrightarrow{i_l/o_l} x^l \in T^* \\
\Leftrightarrow \quad & \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m \\
& \forall 1 \leq l \leq m : x^{l-1} \xrightarrow{i_l/o_l} x^l \in T^* \\
& \Leftrightarrow \exists k : \left[\begin{array}{l} x^{l-1} \xrightarrow{i_l/o_1} x_2 \in T \wedge \\ x_2 \xrightarrow{i_2/o_2} x_3 \in T \wedge \\ \ldots \\ x_k \xrightarrow{i_k/o_l} x^l \in T \wedge \\ \forall 1 < j \leq k : i_j = o_{j-1} \end{array} \right]
\end{aligned}
$$

From Definitions 6 and 7

$$\Leftrightarrow \quad \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m$$
$$\forall 1 \leq l \leq m : \exists k :$$
$$i_l/o_l \in \mathcal{CL}(i_l/o_1.i_2/o_2 \ldots i_k/o_l)$$
From Definition 3

$$\Leftrightarrow \quad \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m$$
$$\forall 1 \leq l \leq m :$$
$$i_l/o_l \in \mathcal{CL}(C_1(i_l/o_l) \| C_2(i_l/o_l) \ldots \| C_N(i_l/o_l))$$
where $C_1(i_l/o_l)$ is the sequence of transitions from A_1

$$\Leftrightarrow \quad \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m$$
$$\forall 1 \leq l \leq m : i_l/o_l \in \mathcal{CL}(\|_{n \leq N} C_n(i_l/o_l))$$

It can be easily seen from Definitions 6 and 7

$$i_l/o_l \in \mathcal{CL}(\|_{n \leq N} C_n(i_l/o_l)) \wedge$$
$$i_l/o_l.i_{l+1}/o_{l+1} \in \mathcal{CL}(\|_{n \leq N} C_n(i_l/o_l.i_{l+1}/o_{l+1})) \tag{3.1}$$
$$\Rightarrow \quad \forall n \leq N : C_n(i_l/o_l) \preceq C_n(i_l/o_l.i_{l+1}/o_{l+1})$$

Proceeding further,

$$A_G \sqsubseteq \mathtt{U} \quad \Leftrightarrow \quad \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m$$
$$\mu \in \mathcal{CL}(\|_{n \leq N} C_n(\mu))$$
$$\Leftrightarrow \quad \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m$$
$$\mu \text{ is realizable from } \{C_n(\mu)\}$$
From Definition 4
$$\Leftrightarrow \quad \exists C : (I_g \times O_g)^* \rightarrow \prod_{n \leq N} (I_n \times O_n)^*$$
From Equation 3.1 and Definition 5

■

Example 7 *Note that the goal in Figure 3.3 is not simulated by* `U`*-automaton and therefore, there exists no choreographer which can realize A_G from A_1 and A_2. If the requirement of a goal was to generate an* `Appr` *output for* `SSN` *input, it can be simulated by* `U`*-automaton in Figure 3.4(b).*

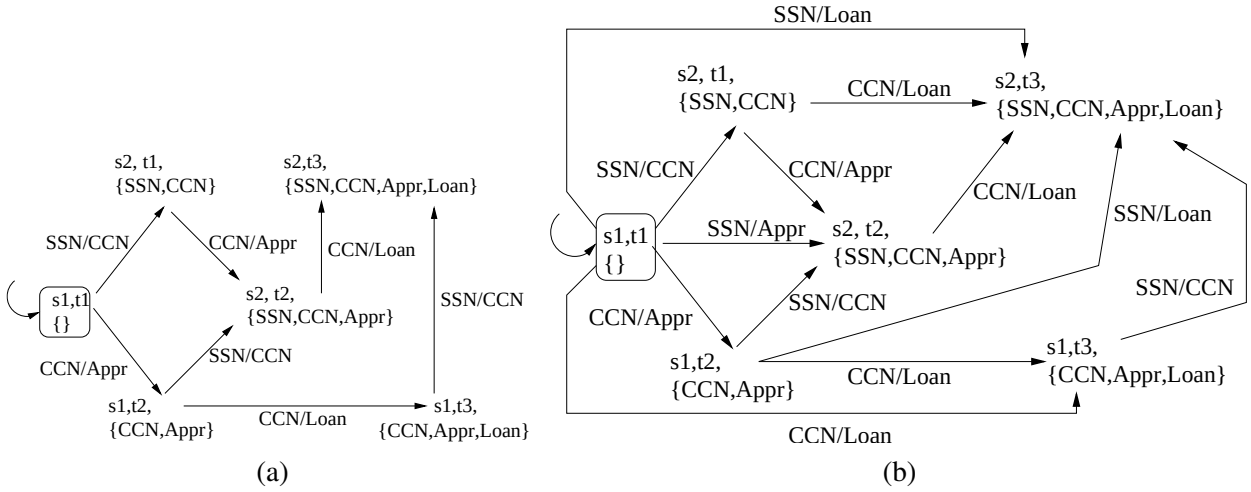


Figure 3.5: (a) $(\|_n^H A_n)$-automaton and (b) $\cup^T$-automaton.

3.4 Transducing Choreographer

In the previous section, we considered a choreographer which is capable of relaying output from one service to the input of another. However, it is not capable of storing the inputs and outputs. The stored information can be used to provide inputs to services at any time. In this section, we consider the choreographer that is capable of such functionality. The added functionality allows choreographer to perform *inducing* and *hiding* of actions. Inducing refers to the case where the choreographer uses some stored information to supply required input to a service while hiding refers to the case where the choreographer absorbs outputs from the services and does not provide it to the client (hidden from the client).

We will refer to such a choreographer as *transducing* choreographer. To identify the behavior that can be realized using a *transducing* choreographer, we will extend the definition of closure of a sequence (see Definition 3) which in turn will enhance the definition of realizability (Definition 4) and empower the choreographer.

Note that, sequence now has a context consisting of the *history* that led a choreographer to see this sequence. The history of a sequence keeps track of the set of inputs and outputs that can be used by the choreographer to transduce the said sequence.

Definition 9 (Transduced-Closure) *Given a sequence* $\alpha = i_1/o_1.i_2/o_2 \ldots i_n/o_n \in (I \times O)^*$*, its transduced closure, denoted by* $\mathcal{CL}^T(\alpha)$*, is equal to*

$$\left\{ \begin{array}{l} i_1/o_1 \ldots i_{m-1}/o_{m-1}.i_m/o_k.i_{k+1}/o_{k+1} \ldots i_n/o_n \mid \\ \quad \exists m < k : \forall j, m \leq j < k : i_{j+1} \in \{i_l, o_l \mid l \leq j\} \end{array} \right\}$$

For example for $\mathcal{CL}^{\mathrm{T}}(a/b.a/c) = \{a/b.a/c,\ a/c\}$. Using the notion of transduced-closure, we next define the notion of transduced-realizability.

Definition 10 (Transduced-Realizability) $\alpha \in (I \times O)^*$ *is said to be transducively-realizable from* $\{\beta_n \mid \beta_n \in (I_n \times O_n)^*\}$ *if* $\exists \mu \in \|_{n \leq N}\ \beta_n : \alpha \in \mathcal{CL}^T(\mu)$.

A transducing choreographer that uses inducing and hiding (w.r.t. inputs and outputs seen in past) besides "chaining" is defined as follows.

Definition 11 (Transducing-Choreographer) *Given a goal automaton* $G = (X_g, X_g^0, I_g, O_g, T_g)$ *and* N *service automaton of the form* $A_n = (X_n, X_n^0, I_n, O_n, T_n)$ *for* $n \leq N$ *such that* $I_g \subseteq \bigcup_n I_n$ *and* $O_g \subseteq \bigcup_n O_n$*, a transducing-choreographer is function of the form* $C^T : (I_g \times O_g)^* \rightarrow \prod_{n \leq N} (I_n \times O_n)^*$ *satisfying*

1. $\forall \mu, \nu \in (I_g \times O_g)^* : \mu \preceq \nu \Rightarrow \forall n \leq N : C_n^T(\mu) \preceq C_n^T(\nu)$
2. $\forall \mu \in (I_g \times O_g)^* : \mu$ *is transducively-realizable from* $\{C_n^T(\mu)\}$

Transducing choreographer is more powerful than the simple choreographer as the former is capable of choreographing more functionality (see Definition 9) from the existing services than the later.

Proceeding in a fashion similar to the one followed in Section 3.3, we first define an interleaving product with history, which maintains a history set of the seen inputs and outputs with each state.

Definition 12 (Interleaving Product with History) *Given a set of service automata* $A_1, A_2, \ldots, A_N$ *where* $A_n = (X_n, X_n^0, I_n, O_n, T_n)$ *for* $1 \leq n \leq N$*, the interleaving product with history is defined to be the automaton* $\|_n^H A_n = (X_H, X_H^0, I, O, T_H)$ *where* $X_H \subseteq X_1 \times X_2 \times \ldots \times X_N \times 2^{I \cup O}$ *where*

$I = \bigcup_{n \leq N} I$ *and* $O = \bigcup_{n \leq N} O$; $X_H^0 = X_1^0 \times X_2^0 \times \ldots \times X_N^0 \times \{\emptyset\}$ *and* $T_H \subseteq X_H \times I \times O \times X_H$ *such that*

$$(x_1, \ldots, x_N, h) \xrightarrow{i/o} (x'_1, \ldots, x'_N, h') \in T_H \Leftrightarrow$$
$$\left[\begin{array}{l} \exists i \leq N : x_i \xrightarrow{i/o} x'_i \in T_i \wedge \\ \forall j \leq N, j \neq i : x'_j = x_j \ \wedge h' = h \cup \{i, o\} \end{array}\right]$$

The $(\|_n^H A_n)$-automaton is similar to $(\|_n A_n)$-automaton with the exception that the history of i/o are tracked at each state (so these can be used for internal inducing even when there is no external input).

Example 8 *Figure 3.5(a) shows the* $(\|_n^H A_n)$*-automaton generated from* A_1 *and* A_2 *in Figures 3.2(a,b). Every state is labeled by the states of* A_1 *and* A_2 *and also shows that inputs and outputs that a transducing choreographer can use. For example at the start state* $(1, 1)$*, the choreographer does not have any stored information as it has no history of interaction while if the services are at state* $(2, 2)$*, the choreographer contains the information on* `SSN`*,* `CCN`*,* `Appr` *obtained from its prior interactions.*

The $(\|_n^H A_n)$-automaton represents the information available to a transducing choreographer. To identify the sequences that can be choreographed by chaining along with inducing and hiding the transitions available in the $(\|_n^H A_n)$-automaton, we define the operation of transduced-closure.

Definition 13 (Transduced-Closure of Automaton) *Given an automaton with history* $A_H = (X_H, X_H^0, I, O, T_H)$*, the transduced-closure of* A_H *is the automaton* $\mho^T = (X_H, X_H^0, I, O, T_H^*)$ *where* $(x, h) \xrightarrow{i/o} (x_k, h_k) \in T_H^*$ *if and only if*

$$\exists k : \left[\begin{array}{l} (x, h) \xrightarrow{i/o_1} (x_1, h_1) \in T_H \wedge \\ (x_1, h_1) \xrightarrow{i_1/o_2} (x_2, h_2) \in T_H \wedge \\ \ldots \\ (x_{k-1}, h_{k-1}) \xrightarrow{i_{k-1}/o} (x_k, h_k) \in T_H \\ \wedge \forall 1 \leq j < k : i_j \in h_j \end{array}\right] \tag{3.2}$$

Example 9 *Consider the* $\mho^T$*-automaton (Figure 3.5(b)) obtained from the* $(\|_n^H A_n)$*-automaton in Figure 3.5(a).* $\mho^T$*-automaton contains transitions that can be realizable from using the history information*

Figure 3.6: Transducing choreographer.

at each state. The newly added transitions are labeled with $[i/o]$*. State* $(1,1)$ *has a transition to state* $(1,3)$ *on* `[CCN/Loan]`*. The transition is obtained from the transitions* $(1,1) \xrightarrow{\texttt{CCN/Appr}} (1,2)$ *and* $(1,2) \xrightarrow{\texttt{CCN/Loan}} (1,3)$*. After the first transition, both* `CCN` *and* `Appr` *are available for future input at state* $(1,2)$*. The second transition from* $(1,2)$ *therefore does not rely on the environment to provide its enabling input; the input can be provided by a transducing choreographer by using the* `CCN` *from the information stored at* $(1,2)$*. The second transition produces the output* `Loan`*. Similar transition is added from* $(2,1)$ *to* $(2,3)$ *on* `[CCN/Loan]`*, following which* $(1,1) \xrightarrow{[SSN/Loan]} (2,3)$ *is also added to* $\mathcal{U}^T$*-automaton.*

Theorem 2 *Given a goal* A_G *and a set of services* $A_1, A_2, \ldots, A_N$*, the goal can be realizable from the composition of* A_n*s with a transducing choreographer if and only if* $A_G \sqsubseteq \mathcal{U}^T$ *where* $\mathcal{U}^T$ *is the transduced-closure of the* $(\|_n^H A_n)$*-automaton obtained by taking interleaving product with history of the automata* $\{A_n\}$*s.*

Proof: Let $A_G = (X_g, X_g^0, I_g, O_g, T_g)$ and $\mathcal{U}^T = (X_H, X_H^0, I, O, T_H^*)$.

$$\begin{aligned} A_G \sqsubseteq \mathcal{U}^T \quad &\Leftrightarrow \quad \forall x_g^0 \in X_g^0 : \exists (x^0, h^0) \in X_H^0 : x_g^0 \sqsubseteq x^0, h^0 \\ &\Leftrightarrow \quad \left(\begin{array}{l} \forall x_g^1 : x_g^0 \xrightarrow{i_1/o_1} x_g^1 \in T_g \Rightarrow \\ \exists x^1 : x^0 \xrightarrow{i_1/o_1} x^1 \in T^* \wedge x_g^1 \sqsubseteq x^1 \end{array} \right) \end{aligned}$$

Therefore,

$$
\begin{aligned}
A_G \sqsubseteq \mho^T \quad \Leftrightarrow \quad & \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m \\
& \text{s.t. } \forall 1 \leq l \leq m : x_g^{l-1} \xrightarrow{i_l/o_l} x_g^l \in T_g \\
& \exists \nu = \mu \text{ s.t. } \forall 1 \leq l \leq m : \\
& \qquad (x^{l-1}, h^{l-1}) \xrightarrow{i_l/o_l} (x^l, h^l) \in T_H^* \\
\Leftrightarrow \quad & \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m \\
& \forall 1 \leq l \leq m : (x^{l-1}, h^{l-1}) \xrightarrow{i_l/o_l} (x^l, h^l) \in T_H^* \\
\Leftrightarrow \exists k : & \left[\begin{array}{l} (x^{l-1}, h^{l-1}) \xrightarrow{i_l/o_1} (x_1, h_1) \in T_H \wedge \\ (x_1, h_1) \xrightarrow{i_1/o_2} (x_2, h_2) \in T_H \wedge \\ \ldots \\ (x_{k-1}, h_{k-1}) \xrightarrow{i_{k-1}/o_l} (x^l, h^l) \in T_H \\ \wedge \forall 1 \leq j < k : i_j \in h_j \end{array} \right] \\
& \text{From Definitions 12 and 13} \\
\Leftrightarrow \quad & \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m \\
& \forall 1 \leq l \leq m : \exists k : \\
& \qquad i_l/o_l \in \mathcal{CL}^{\mathrm{T}}(i_l/o_1.i_2/o_2 \ldots i_k/o_l) \\
& \text{From Definition 9} \\
\Leftrightarrow \quad & \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m \\
& \forall 1 \leq l \leq m : \\
& i_l/o_l \in \mathcal{CL}^{\mathrm{T}}(C_1^T(i_l/o_l) \| \ldots \| C_N^T(i_l/o_l)) \\
& \text{where } C_1^T(i_l/o_l) \text{ is the sequence of} \\
& \text{transitions from } A_1 \\
\Leftrightarrow \quad & \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m \\
& \forall 1 \leq l \leq m : i_l/o_l \in \mathcal{CL}^{\mathrm{T}}(\|_{n \leq N} C_n^T(i_l/o_l))
\end{aligned}
$$

It can be easily seen from Definitions 12 and 13

$$\begin{aligned} & i_l/o_l \in \mathcal{CL}^{\mathrm{T}}(\|_{n\leq N} C_n^T(i_l/o_l)) \wedge \\ & i_l/o_l.i_{l+1}/o_{l+1} \in \mathcal{CL}^{\mathrm{T}}(\|_{n\leq N} C_n^T(i_l/o_l.i_{l+1}/o_{l+1})) \\ \Rightarrow \quad & \forall n \leq N : C_n^T(i_l/o_l) \preceq C_n^T(i_l/o_l.i_{l+1}/o_{l+1}) \end{aligned} \tag{3.3}$$

Proceeding further,

$$\begin{aligned} A_G \sqsubseteq \mathtt{U}^T \quad \Leftrightarrow \quad & \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m \\ & \qquad \mu \in \mathcal{CL}^{\mathrm{T}}(\|_{n\leq N} C_n^T(\mu)) \\ \Leftrightarrow \quad & \forall \mu = i_1/o_1.i_2/o_2 \ldots i_m/o_m \\ & \qquad \mu \text{ is transducively-realizable from } \{C_n^T(\mu)\} \\ & \qquad \text{From Definition 10} \\ \Leftrightarrow \quad & \exists C^T : (I_g \times O_g)^* \rightarrow \prod_{n\leq N} (I_n \times O_n)^* \\ & \qquad \text{From Equation 3.3 and Definition 11} \end{aligned}$$

■

Example 10 *For the Definition 11, if each goal sequence is executable in the $\mathtt{U}^T$-automaton, then it is inferred that the goal is realizable from the existing services using a transducing choreographer. This is verified using the simulation relation (Definition 8). In the current example the $\mathtt{U}^T$ automaton in Figure 3.5(b) simulates the goal in Figure 3.3. Therefore, the goal can be realizable from the existing service automata A_1 and A_2 (Figure 3.2) by using a transducing choreographer. The choreographer first relays the `SSN` information from the client to A_1 and then uses the `CCN` output from A_1 as input to A_2 twice to get the appropriate output `Loan` which is then relayed to the client. The transducing choreographer is shown in Figure 5.15.*

Once it is verified that the goal service is simulated by the universal service automaton ($\mathtt{U}$ or $\mathtt{U}^T$ as the case may be), a choreographer can be synthesized by first identifying for each goal transition a corresponding simulating transition in the universal service automaton, and next identifying the set of service transition sequences that realize it. (Note the information about the set of service transition sequences realizing a transition of the universal service automaton is embedded in its definition.) The text of this chapter forms the basis of [14].

CHAPTER 4. Centralized Choreography: On-the-fly Computation

4.1 Preliminaries on Logic Programming

XSB [1], developed at SUNY, Stony Brook is based on Prolong-style SLD resolution with tabling. Tabling, which is essentially a memorizing technique, allows XSB to compute least model solutions of normal logic programs and avoid repeated subcomputations. XSB relations/predicates are defined as

```
R :- R1, R2, ..., Rn. S :- S1; S2; ...; Sun.
```

where relation `R` evaluates to true if *all* (conjunction) the relations `R1`, `R2`, . . ., `Rn` evaluates to true. On the other hand, relation `S` evaluates to true if *at least* one of (disjunction) the relation `S1`, `S2`, . . ., `Sun` evaluates to true. A relation with no right hand side(rhs) of `:-` is referred to as fact.

Consider a simple encoding of a graph in XSB. The edges in the graph are defined as `edge`-facts of the form as shown in Figure 4.1(a). The reachability between states in the above graph can be encoded in XSB as `reach`-relations (Figure 4.1(a)). The predicate `reach` is defined using two rules. The first rule states that a state is reachable from another if there is an edge between the two. Observe that, `S` and `T` are variables[1] which are existentially quantified. The second rule computes the transitive closure: a state `T` is reachable from `S` if there is an edge from `S` to `S1` and `T` is reachable from `S1`.

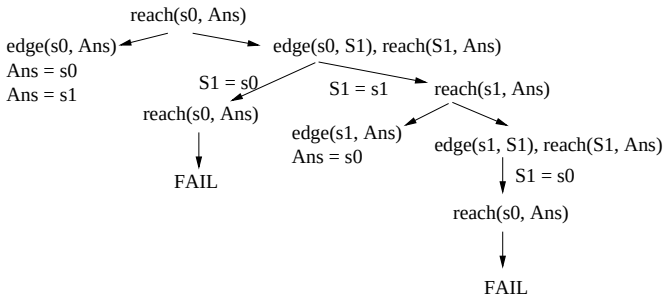
If we want to find all the reachable states from `s0`, we evaluate the query `reach(s0, Ans)` using the above program; on termination, the variable `Ans` evaluates (bounded) to state reachable from `s0`. Figure 4.1(b) presents the execution tree of the above query. Using the first rule, `reach(s0,s0)` and `reach(s0, s1)` evaluates to true. However, evaluation of the second rule fails to terminate because `reach(s0, Ans)` invokes `edge(s0,s0)` followed by `reach(s0, Ans)`.

[1] Upper-case alphabets denote logical variables and lower case alphabets denote constants.

```
edge(s0, s0). edge(s0, s1). edge(s1, s0). edge(s2, s1). edge(s3,
s2).

reach(S, T) :- edge(S, T). reach(S, T) :- edge(S, S1), reach(S1, T).
```

(a) `reach` program in XSB



(b) Evaluation tree for `reach(s0, Ans)`

Figure 4.1: Reachability in XSB

Tabling in XSB. This can be avoided by using tabling feature of XSB and using the directive `:- table reach/2`. The directive ensures that the least model of the relation `reach` is computed. In the current context, `reach(s0,Ans)` will fail if it leads to invocation of `reach(s0, Ans)`. Whenever a failure occurs along an execution path, XSB backtracks automatically to the last choice point to evaluate and execute along an alternate path if possible. In Figure 4.1(b), `reach(s0, Ans)` leads to `edge(s0, S1), reach(S1, Ans)` which grounds `S1` to `s0` as `edge(s0,s0)` is a fact and fails on `reach(s0, Ans)`. It then backtracks and selects another fact from edge: `edge(s0, s1)` and continues with the evaluation of `reach(s1, Ans)`. If there is no choice point available, the predicate execution terminates and returns the evaluation of variable (in this case `Ans`) if there is one; otherwise the predicate is said to be unsatisfiable. The invocation `reach(s0, Ans)` terminates and correctly produces the results for `Ans`. Observe that, states that are not reachable from `s0` are never explored in the above execution, because the execution of any recursive `reach` predicate occurs in a goal-directed fashion for a specific valuation of the first argument (source state).

4.2 Local and On-the-fly Algorithm

4.2.1 Logical Encoding

Encoding I/O Automata. The i/o automaton for a service is described as logic program using facts describing the transitions and the start state. For example, automata A_1 and A_2 from Figure 3.2 are encoded as:

```
servicetrans(s1, (ssn, ccn), s2).
startservice(s1).   %% start state of A1

servicetrans(t1, (ccn, appr), t2). servicetrans(t2, (ccn, loan),
t3).
startservice(t1).   %% start state of A2

startcompose([s1,t1]). %% for interleaving product
```

where `s1` and `s2` are the states in A_1 and `t1`, `t2` and `t3` are states in A_2. The predicate `servicetrans` has three arguments; the first argument denotes source state of the transition, the second argument is a tuple denoting the input and the output associated with the transition and the last argument denotes destination state of the transition. Similarly, the goal i/o automaton (Figure 3.3) is encoded as:

```
goaltrans(g1, (ssn, loan), g2). startgoal(g1).
```

Encoding Interleaving Product. The interleaving product of the services (Definition 12) is encoded as XSB relations:

```
partrans(([St|Sts], Hist), (I, O),
                        ([NewSt|Sts], NewHist)) :-
    servicetrans(St, (I, O), NewSt),
    append([I, O], Hist, NewHist).

partrans( ([St|Sts], Hist), (I, O),
                ( [St|NewSts], NewHist) ) :-
    partrans( (Sts, Hist), (I, O), (NewSts, NewHist)).
```

There are two rules defining the relation `partrans`. First, consider the Rule 1 and the first argument of `partrans`. The first argument denotes tuple: list of service states and the associated history. List in XSB is represented as `[St|Sts]` where `St` represents the first element (head) of the list and `Sts` represents the rest (tail) of the list. Therefore Rule 1 states that there exists a transition in the interleaving product if the service-state present at the head of the list has a transition and the rest of service-states remain unaltered. Furthermore, the history set `Hist` is updated to `NewHist` to include `I` and `O` using the `append` predicate. Rule 2, on the other hand, states that there exists a transition in the interleaving product if any service-state in the tail of the list has a transition. Note that, we just encoded the rules for transition relation of the interleaving product and do not actually generate the product graph.

Encoding Transduced Closure. The transduced closure transition relation (Definition 22) is similarly encoded as follows:

```
closuretrans((Sts, Hist), (I, O),
                          (NewSts, NewHist)) :-
   partrans((Sts, Hist), (I, O), (NewSts, NewHist)).

closuretrans((Sts, Hist), (I, O),
                          (NewSts, NewHist)) :-
   partrans((Sts, Hist), (I, _), (StsTemp, HistTemp)),
   histtrans((StsTemp, HistTemp), (_, O),
                                  (NewSts, NewHist)).
```

The first rule states that there is a transduced closure transition if there is an interleaved product transition. The second rule states that a transduced closure transition exists if there is an interleaved product transition followed by a sequence of transitions (computed by `histtrans` described below) where inputs are provided from the history set `HistTemp`. The notation "_" denotes variable whose valuation is not captured in the evaluation. Observe that, `I` in `closuretrans` is obtained from `partrans` and `O` is obtained from `histtrans`. The rule follows directly the Definition 22 where `histtrans` is computing $\exists k : (s_1, h_1) \xrightarrow{i_1/o_2} (s_2, h_2) \xrightarrow{i_2,o_3} \ldots \xrightarrow{i_{k-1},o} (s_k, h_k) \wedge \forall 1 \leq j \leq k : i_j \in h_j$ (Equation 3.2). The definition is explained in details below:

```
histtrans((Sts, Hist), (I, O), (NewSts, NewHist)) :-
   partrans((Sts, Hist), (I, O), (NewSts, NewHist)),
   member(I, Hist).

histtrans((Sts, Hist), (I, O), (NewSts, NewHist)) :-
   partrans((Sts, Hist), (I, _), (StsTemp, HistTemp)),
   member(I, Hist),
   histtrans((StsTemp, HistTemp), (_, O),
                                  (NewSts, NewHist)).
```

The rules are similar to `closuretrans`. The first rule corresponds to the base case: $(s_{k-1}, h_{k-1}) \xrightarrow{i_{k-1},o} (s_k, h_k)$ while the second rule corresponds to the transitivity. In each rule, the inputs of `partrans` are provided by the `Hist` (predicate `member(I, Hist)` evaluates to true if `I` is present in `Hist`).

Encoding Simulation Relation. The final step of the choreographer existence problem is to verify simulation of the goal automaton by the transduced closure automaton. Recall from Definition 8, two states are said to be simulation equivalent if and only if they are related by the largest simulation relation $\sqsubseteq$: $s_1 \sqsubseteq s_2 \;\Rightarrow\; [\forall t_1 : s_1 \xrightarrow{i/o} t_1 \Rightarrow (\exists t_2 : s_2 \xrightarrow{i/o} t_2 \;\wedge\; t_1 \sqsubseteq t_2)\,]$. However, evaluation of logic program results in the least model computation. In order to encode simulation as logic program, we consider the negation of the above relation, i.e., two states are said to be not simulation equivalent if and only if they are related by the least not-simulation relation $\not\sqsubseteq$ (dual of $\sqsubseteq$) defined as follows:

$$s_1 \not\sqsubseteq s_2 \;\Leftarrow\; [\exists t_1 : s_1 \xrightarrow{i/o} t_1 \;\wedge\; (\forall t_2 : s_2 \xrightarrow{i/o} t_2 \;\Rightarrow\; t_1 \not\sqsubseteq t_2)] \tag{4.1}$$

Therefore, from Theorem 3 a goal A_G is *not realizable* from choreographed $\mathtt{U}^T$ if and only if $A_G \not\sqsubseteq \mathtt{U}^T$. The above equation can be directly encoded in XSB as:

```
:- table nsim/2. nsim(S1, S2) :-
   goaltrans(S1, A, T1),
   forall(closuretrans(S2, A, T2), nsim(T1, T2)).

forall(closuretrans(S2, A, T2), nsim(T1, T2)) :-
    closuretrans(S2, A, T2),
    (nsim(T1, T2) -> fail; !, fail).
forall(_, _).
```

The predicate `nsim(S1,S2)` corresponds to `S1` $\not\sqsubseteq$ `S2` relation. It evaluates to true when there exists a transition in the goal A_G from `S1` which is not simulated by any transition (denoted by `closuretrans`) in $\mathtt{U}^T$ from `S2`. Observe that, the predicate is "tabled" (Section 4.1) which ensures that evaluation of `nsim` corresponds to the least solution for $\not\sqsubseteq$ relation (Equation 4.1).

The predicate `forall` corresponds to $\forall t_2 : s_2 \xrightarrow{i/o} t_2 \Rightarrow t_1 \not\sqsubseteq t_2$ in Equation 4.1. The evaluation of `forall` can be explained as follows. It identifies a matching `closuretrans` (on same i/o as the goal transition) and verifies whether `nsim(T1, T2)` holds true. If `nsim(T1, T2)` evaluates to true, it *fails* and backtracks to obtain another result for `closuretrans`; otherwise it fails and does not backtrack—backtracking is terminated using the operator "!", referred to as *cut*. If `closuretrans` evaluates to false, then predicate `forall` succeeds via the second rule, in which case the instance of `nsim` that invoked the `forall` also succeeds.

Given the service and goal automata, the existence of choreographer is verified by invoking `nsim(GoalState, (ServiceStates, []))` where `startgoal(GoalState)` and `statecompose(ServiceStates)` evaluates to true. The empty list `[]` associated with `ServiceStates` denotes that the history-set is empty initially. If `nsim(GoalState, (ServiceStates, []))` evaluates to true then the choreographer does not exist; otherwise it does.

Once the existence of a choreographer is established, we analyze the transitions explored to prove the existence and the states in the $\mathtt{U}^T$-automaton that are similar to states in goal automaton to synthesize the choreographer. This ensures that synthesis of choreographer also explores only the transitions that are required to prove its existence.

4.2.2 Example Run

In the example, in order to verify the existence of a choreographer, we invoke `nsim(g1, ([s1,t1],[]))` where `g1` is the start state of the goal and `([s1,t1],[])` is the start state of $\mathtt{U}^T$-automaton corresponding to the services. The execution trace is presented in Figure 4.2. Each step shows the expansion of a predicate by the rhs of its rule. Some expansions are shown with grounding of variable when a predicate is satisfied. For example `nsim`, at the root, is expanded to conjunction of predicates `goaltrans` and `forall`. The predicate `goaltrans` evaluates to true and grounds the valuation of the variable `A` to

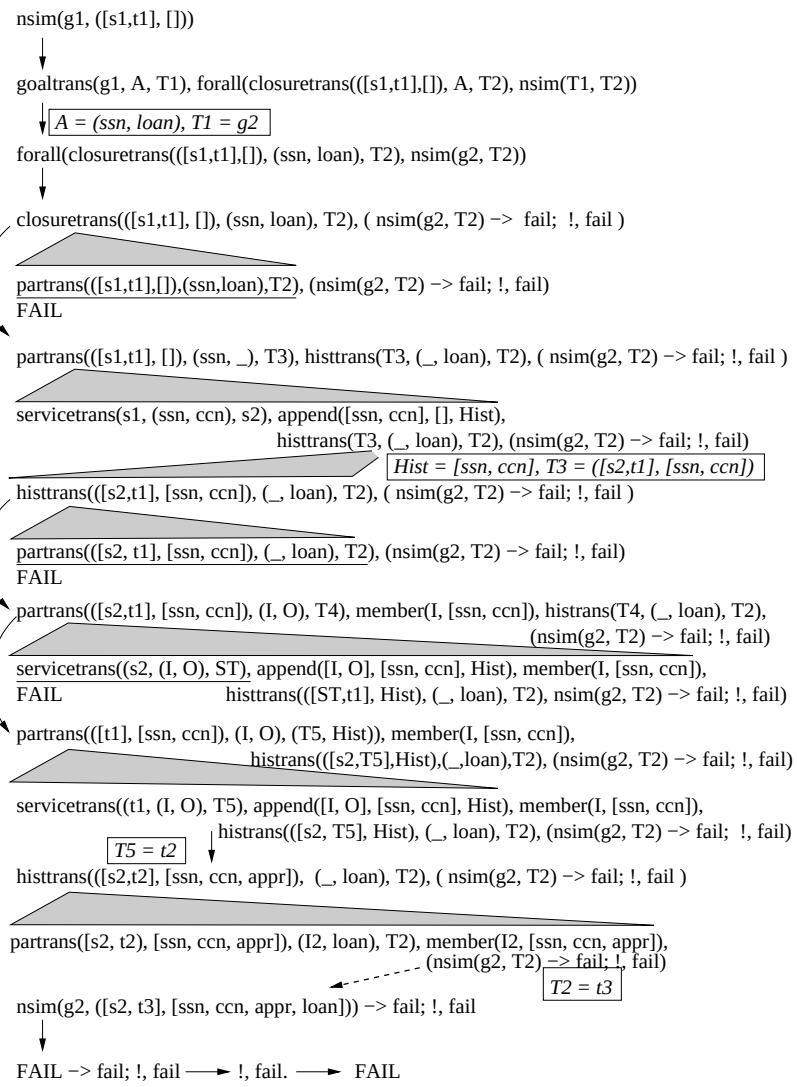


Figure 4.2: Execution tree.

`(ssn,loan)` and `T1` to `g2` (corresponding to the goal transition). Whenever a failure is encountered, backtracking is performed from the last choice point. For example, first invocation of `closuretrans` fails due to failure to satisfy `partrans`—there is no transition defined from `((s1, t1), [])` on input `ssn` and output `loan`. As a result, prolog engine backtracks to obtain an alternate way to satisfy `closuretrans` via the second rule of `closuretrans`. Finally, a query to predicate `nsim(g2, ([s2, t2], [ssn,ccn,appr,loan]))` is made which fails as there is goal-transition from `g2`. This leads the execution to evaluate "`!, fail`", i.e., go over a cut and as mentioned above backtracking is stopped. This implies that for a goal transition sequence, a sequence of transitions in the $\mathtt{U}^T$-automaton is ob-

tained that simulates the goal transition sequence. his leads to As a result, `nsim(g1, ([s1,t1],[]))` fails, i.e., `g1` is simulated by `([s1,t1], [])` which proves that a choreographer exists.

Observe that, 3 transitions in $\mathtt{U}^T$-automaton are explored

$$(s_1, t_1)\emptyset] \xrightarrow{SSN/CCN} [(s_2, t_1)\{SSN, CCN\},$$
$$(s_2, t_1)\{SSN, CCN\} \xrightarrow{CCN/Appr} (s_2, t_2)\{SSN, CCN, Appr\}$$
$$(s_2, t_2)\{SSN, CCN, Appr\} \xrightarrow{CCN/Loan} (s_2, t_3)\{SSN, CCN, Appr\}$$

instead of the entire automaton, because the evaluation proceeds in a goal-directed fashion and does not explore transitions that are not triggered by `SSN` as input from the start state of the $\mathtt{U}^T$-automaton.

4.3 Experimental Results

We evaluate the technique using common use cases. The first example is an travel reservation service, modified from [4], where the aim is to develop a composite service (a) for reserving transportation (car-rental) from source address to the origin airport, from destination airport to destination hotel and (b) for buying the airline tickets and (c) for booking accommodation (hotel). The composition is realized from pre-existing services one each for car-rental, air-ticket purchase and hotel reservation. Note that the car-rental service is used twice in the example, once to reserve car from source address to the source airport and then to reserve car from destination airport to the hotel. The second example involves ordering, buying and shipping items, simplified from [19]. The choreographer in this case uses individual services to do credit check, item availability check and shipping. Finally, we have also experimented with a loan approval service where the goal is approval for a specific loan amount given the credit information of the user. Once the credit check is validated, approval of specific amount may depend on approval from several different branches of loan office. We increased the number of states and transitions in the goal as well as the number of services by varying the number of branches that need to approve a loan.

For each of the above examples, Table 4.1 presents the size of the goal, number of existing services involved in realizing the goal, size of the $\mathtt{U}^T$-automaton, number of transitions explored by our local and on-the-fly algorithm and the size of the choreographer generated. The table shows that the number of states and transitions generated and explored is much less than the total number of states and tran-

Name	Goal Size		No. of	$\mathtt{U}^T$ size		Transitions	Choreographer	
	States	Trans.	Services	States	Trans	Gen. & Expl.	States	Trans
Travel Reservation	9	8	4	400	9659	28	15	14
Purchase & Ship	4	3	3	32	183	8	8	7
Loan 1	7	6	4	48	610	6	7	6
Loan 2	9	8	5	144	4548	8	8	8
Loan 3	11	10	6	432	26712	10	9	10

Table 4.1: Experimental Results.

sitions in the $\mathtt{U}^T$-automaton providing strong testimony that our local and on-the-fly algorithm can be effectively applied in practical setting. This chapter forms the basis for [15] as also [18].

CHAPTER 5. Optimum Decentralization: Formulation and Synthesis

As mentioned in the previous chapters, Web Services composition, that composes existing services to realize a target service known as the *goal* service has followed two kinds of methodology. One which depends on a centralized mediator (often referred to as the choreographer) to realize a goal service. In this chapter, we explore the other approach which is decentralized choreography where mediators are placed physically close to the services as well as the client for optimality of performance (of computation and communication costs) [39, 40].

There is a cost associated with using the service, as well as a cost for communicating messages to and from the services; the first being the computation cost and the second the communication cost. The cost is contextual and may be represented in terms of monetary value, response time or network usage. For example, a bookstore service can charge a fee for using their search service. This involves searching in their database and producing the result. Overall cost for realizing a goal service can be computed by assigning appropriate weights to represent the computation and the communication costs.

Illustrative Example. We present a brief overview of the problem along with our solution mechanism using the following running example obtained from [40]. There are three existing services (Figure 5.1(a)) `S1`, `S2` and `S3`. `S1` takes as input name of a person (`nP`) and provides his/her address (`aP`). Service `S2` can perform all input/output operations of `S1`. In addition, it can take as input name of a point of interest or business (`nC`) and output its address (`aC`). And finally, `S3` takes as input two addresses and computes the route (`r`) between them. The developer wants to create a new service, `G`, for a client which takes (`nP`) and (`nC`) as inputs and provides the route between them (Figure 5.1(b)). The existing centralized solutions will develop a single choreographer; one such choreographer `C0` is presented in Figure 5.1(c). Observe that the choreographer acts as an intermediary between all the services and the

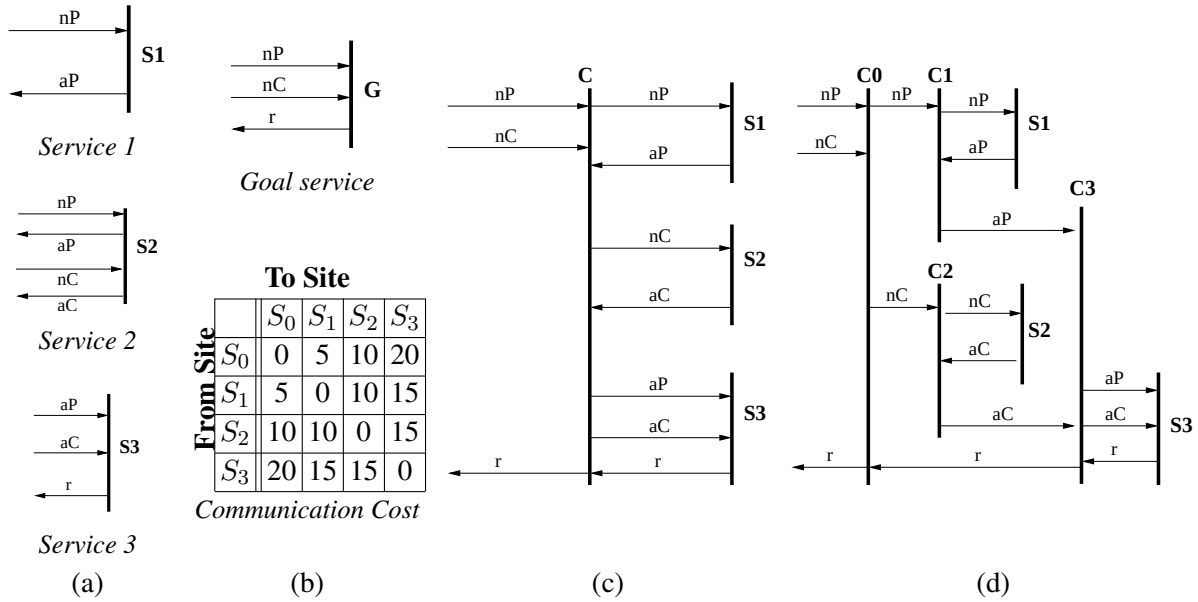


Figure 5.1: (a) Existing Services, (b) goal service & cost metrics, (c) centralized and (d) decentralized choreography

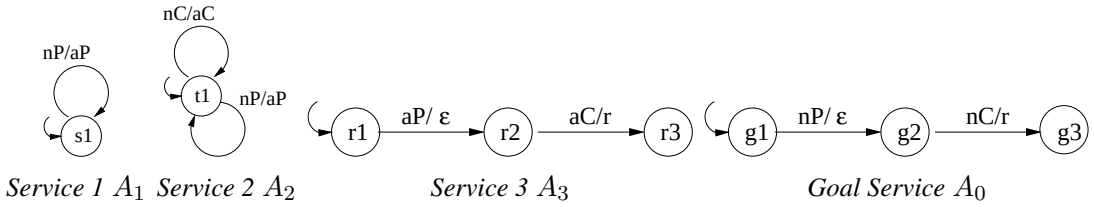


Figure 5.2: i/o-automata representation

client. This bottleneck may cause degradation of performance due to large delay between the request and response from and to the client. It may also cause additional traffic in terms of exchanged messages thereby consuming available bandwidth unnecessarily and possibly delaying other network intensive operations.

Now consider that the cost of communication (in terms of network traffic, bandwidth etc.) between the client and services and between any services are as in the table of Figure 5.1(b). In the table we refer to the client site as `s0`. It is desirable to develop a mechanism such that all the messages are not required to be relayed via a centralized choreographer such as `C0`. Instead multiple choreographers are deployed at servers which are at close proximity (physically or in terms of request/response delay)

to the existing services. The objective is to implement the goal service with minimum possible cost (following the cost table) where message exchanges do not necessarily require relaying through C0.

A possible solution for the current example is presented in Figure 5.1(d) where in addition to C0, three choreographers C1, C2, and C3 are deployed near the existing service sites. Observe that, the messages (aP and aC) are not required by the client and as such they are not sent to C0. Also, observe that service S2 is not used to obtain the aP from input nP. This is because the cost of communication between the client and S2 and between S2 and S3 is more than the corresponding communication cost for service S1. In short, four choreographers are used to minimize the cost of exchanging input/output messages between the services and the client.

Objective. The objective of designing an *optimal decentralization choreography scheme* requires designing the behaviors of each site-specific choreographer such that an overall cost of choreography (as measured using communication and computation costs) is minimized. Note some site-specific choreographers may not be needed and their behaviors may turn out to be empty.

Proposed solution. The solution proposed in this paper has two steps. In the first step, we identify whether there exists a choreographed composition of existing services such that the given goal service, represented as i/o-automata (Section 5.4), can be realized. This is solved by computing the universal (all possible) choreographed behaviors (via interleaving and transduced-closure) that can be realized from the existing services and verifying (via simulation) whether the goal behavior can be subsumed by the universal behaviors (Section 5.5). The next step (Section 5.6) is to synthesize the choreographer at each service site such that an overall communication and computation cost of realizing the goal service is minimized. For this, the transitions in the universal composite behaviors of the existing services, that *subsume* the goal behavior, is annotated with their cost valuations. For example the cost of the composite behaviors as presented in Figure 5.1(c) is equal to the sum of the cost of exchanging messages between C0 and S1 twice, between C0 and S2 twice, and between C0 and S3 thrice. We identify the cost of all possible choreographed compositions and select the one which has the minimum cost by way of a backward search (as in dynamic programming).

The **contributions** in this chapter can be summarized as follows. This is a first approach which presents an automated solution to optimum decentralized choreography for Web services composition. Our approach is based on I/O-automata representation of the services and the goal, and identifies appropriate choreography scheme using the notions of universal service (obtained as a transduced-closure of given services), simulation relation and optimal worst-case path-cost computation for a graph. The contributions in this chapter have been published in [16].

Figure 5.2 presents the I/O-automata models for the three services and goal described in Figure 5.1(a, b). The automaton A_i ($i = 1, 2, 3$) corresponds to the ith service and the automaton A_0 corresponds to the goal. The start states of the automata have curved arrows pointed to them.

5.1 Choreographer Existence

The individual services can be used to implement a goal service by realizing each input/output computations of the goal as a sequence of input/output computations of the existing services such that the input (resp., output) of the first (resp., last) computation in the sequence is the same as the input (resp., output) of the goal. In order to execute an input/output computation, input to a service is provided by the services or client which have seen and/or stored the said input. In other words, the input to a service comes from the "history" of information present in another service or the client. To formalize these concepts we define the notion of *interleaving product with distributed history*.

5.1.1 Product with Distributed History

We allow a choreographer to be associated with each service site and the client site. Suppose there are N services located at sites $1, \ldots, N$. The service at site-n ($n \in \{1, \ldots, N\}$) is modeled as an I/O-automaton $A_n = (S_n, S_n^0, I_n, O_n, \Delta_n)$. We designate site-0 as the site interfacing with the client. Associated with each site-specific choreographer is a site-specific history set consisting of the inputs and outputs seen and stored.

A certain input/output transition of the desired goal service, can be implemented by executing a sequence of input/output transitions at various sites. To facilitate the computation of all such sequences we define the notion of *interleaving product with distributed history*. A local history set of a site

consists of all inputs and outputs that it has seen in past. (Elements in this local history set can be used to supply the input of a transition to be executed in future; this will become clear in subsequent discussions.)

In the interleaving product, exactly one site participates in the execution of each transition, and accordingly each transition is tagged with a site-index n, that identifies the participant site. Execution of such a transition at site-n augments its local history by the input-output pair of the transition. The following definition of Interleaving Product With Distributed History ($\|_n^{\vec{H}} A_n$) captures all possible interleaved behaviors of the service automata and the associated distributed histories. Note the distributed history is essential for the synthesis of a decentralized choreographer. Only a centralized history was considered in the earlier chapter. In the following, the notation $\vec{v}(n)$ is used to denote the nth element of a vector/tuple $\vec{v}$.

Definition 14 ($\|_n^{\vec{H}} A_n$ Automaton) *Given service automata* $\{A_n = (S_n, S_n^0, I_n, O_n, \Delta_n) | 1 \leq n \leq N\}$*, their* interleaving product with distributed history *is defined as the I/O-automaton* $\|_n^{\vec{H}} A_n = (\vec{S} \times \vec{H}, \vec{S^0} \times \vec{H^0}, I_0, O_0, \Delta_{\vec{H}})$ *where* $\vec{S} = \prod_{n=1}^{N} S_n$, $\vec{S^0} = \prod_{n=1}^{N} S_n^0$,

$\vec{H} = \prod_{n=1}^{N} 2^{I_n \cup O_n}$, $\vec{H^0} = \prod_{n=1}^{N} \{\emptyset\}$,

$I_0 = \bigcup_{n=1}^{N} I_n, O_0 = \bigcup_{n=1}^{N} O_n$, *and*

$(\vec{s}, \vec{h}) \xrightarrow[n]{i/o} (\vec{s'}, \vec{h'}) \in \Delta_{\vec{H}}$ *if and only if*

$$\vec{s}(n) \xrightarrow{i/o} \vec{s'}(n) \in \Delta_n \ \wedge \ \vec{h'}(n) = \vec{h}(n) \cup \{i, o\} \ \wedge$$
$$\forall m \neq n : \vec{s'}(m) = \vec{s}(m) \wedge \ \vec{h'}(m) = \vec{h}(m).$$

In the definition of $\Delta_{\vec{H}}$, the first conjunct states that whenever a constituent service makes a move, the $\|_n^{\vec{H}} A_n$ automaton also makes a move with the same transition label. The second conjunct states that the local history of the participant service is updated with the input and output transition labels. The third and fourth conjuncts state the facts that other services do not change states or their corresponding local histories. Thus, if a service gets the input i and produces output o, its local history is enriched by $\{i, o\}$. The above definition paves the way for chaining input/output computations which we capture using the transduced-closure operation defined in the next subsection.

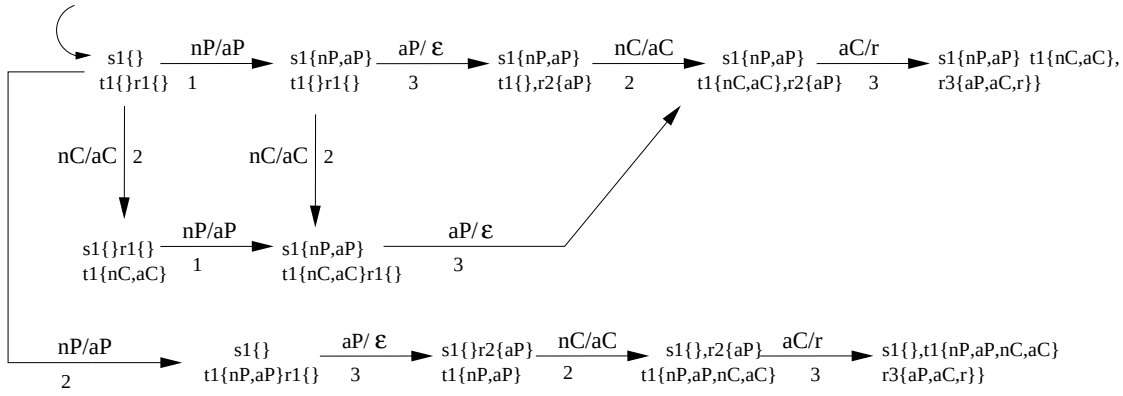


Figure 5.3: Interleaving Product Automaton, $\|_n^{\vec{H}} A_n$

Example 11 *Figure 5.3 depicts a part of the automaton $\|_n^{\vec{H}} A_n$ for A_1, A_2 and A_3 presented in Figure 5.2. Observe that, the history of the start state is empty. Every state is shown with the local history associated with it and transitions are labeled with the participating service responsible for the transition. Thus, $s1\{\}t1\{\}r1\{\}$ changes its configuration to $s1\{\}t1\{nC,aC\}r1\{\}$ as A_2 makes a move.*

Since it is an interleaving product, any of the 3 services can make a move. For example, if A_1 makes a move taking input nP and producing output aP, the destination state represents the change in the configuration of A_1 and captures the updated local history, showing that $\{nP, aP\}$ are stored at site-1 for future computations. Also, the location where the service is invoked, is labeled below the transitions. Note that, the configuration of the service involved in the input and output only changes and the configuration of the other services remain intact during a transition.

5.1.2 Transduced Closure Automaton

The transduced-closure operation is applied on the interleaving product automaton with distributed history to compute the universe of all services that can be implemented by choreographing the existing services. A site can get data from its own local history or from the local history of another site to execute future transitions. There is a cost associated with any such communication of data, and we use $c(n, m) \in \Re_+$, where $\Re_+$ is the set of nonnegative reals, to denote the (cheapest) cost of communicating a data from site-n to site-m. The cost can be any numeric valuation quantifying various aspects of communication; e.g., network traffic, distance between servers, number of hops for each communica-

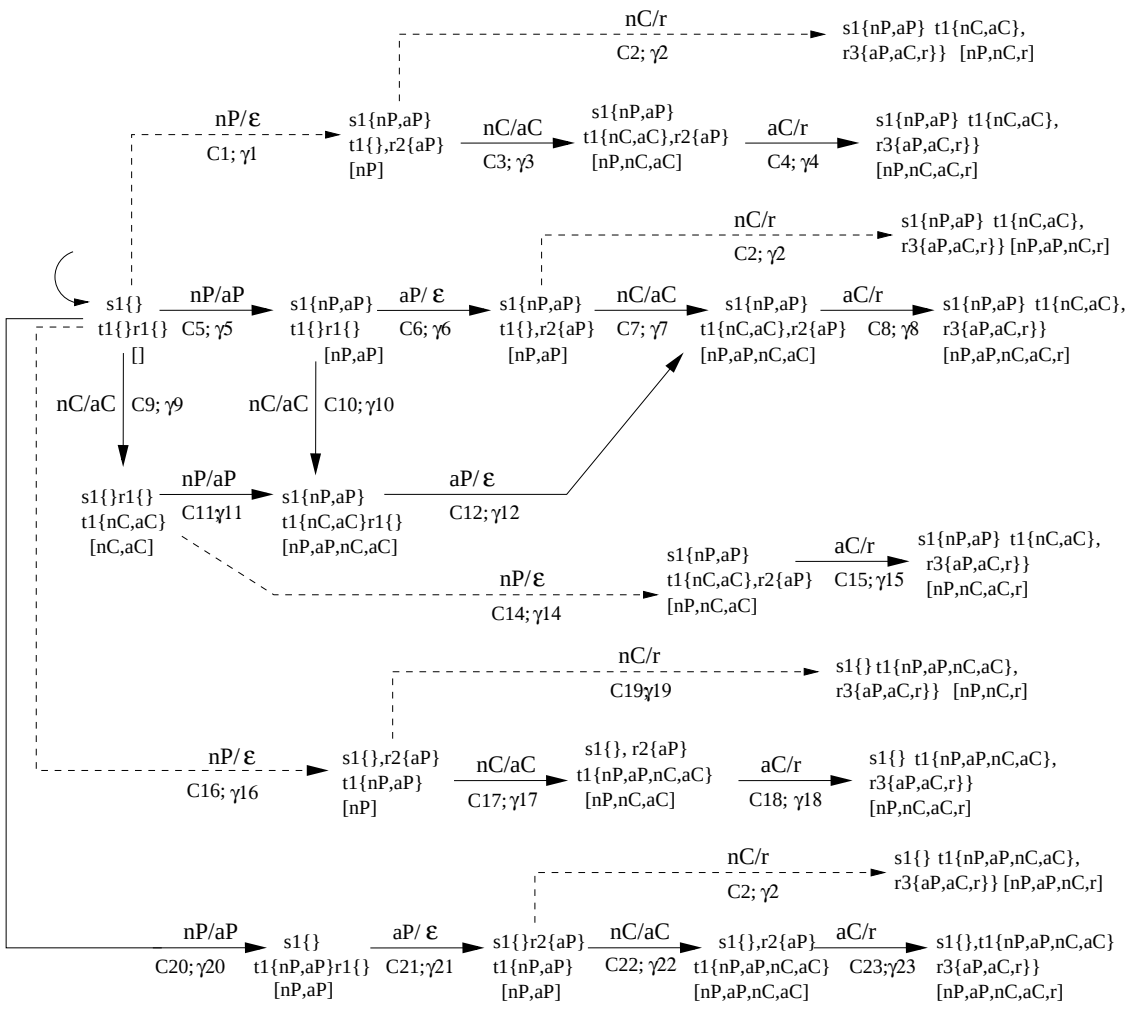


Figure 5.4: Transduced Closure Automaton, U

tion. Note that, communication between a pair of sites n and m will in general involve multiple options (such as different routes either directly between n and m or via intermediate nodes), and $c(n, m)$ denotes the cheapest option. The Table in Figure 5.1(b) presents the communication costs one site to another for our example. Since, each site is equipped with local history to store data, utilizing them, a sequence of input/output computations can be performed by the various site-services without the intervention of the client-site choreographer. The inputs for a computation should be produced from the history of the *nearest* site repository, whereas the outputs are sent to the client-site only when needed. The universe of all target or goal services that can be accomplished in this manner is computed via

the *transduced-closure* of an automaton with distributed history ($(\|_n^{\vec{H}} A_n)^T$), defined as follows. Note this definition is significantly different from the notion of transduced-closure introduced in the previous chapters, for it is designed to support decentralization and optimality. It might be noted that the union of all local histories would give us the *global history* which was introduced earlier in the centralized solution.

Definition 15 ($(\|_n^{\vec{H}} A_n)^T$ Automaton) *Given an interleaving product automaton with distributed history* $\|_n^{\vec{H}} A_n = (\vec{S} \times \vec{H}, \vec{S^0} \times \vec{H^0}, I_0, O_0, \Delta_{\vec{H}})$ *of* $\{A_n | 1 \leq n \leq N\}$*, its transduced-closure is the automaton* $(\|_n^{\vec{H}} A_n)^T = (\vec{S} \times (2^{I_0 \cup O_0} \times \vec{H}), \vec{S^0} \times (\emptyset \times \vec{H^0}),\ I_0, O_0, \Delta_{\vec{H}}^T)$*, where* $(\vec{s}, (h_0, \vec{h})) \xrightarrow[c,\gamma]{i/o} (\vec{s'}, (h_0', \vec{h'})) \in \Delta_{\vec{H}}^T$ *if and only if*

1. $\exists m.$
$$\left[\begin{array}{l} (\vec{s}, \vec{h}) \xrightarrow[n_1]{i/o_1} (\vec{s}_2, \vec{h}_2) \in \Delta_{\vec{H}} \wedge \\ (\vec{s}_2, \vec{h}_2) \xrightarrow[n_2]{i_2/o_2} (\vec{s}_3, \vec{h}_3) \in \Delta_{\vec{H}} \ \wedge \ldots \\ (\vec{s}_m, \vec{h}_m) \xrightarrow[n_m]{i_m/o} (\vec{s'}, \vec{h'}) \in \Delta_{\vec{H}} \wedge \\ \forall 2 \leq k \leq m : i_k \in h_0 \cup \bigcup\limits_{1 \leq n \leq N} \vec{h}_k(n) \end{array}\right]$$

2. $h_0' = h_0 \cup \{i, o\}$

3. $c :=$
$$\left[\begin{array}{l} \sum\limits_{k=2}^{m}{}_{MIN} \left[\begin{array}{l} \{c(n, n_k) \mid 1 \leq n \leq N : i_k \in \vec{h}_k(n)\} \\ \cup \{c(0, n_k) \mid i_k \in h_0\} \end{array}\right] \\ + c(0, n_1) \ + \ c(n_m, 0) \end{array}\right]$$

4. $\gamma = (\mathtt{src}_2, \ldots, \mathtt{src}_m)$ *where for* $2 \leq k \leq m$*:*
$$\mathtt{src}_k := \mathtt{arg} \left[MIN \left[\begin{array}{l} \{c(n, n_k) \mid 1 \leq n \leq N : i_k \in \vec{h}_k(n)\} \\ \cup \{c(0, n_k) \mid i_k \in h_0\} \end{array}\right]\right]$$

We call $U := (\|_n^{\vec{H}} A_n)^T$ *to be the* universal *service automaton corresponding to the service-automata* $\{A_n \mid 1 \leq n \leq N\}$.

Observe that, in the above definition, the states of U are represented by the states of $\|_n^{\vec{H}} A_n$ coupled with elements from $2^{I \cup O}$. The extra elements represent a history set of the client-site, i.e., the inputs

and outputs seen by the client-site choreographer. The above definition states that every transition in $\|_n^{\vec{H}} A_n$ is also a transition in the U automaton. Additionally, U automaton includes transduced-closure of transitions. The transduced-closure of a sequence of service-transitions is possible when certain conditions are satisfied. The first condition of the states that the source state, the input, the destination state, and the output of the transduced-closure transition matches respectively with the source state and the input of the first transition in the sequence, and destination state and the output of the last transition in the sequence. Furthermore, the input of each transition in the sequence should be present in some local history associated with its source state.

The second condition states that the history at the client site is updated by the input and output of the transduced-closure transition (since such a transition implements a goal transition whose input and output are relayed by the client-site choreographer). The third condition computes the overall cost of a transduced-closure transition as the sum of the costs of all the individual transitions in the sequence. The last condition identifies the site from which the input of an intermediate transition is obtained—it is the site possessing the input in its local history and nearest (in terms of communication cost) to the site executing the transition. Each transduced-closure transition is annotated with its cost c and the tuple γ of the nearest sources from where the inputs (for the sequence of transitions implementing the transduced-closure transition) are obtained.

Example 12 *Figure 5.4 depicts a part of the transduced closure automaton U obtained from $\|_n^{\vec{H}} A_n$ (Figure 5.3) of A_1, A_2 and A_3 (Figure 5.2). The history at the client site, h_0 is shown within [ ]. The dotted transition corresponds to the transitions obtained via the transduced-closure of a sequence of transitions. E.g., the transition* $s1\{\}t1\{\}r1\{\}[] \xrightarrow[40,1]{nP/\epsilon} s1\{nP,\ aP\}t1\{\}r2\{aP\}[nP]$ *is obtained from* $s1\{\}t1\{\}r1\{\} \xrightarrow[1]{nP/aP} s1\{nP,aP\}t1\{\}r1\{\} \xrightarrow[3]{aP/\epsilon} s1\{nP,aP\}t1\{\}r2\{aP\}$. *Note that the history of the source state of the second transition has* aP *available at site-*1. *Thus* $src_2 = 1$. *The cost of this transduced-closure transition is* $c = c(0,1) + c(1,3) + c(3,0) = 5 + 15 + 20 = 40$. *For simplicity, in the example we have depicted the communication costs, though, the computation costs at corresponding sites if present can be added to this cost for obtaining the cost labels of transitions.*

5.1.3 Realizability of goal

A given goal service A_0 is realizable from the existing services under a centralized/decentralized choreographer if and only if all input/output behaviors of A_0 are also present in the universal service automaton U. Note that the inputs in A_0 come from the client and the outputs from A_0 go to the client. Similarly the transition labels in U have inputs coming from the client and the outputs going to the client. The realizability of a goal using the existing services is verified using checking whether A_0 is simulated by U.

Definition 16 (Simulation [55]) *Given a goal automaton $A_0 = (S_0, S_0^0, I_0, O_0, \Delta_0)$ and an universal service automata $U = (S_U, S_U^0, I_0, O_0, \Delta_U)$, a state $s_1 \in S_0$ is simulated by a state $s_2 \in S_U$ if and only if they are related by the largest simulation relation denoted by $s_1 \sqsubseteq s_2$ and defined as: $s_1 \sqsubseteq s_2 \Rightarrow [\forall t_1 : s_1 \xrightarrow{i/o} t_1 \in \Delta_0 \Rightarrow (\exists t_2 : s_2 \xrightarrow[c,\gamma]{i/o} t_2 \in \Delta_U \wedge t_1 \sqsubseteq t_2)]$. A_0 is said to be simulated by U, denoted by $A_0 \sqsubseteq$ U, if all states in S_0^0 are simulated by some state in S_U^0.*

Then we have the following result.

Theorem 3 *Given a goal A_0 and a set of services $\{A_n \mid 1 \leq n \leq N\}$, the goal is realizable from the choreography of $\{A_n \mid 1 \leq n \leq N\}$ if and only if $A_0 \sqsubseteq$ U where U is the transduced-closure of the $(\|_n^{\vec{H}} A_n)$-automaton, and $(\|_n^{\vec{H}} A_n)$ is the interleaving product with distributed history of the automata $\{A_n \mid 1 \leq n \leq N\}$.*

Example 13 *It can be seen that the goal A_0 given in Figure 5.2 is simulated by the U automaton in the Figure 5.4. Thus A_0 can be realized by choreographing the services A_1, A_2, A_3 of Figure 5.2.*

It can be verified that $A_0 \sqsubseteq$ U holds if and only if $A_0 \sqsubseteq A_0 \times$ U holds, where $A_0 \times$ U denotes "simulating synchronous product" of A_0 and U as defined below:

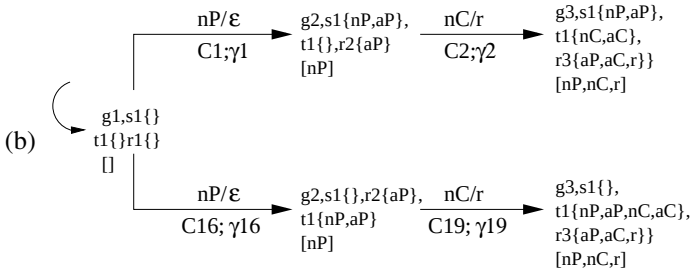
Definition 17 (Simulating Synchronous Product) *Given a goal $A_0 = (S_0, S_0^0, I_0, O_0, \Delta_0)$ and an universal service automaton U $= (S_U, S_U^0, I_0, O_0, \Delta_U)$, their simulating synchronous product is the automaton $A_0 \times$ U $= (S_0 \times S_U, S_0^0 \times S_U^0, I_0, O_0, \Delta_\times)$, where*

$$(s_0, s_u) \xrightarrow[c,\gamma]{i/o} (s_0', s_u') \in \Delta_\times \Leftrightarrow \left\{ \begin{array}{l} s_0 \xrightarrow{i/o} s_0' \wedge \; s_u \xrightarrow[c,\gamma]{i/o} s_u' \\ \wedge \; s_0 \sqsubseteq s_u \wedge s_0' \sqsubseteq s_u'. \end{array} \right\}$$

(a)

Label	Sources	Label	Sources	Label	Sources
γ_1	-	γ_2	-	γ_3	-
γ_4	-	γ_5	-	γ_6	-
γ_7	-	γ_8	-	γ_9	-
γ_{10}	-	γ_{11}	-	γ_{12}	-
γ_{13}	1	γ_{14}	-	γ_{15}	-
γ_{16}	2	γ_{17}	2	γ_{18}	1
γ_{19}	-	γ_{20}	2	γ_{21}	-
γ_{22}	-	γ_{23}	2	γ_{24}	2

(b)



(c)

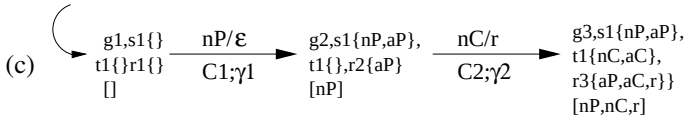


Figure 5.5: (a) Valuations of γ_is, (b) Simulating synchronous product $A_0 \times \mathtt{U}$, and (c) Mincost Choreography Automaton, C.

Usually the size of $A_0 \times \mathtt{U}$ is smaller compared to the size of $\mathtt{U}$ (since size of A_0 is smaller compared size of $\mathtt{U}$). Hence it is preferable to check whether $A_0 \sqsubseteq A_0 \times \mathtt{U}$ holds (as opposed to checking whether $A_0 \sqsubseteq \mathtt{U}$ holds).

Example 14 *Figure 5.5(b) shows the simulating synchronous product of the goal automaton A_0 in Figure 5.2 and universal service automaton U in Figure 5.4. It can be seen from inspecting $A_0 \times \mathtt{U}$ and A_0 that $A_0 \sqsubseteq A_0 \times \mathtt{U}$ holds. Here, $A_0 \times \mathtt{U}$ has two paths from the start state both of which can yield choreographers. From the associated histories in the paths it can be seen that one path uses service A_1 for computing `nP/aP` and other path uses service A_2 for the same. In the following sections we introduce the algorithm for choosing the optimal solution.*

5.2 Optimum Decentralization for Realizing Acyclic Goals

Realizability of a goal A_0 by choreographing a set of services $\{A_n \mid 1 \leq n \leq N\}$ is guaranteed by the satisfaction of $A_0 \sqsubseteq A_0 \times \mathtt{U}$. It is possible that A_0 can be simulated by $A_0 \times \mathtt{U}$ in multiple ways since $A_0 \times \mathtt{U}$ can possess multiple subautomata each of which can simulate A_0. Thus there can be multiple realizations of A_0, each with its own cost (as defined below). Our goal then is to find an optimum cost realization of A_0, which we approach by finding an optimal cost subautomaton of $A_0 \times \mathtt{U}$ that simulates A_0. In what follows next, we assume for simplicity of presentation that A_0 is loop-free. This then implies that $A_0 \times \mathtt{U}$ is also loop-free. We proceed by defining the cost of a subautomaton of $A_0 \times \mathtt{U}$.

Definition 18 (Cost of a loop-free Automaton) *Given a loop-free I/O-automaton with its each transition labeled by a cost, we define the cost of a path to be the sum of the costs of all the transitions in the path. We define the cost of a state to be the maximum cost among all paths originating at that state and terminating at a deadlocking state (state with no outgoing transitions). The cost of an automaton is defined to be the maximum cost among all its initial states. We represent the cost of a loop-free I/O-automaton A as $\mathtt{cost}(A)$.*

Using the notion of cost from Definition 18, we define the minimum cost choreography automaton for realizing A_0 from $\{A_n \mid 1 \leq n \leq N\}$.

Definition 19 (MinCost Choreography Automaton) *Given a goal automaton A_0 and an universal service automaton $\mathtt{U}$ such that $A_0 \sqsubseteq A_0 \times \mathtt{U}$, the minimum cost for choreography is obtained as the cost of a subautomaton C of $A_0 \times \mathtt{U}$ such that $A_0 \sqsubseteq C$ and for all subautomata C' of $A_0 \times \mathtt{U}$ with $A_0 \sqsubseteq C'$, it holds that $\mathtt{cost}(C) \leq \mathtt{cost}(C')$.*

5.2.1 Computing optimum cost for acyclic case

In the following we present an algorithm for computing a subautomaton of $A_0 \times \mathtt{U}$ from which an optimum choreographer can be extracted. Given a state (s_0, s_u) of $A_0 \times \mathtt{U}$, a certain goal transition $s_0 \xrightarrow{i/o} s_0'$ may be simulated by multiple transitions of the type $(s_0, s_u) \xrightarrow[\bar{c},\bar{\gamma}]{i/o} (\bar{s}_0, \bar{s}_u)$ in $A_0 \times \mathtt{U}$. The

algorithm identifies the minimum cost option by searching over all alternatives. The computation starts from the deadlocking states by assigning their cost to be zero, and recursively proceeds backwards by assigning costs to the predecessor states. In this example, for the sake of brevity, we consider the communication costs as described before. The computation costs will be introduced for solving the case where goals can have loops.

Algorithm 1 $\mathtt{cost}(s_0, s_u) =$

$$\left\{ \begin{array}{l} 0 \quad \textit{if } (s_0, s_u) \textit{ is a deadlocking state} \\ \\ \underset{s_0 \xrightarrow{i/o} s_0' \in \Delta_0}{\mathit{MAX}} \left[\underset{(s_0,s_u) \xrightarrow[\bar{c},\bar{\gamma}]{i/o} (\bar{s}_0,\bar{s}_u) \in \Delta_\times}{\mathit{MIN}} \{\bar{c} + \mathtt{cost}(\bar{s}_0, \bar{s}_u)\} \right] \\ \\ \quad \textit{otherwise} \end{array} \right\}$$

In the above, the first case states that the cost of a deadlocking state (s_0, s_u) is 0 as there is no path from such a state. In other words, the *leaf* states of the automaton are assigned cost of 0. The second case corresponds to non-deadlocking states. The cost of such a state (s_0, s_u) can be understood in two steps. In the first step (minimization), we identify the cheapest way of simulating a goal transition $s_0 \xrightarrow{i/o} s_0'$ originating at s_0. This corresponds to a transition of $A_0 \times \mathtt{U}$ labeled by i/o having the least sum of the cost of the transition and the cost of its destination state. In the second step (maximization), we identify the worst cost of simulating a transition of the type $s_0 \xrightarrow{i/o} s_0'$ originating at s_0. Thus, the second case boils down to a recursive computation where we update the cost of a state starting from the deadlocking states, taking the cheapest alternatives. Once, all the states are marked to have a cost, there may be more than one option to simulate a particular sequence of inputs/outputs. We pick the cheapest way to simulate each such sequence. And the maximum among them would give the cost of the automaton.

Example 15 *For our running example, the automaton C, shown in Figure 5.5(c), represents the optimum choreographer (from the two candidate choreographers: Figure 5.5(b)). One choreography obtains the* `nP/aP` *from* A_1 *(service 1) while the other does the same operation using* A_2 *(service 2). In both case,* `aP` *is sent to* A_3 *(service 3) for outputting* `r`*. Note that the sum of communication cost*

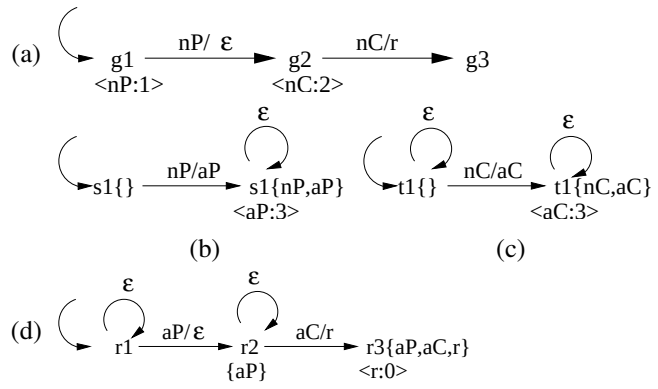


Figure 5.6: Site-specific Choreographers: (a) C_0, (b) C_1, (c) C_2 and (d) C_3.

between client-site and service 1 and between service 1 and service 3 is less than the sum of cost of communication between client-site and service 2 and between service 2 and service 3 (see Figure 5.1(b)). As a result, the transition simulating $\mathtt{nP}/\epsilon$ *has two different costs associated with it in Figure 5.5(b): in one it is the sum of communication cost between client to service 1, service 1 to service 3 and service 3 to client (total:* 40*); in the other it is the sum of the communication cost between client to service 2, service 2 to service 3 and service 3 to client (total:* 45*). The first case produces a minimum cost choreography strategy.*

5.2.2 Synthesizing optimum choreographers

Starting from a subautomaton C of $A_0 \times \mathtt{U}$ representing an optimum choreography scheme, a set of site-specific choreographers achieving the optimum cost can be obtained using the following algorithm.

Algorithm 2 (Site-specific Choreographer) *Given a minimum cost choreography subautomaton* C *of* $A_0 \times \mathtt{U}$*, the choreographer at site-*n *(*$0 \leq n \leq N$*) is a communicating I/O-automaton* $C_n = (S_n \times 2^{I_n \cup O_n}, S_n^0 \times \{\emptyset\}, I_n, O_n, E_n, \Delta_n^C)$*, where* S_n, S_n^0, I_n, O_n *are as in the site-*n *service or goal automaton* A_n*, while* $E_n : S_n \times 2^{I_n \cup O_n} \rightarrow 2^{(I_0 \cup O_0) \times \{0,\ldots,N\}}$ *labels each state of* C_n *with a set of data (that* C_n *should send in this state) along with their destinations.* E_n *and* Δ_n^C *are obtained as follows. Suppose there exists a transition* $(s_0, \vec{s_1}, (h_0, \vec{h_1})) \xrightarrow[c,\gamma]{i/o} (s_0', \vec{s}_{m+1}, (h_0', \vec{h}_{m+1}))$ *in* C*, which implies:*

- $\exists m.$ $\left[\begin{array}{l} (\vec{s},\vec{h}) \xrightarrow[n_1]{i/o_1} (\vec{s}_2,\vec{h}_2) \in \Delta_{\vec{H}} \wedge \\ (\vec{s}_2,\vec{h}_2) \xrightarrow[n_2]{i_2/o_2} (\vec{s}_3,\vec{h}_3) \in \Delta_{\vec{H}} \ \wedge \ldots \\ (\vec{s}_m,\vec{h}_m) \xrightarrow[n_m]{i_m/o} (\vec{s}_{m+1},\vec{h}_{m+1}) \in \Delta_{\vec{H}} \wedge \\ \forall 2 \le k \le m : i_k \in h_0 \cup \bigcup\limits_{1\le n\le N} \vec{h}_k(n) \end{array}\right]$
- $h'_0 = h_0 \cup \{i, o\}$
- $c = \left[\begin{array}{l} \sum\limits_{k=2}^{m} \mathit{MIN} \left[\begin{array}{l} \{c(n,n_k) \mid 1 \le n \le N : i_k \in \vec{h}_k(n)\} \\ \cup\{c(0,n_k) \mid i_k \in h_0\} \end{array}\right] \\ \\ + \ c(0,n_1) \ + \ c(n_m,0) \end{array}\right]$
- $\gamma = (\mathit{src}_2, \ldots, \mathit{src}_m)$ *where for* $2 \le k \le m$*:*
  $\mathit{src}_k := \mathit{arg}\left[\mathit{MIN}\left[\begin{array}{l} \{c(n,n_k) \mid 1 \le n \le N : i_k \in \vec{h}_k(n)\} \\ \cup \{c(0,n_k) \mid i_k \in h_0\} \end{array}\right]\right]$
- $s_0 \xrightarrow{i/o} s'_0 \in \Delta_0$

Then

1. $\forall 1 < k \le m$*:* $[(n_k = n) \Rightarrow$
   $(\vec{s}_k(n),\vec{h}_k(n)) \xrightarrow{i_k/o_k} (\vec{s}_{k+1}(n),\vec{h}_{k+1}(n)) \in \Delta_n^C]$,
2. $\forall 1 < k \le m$*:* $[(\mathit{src}_k = n) \Rightarrow (i_k, n_k) \in E_n(\vec{s}_k(n),\vec{h}_k(n))]$,
3. $(n_m = n) \Rightarrow (o, 0) \in E_n(\vec{s}_{m+1}(n),\vec{h}_{m+1}(n))$,
4. $(i, n_1) \in E_0(s_0, h_0)$,
5. $(s_0, h_0) \xrightarrow{i/o} (s'_0, h'_0) \in \Delta_0^C$.

Observe that the service-site choreographer C_n is a subautomaton of the "history-augmented" service automaton A_n[1], but with the added feature that it can perform certain transmissions at its states (as

[1]Item 1 ensures that the choreographer at every site is subautomaton of the automaton of the service at that site. For the client-site choreographer, item 5 ensures that it is a subautomaton of the goal automaton.

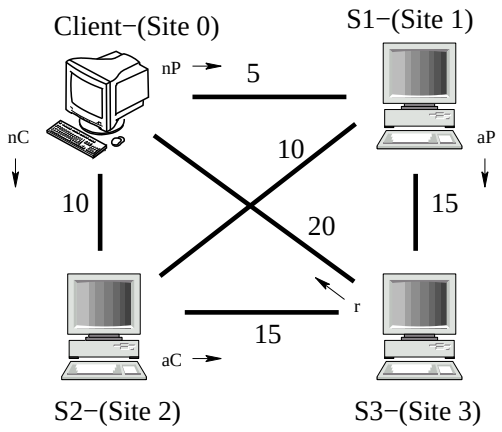


Figure 5.7: Decentralized execution

determined by the labeling function E_n). If the kth service-transition in a transduced-closure transition is such that $n_k = n$ (item 1), it implies that the kth service-transition is executed at site-n, and in which case site-n performs this computation (and doesn't initiate any communication). On the other hand if the kth service-transition is such that $\texttt{src}_k = n$ (item 2), then site-n is the source for the input i_k, which it sends to the site-n_k (where the kth service-transition is executed). When $n_m = n$ (item 3), the last service-transition in a transduced-closure transition is executed at site-n which sends the output o to site-0. Similarly, site-0 sends the input i to the site-n_1 (item 4) that executes the 1st service-transition in the sequence.

Example 16 *Figure 5.6 shows the various site-specific choreographers as obtained by applying Algorithm 2. The labeling of states as implied by the E_n function is shown within <>. In Figure 5.6(a), we obtain the choreographer C_0 at the client-site. C_0 communicates `nP` to the choreographer C_1 at site-1, and `nC` to the site-2 choreographer C_2 from the initial and the next successor states, respectively. In Figure 5.5(a), it can be seen that after C_1 (at service 1 site) obtains `nP` from C_0, it computes the output `aP` and sends that to C_3. Other choreographers can be explained in similar fashion. They realize the goal in with minimum cost (as per cost table in Figure 5.1(b)) as the choreographers are obtained from the minimum cost automaton in Figure 5.5(c). A choreographed view of the solution is depicted in Figure 5.7. The paths are labeled with the cost and the destination of the different messages are shown by arrows. Note that the optimal path depicted for such a message may not be the direct channel be-*

tween the two concerned services, but may be through other nodes as determined by the routing layer to achieve that optimal cost. In the solution, `nP` *is sent to* C_1*, whereas it could also have been sent to the service 2 site. But that would cost more in the overall optimization scheme.* `nC` *is sent to* C_2 *and the outputs are directly sent to* C_3 *which generates* `r`*, after getting the two inputs. Finally* `r` *is sent back to* C_1 *at client site. While in this example we depicted the communication costs, in the next section we will add the computation costs at each service in the overall optimization scheme.*

We developed a tool in java which takes as input the various service models with computation costs for each transition and a cost model for communication among the services, and produces as output the optimal decentralization scheme as the various communicating automata in the different sites. The tool is elaborated in the Appendix.

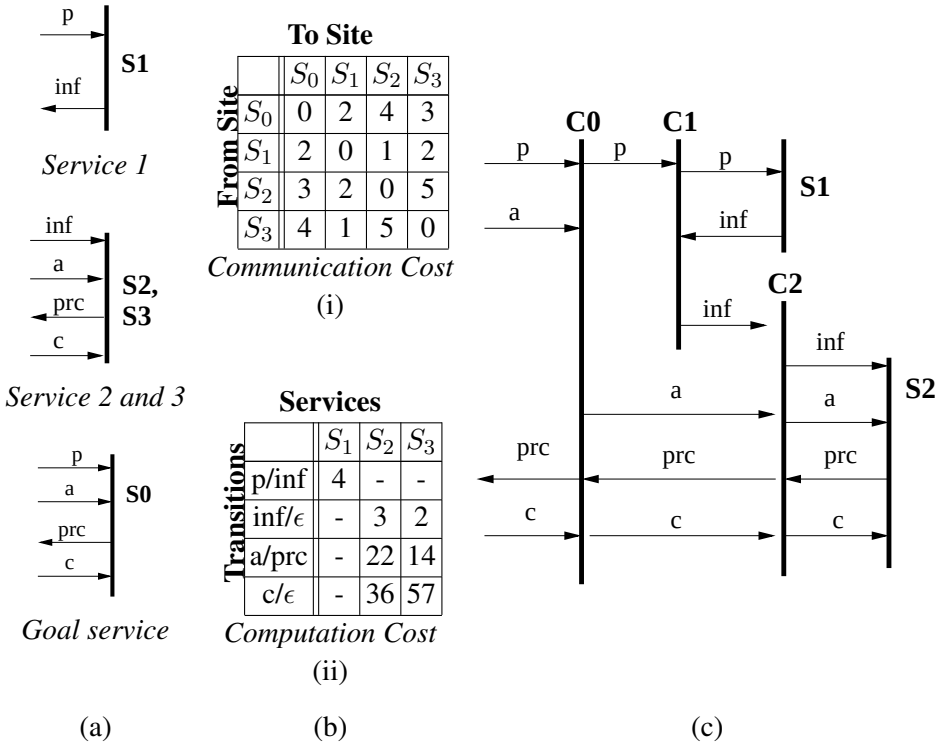


Figure 5.8: (a) Existing Services, (b) goal service & cost metrics, and (c) decentralized choreography

5.3 Optimal choreography for Goal having cycles

In the preceding sections, we presented the formulation and algorithms for computing the optimum decentralized choreographed composition for goal services that are loop-free. The end result is a set of choreographers derived from our algorithm which are to be placed at the corresponding sites where the participating services reside. In this paper, we consider a generalized scenario where the goal service can have a sequence of operations which can be repeated any number of times, thus forming a cycle in its behavioral specification as an automaton. We proceed by introducing the problem with an illustrative example and provide an overview of our solution mechanism. The example will be used in the rest of the paper to explain the salient aspects of our technique. The technique forms the basis of [17].

Illustrative Example. Figure 5.8(a) presents sequence diagrams of three services `S1`, `S2` and `S3`. `S1` takes as input a product name (`p`) and provides some information (`inf`) about the product, such as

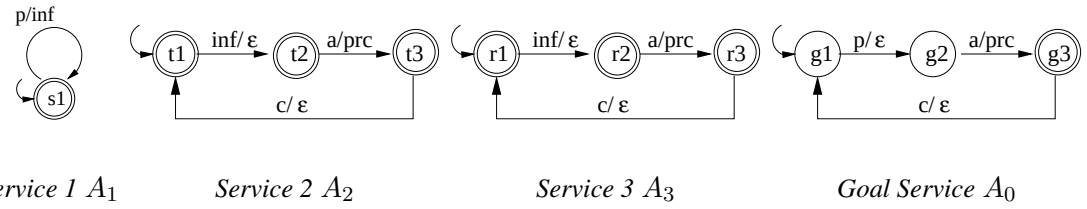


Figure 5.9: i/o-automata representation of the services

weight, size etc. Service `S2` takes as input the information (`inf`) about a product and a shipping address (`a`) and provides as output the price (`prc`) for shipping the product to the address `(a)`. The client can decide to cancel (`c`) the shipping after the quote (`prc`) is given out as output. `S3` is a service similar to `S2`, the difference being it is located somewhere other than `S2` and has different computation costs for its operations and communication costs of inputs and outputs from and to the client are different as well. Note that `S2` and `S3` are shown by the same sequence diagram. The developer wants to create a new service, `S0`, for a client which takes product (`p`) and address (`a`) as inputs and provides the price (`prc`) of shipping `(p)` to (`a`), which is also depicted in (Figure 5.8(a)).

A decentralized realization of the choreography will use the services `S1`,`S2` and `S3` to provide the goal `S0` of the client. This will entail placing choreographers (`C0`, `C1`, `C2`, `C3`) at the respective sites.

Now consider that the cost of communication (in terms of usage of the network) between the client and services and between any services (table of Figure 5.8(b-i)). Another table presents the computation costs for each operation in each of the services and is shown in Figure 5.8(b-ii). Each operation is showed in its input/output format, depicting the input required to do the operation and the output available after the operation. The cost here can be viewed as the charge for using that operation of the service. Observe that, the computation costs for the same operation by services `S2` and `S3` are different. We assume without loss of generality, the communication and computation costs are normalized to the same unit of measure.

Objective. Given a set of services, a goal, and communication and computation costs, our objective is to devise an *optimal decentralization choreography scheme* that will realize the goal from the existing services by incurring the minimum cost. To satisfy the above objective, multiple choreographers are deployed at servers which are at close proximity (physically or in terms of request/response

delay) to the existing services. Not all sites may have choreographers in this scheme as they may not participate in the optimal choreography scheme.

A possible choreography scheme is shown in Figure 5.8(c). It may be noted that the service `S3` remains unused in the scheme. This is because both services `S2` and `S3` provide the same functionality. Thus, any one of them can be used to realize the goal. The realization of the goal involves the communication of `p` from `C0` at the client site to `C1` at site 1, `inf` from the choreographer `C1` to `C2` which is the choreographer at site 2, `a` from `C0` to `C2`, and finally output `prc` from `C2` to `C0`. Moreover the client can input a request `c` to cancel order, which similarly needs to be communicated to `C2`. If `S3` is to be used, the communication needs to be between the choreographers `C0`,`C1` and the choreographer `C3` deployed at site 3. If `S2` is used, the overall cost of the cancelation operation from the perspective of the client would be communication cost from `C0` to `C2` added to the computation cost at `S2`, which is $4+36 = 40$. If instead `S3` is used, then by similar calculations the cost would turn out to be $3 + 57 = 60$. The cost of the composite behaviors as presented in Figure 5.8(c) is equal to the sum of the cost of exchanging messages as shown between `C0`, `C1` and `C3` and the respective computation costs at `S1` and `S2` to produce the desired outputs. Weighing the communication costs and the computation costs for each operation, `S2` might be a cheaper alternative than `S3`.

Proposed solution. All services and the goal are represented by i/o-automata (Figure 5.9). Observe that each of the automata (services and goal) possess loops. The states enclosed in double-circle denote final states. For a goal service, reaching any final state signifies completion of a task. In contrast, being at a non-final state signifies a pending task. Furthermore, if a final state in the goal is non-deadlocking, another task initiates from that final state and its completion occurs when a subsequent final state is reached. Note that it is the goal service that defines task-completions in terms of its final states. When a final state is reached in the goal, any state can be reached in the services used to realize the goal. For this reason any state in the given services is treated final. The objective of optimization is to choreograph the existing services in such a way that regardless of the history of evolution, any pending task is completed in a minimal cost (the worst cost between any state and its nearest reachable final states is minimized).

To solve the above problem, we follow our approach of computing the interleaving product and

transduced closure of the services, which generates all the choreographed behaviors of the existing services (in form of a "universal automaton"). Then, we check using simulation whether the goal behaviors are subsumed by the universal automaton behaviors. The universal automaton is constructed to keep track of the costs for computations at respective sites as well as of communications from one site to another. This is done by annotating the transitions with the cost required to realize them. An optimum solution is derived as an optimum cost subautomaton of the universal automaton that simulates the goal automaton. As mentioned above the objective function is to ensure that regardless of the history of evolution, the worst cost from any state to its nearest reachable final state is minimized. We reduce the search space by taking the synchronous product of the universal automaton with the goal automaton, and compute an optimum subautomaton of this product automaton. Using the optimum subautomaton, we next obtain the choreographers at the service sites simply by applying projection operations.

The **contributions** of this section can be summarized as follows. This is a first approach which presents an automated solution to optimum decentralized choreography for Web services composition, where loops are considered in the goal service. In the absence of loops in the goal specification, the optimum cost computation is already obtained at this point, i.e., no further iteration is required. In the presence of loops however, it is possible that the goal specification doesn't terminate at a final state, and thus the cost of such a final state is non-zero. So there is a possibility of lowering the overall cost if the "payoff" of reaching a final state (namely lowering of costs of other states by completing tasks through reaching the final state) is overshadowed by the "penalty" of reaching it (namely incurring the cost of the final state). Thus further iterations are required to explore this trade-off, which makes the algorithm for specification with loops non-trivially different from the loop-free specifications, where a straightforward backward search starting from terminating states suffices.

5.4 Services as I/O-automata with Final States

As described in the earlier chapters, the behaviors of a Web service can be described as a set of sequences of input and output computations it can perform. The states in an i/o-automaton represent the configuration of a service and the inter-state transitions represent the changes from one configuration to another. The transitions are labeled with input/output actions. We alter the definition of i/o automaton

to accommodate the final states.

Definition 20 (I/O Automaton) An i/o-automaton A is defined by a tuple $(S, S^0, S^F, I, O, \Delta)$, where, S is the set of states, $S^0 \subseteq S$ is the set of initial states, $S^F \subseteq S$ is the set of final states, I is the set of inputs, O is the set of outputs, and $\Delta \subseteq S \times (I \cup \{\epsilon\}) \times (O \cup \{\epsilon\}) \times S$ is the set of transitions. An element of Δ, represented by (s, i, o, s'), is such that $s \in S$ is the origin state of the transition, $i \in I \cup \{\epsilon\}$ is the input to the transition, $o \in O \cup \{\epsilon\}$ is the output of the transition, and $s' \in S$ is the destination state of the transition. We use $s \xrightarrow{i/o} s'$ to denote $(s, i, o, s') \in \Delta$.

Figure 5.9 presents the i/o-automata models for the three services and goal described in Figure 5.8(a). The automaton A_i ($i = 1, 2, 3$) corresponds to the i-th service and the automaton A_0 corresponds to the goal. The start states of the automata have curved arrows pointed to them and the final states are marked with double-circles.

5.5 Choreographer Existence

Each operation of a service is depicted as an input/output operation (transition) of the corresponding i/o-automaton. In order to realize the goal, each input/output operation of the goal needs to be realized by a sequence of input/output operations of the existing services, such that the input of the goal transition matches with the first input of the sequence and the output matches with the last output of the sequence. The inputs provided to a service and the outputs computed by that service can be stored at the corresponding service-site for future use (for use as inputs for future computations or for relaying to other services or to the client). This stored information is referred to as the local *history* of the service-site. To formalize these concepts we define the notion of *interleaving product with distributed history*.

5.5.1 Product with Distributed History

Similar to the acyclic case, the service at site-n ($n \in \{1, \ldots, N\}$) is modeled as an i/o-automaton $A_n = (S_n, S_n^0, S_n, I_n, O_n, \Delta_n)$. Note that here, each state is treated a final state (and so the third tuple-element is the same as the first tuple-element), since after reaching any state the service may no

longer be required for realizing the goal. We designate site-0 as the site interfacing with the client or the goal service.

We beef up Definition 14 with the introduction of final states as follows:

Definition 21 *[$\|_n^{\vec{H}} A_n$ Automaton]*Given service automata $\{A_n = (S_n, S_n^0, S_n, I_n, O_n, \Delta_n) | 1 \leq n \leq N\}$, their *interleaving product with distributed history* is defined as the i/o-automaton $\|_n^{\vec{H}} A_n = (\vec{S} \times \vec{H}, \vec{S^0} \times \vec{H^0}, \vec{S} \times \vec{H}, I_0, O_0, \Delta_{\vec{H}})$ where

$$\vec{S} = \prod_{n=1}^{N} S_n, \;\; \vec{S^0} = \prod_{n=1}^{N} S_n^0,$$

$$\vec{H} = \prod_{n=1}^{N} 2^{I_n \cup O_n}, \;\; \vec{H^0} = \prod_{n=1}^{N} \{\emptyset\},$$

$$I_0 = \bigcup_{n=1}^{N} I_n, O_0 = \bigcup_{n=1}^{N} O_n, \text{and}$$

$$(\vec{s}, \vec{h}) \xrightarrow[n]{i/o} (\vec{s'}, \vec{h'}) \in \Delta_{\vec{H}} \text{ if and only if}$$

$$\vec{s}(n) \xrightarrow{i/o} \vec{s'}(n) \in \Delta_n \;\wedge\; \vec{h'}(n) = \vec{h}(n) \cup \{i, o\} \wedge$$

$$\forall m \neq n : \vec{s'}(m) = \vec{s}(m) \wedge \; \vec{h'}(m) = \vec{h}(m).$$

Note that all states in the interleaving product are final states as the constituent states of those states from the individual services are also final states.

Example 17 Figure 5.10 depicts a part of the automaton $||_n^{\vec{H}} A_n$ for A_1, A_2 and A_3 presented in Figure 5.9. Observe that, the history of the start state is empty. Every state is shown with the local history associated with it and transitions are labeled with the participating service responsible for the transition. For example `s1{}t1{}r1{}` changes its configuration to `s1{p,inf}t1{}r1{}` when A_1 makes a transition, and that transition is annotated with the site-index 1.

5.5.2 Transduced Closure Automaton

In this example we explicitly show the communication and computation costs for the operations performed. The table in Figure 5.8(b-i) presents the communication cost for our example. Utilizing the local histories, a sequence of input/output computations can be performed by the various site-services without the intervention of the client-site choreographer. The inputs for these computations

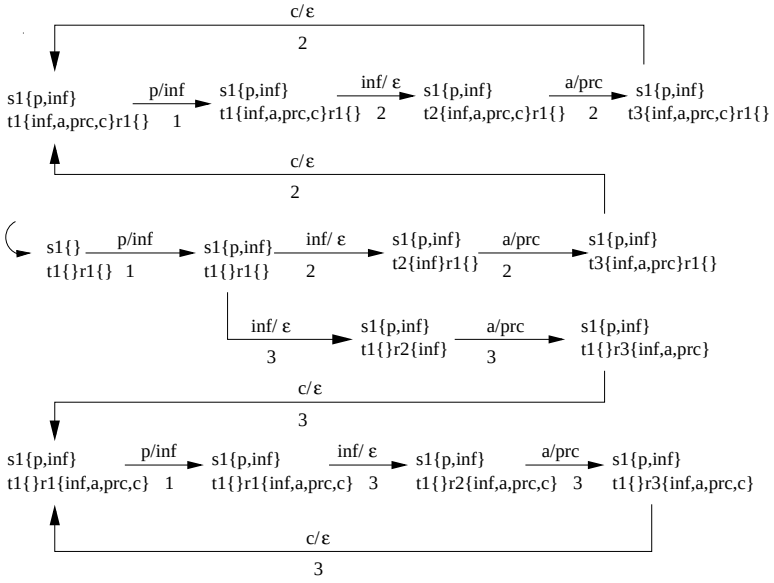


Figure 5.10: Interleaving Product Automaton $\|_n^{\vec{H}} A_n$

are produced from the history of the *nearest* site repository, whereas the outputs are sent to the client-site only when needed. Further note that the following differs from Definition 15 as each transition also incurs a computation cost (for our example it is summarized in Table 5.8(b-ii)). We will denote the computation cost of a transition $s \xrightarrow{i/o} s'$ as $w(s \xrightarrow{i/o} s')$. The universe of all choreographed behavior of existing services that can be accomplished in the manner described above is computed via the *transduced-closure* of the automaton with distributed history ($(\|_n^{\vec{H}} A_n)^T$), and is defined as follows.

Definition 22 ($(\|_n^{\vec{H}} A_n)^T$ Automaton) Given an interleaving product automaton with distributed history $\|_n^{\vec{H}} A_n = (\vec{S} \times \vec{H}, \vec{S^0} \times \vec{H^0}, \vec{S} \times \vec{H}, I_0, O_0, \Delta_{\vec{H}})$ of $\{A_n | 1 \leq n \leq N\}$, its transduced-closure is the automaton $(\|_n^{\vec{H}} A_n)^T = (\vec{S} \times (2^{I_0 \cup O_0} \times \vec{H}), \vec{S^0} \times (\{\emptyset\} \times \vec{H^0}), \vec{S} \times (2^{I_0 \cup O_0} \times \vec{H}), I_0, O_0, \Delta_{\vec{H}}^T)$, where $(\vec{s}, (h_0, \vec{h})) \xrightarrow[c,\gamma]{i/o} (\vec{s'}, (h_0', \vec{h'})) \in \Delta_{\vec{H}}^T$ if and only if

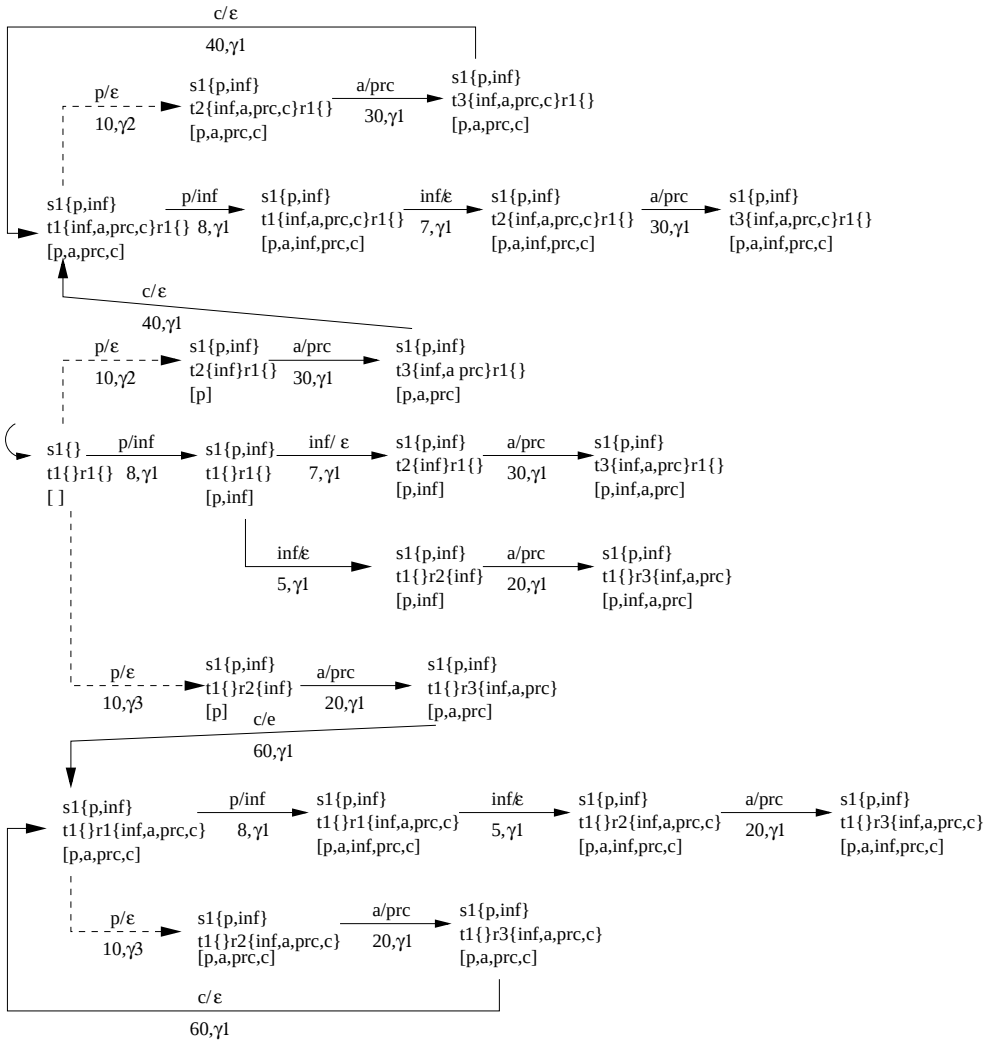


Figure 5.11: Transduced Closure Automaton U

$$1.\ \exists m. \left[\begin{array}{l} (\vec{s},\vec{h}) \xrightarrow[n_1]{i_1/o_1} (\vec{s}_2,\vec{h}_2) \in \Delta_{\vec{H}} \wedge \\ (\vec{s}_2,\vec{h}_2) \xrightarrow[n_2]{i_2/o_2} (\vec{s}_3,\vec{h}_3) \in \Delta_{\vec{H}} \ \wedge \ldots \\ (\vec{s}_m,\vec{h}_m) \xrightarrow[n_m]{i_m/o_m} (\vec{s'},\vec{h'}) \in \Delta_{\vec{H}} \wedge \\ \forall 2 \leq k \leq m : i_k \in h_0 \cup \bigcup\limits_{1 \leq n \leq N} \vec{h}_k(n), \\ i_1 = i, o_m = o \end{array}\right]$$

2. $h'_0 = h_0 \cup \{i, o\}$

3. $c := \begin{cases} c(0, n_1) \ + \ w(\vec{s}(n_1) \xrightarrow{i/o_1} \vec{s}_2(n_1)) \\ \sum_{k=2}^{m} \left[\begin{array}{l} \texttt{MIN} \left[\begin{array}{l} \{c(n, n_k) \mid 1 \leq n \leq N : \\ i_k \in \vec{h}_k(n)\} \\ \cup \{c(0, n_k) \mid i_k \in h_0\} \end{array} \right] \\ + w(\vec{s}_k(n_k) \xrightarrow{i_k/o_k} \vec{s}_{k+1}(n_k)) \end{array} \right] \\ + c(n_m, 0) \end{cases}$

4. $\gamma = (\texttt{src}_2, \ldots, \texttt{src}_m)$ where for $2 \leq k \leq m$:

$$\texttt{src}_k := \texttt{arg} \left[\begin{array}{l} \texttt{MIN} \left[\begin{array}{l} \{c(n, n_k) \mid 1 \leq n \leq N : i_k \in \vec{h}_k(n)\} \\ \cup \{c(0, n_k) \mid i_k \in h_0\} \end{array} \right] \\ + w(\vec{s}_k(n_k) \xrightarrow{i_k/o_k} \vec{s}_{k+1}(n_k)) \end{array} \right]$$

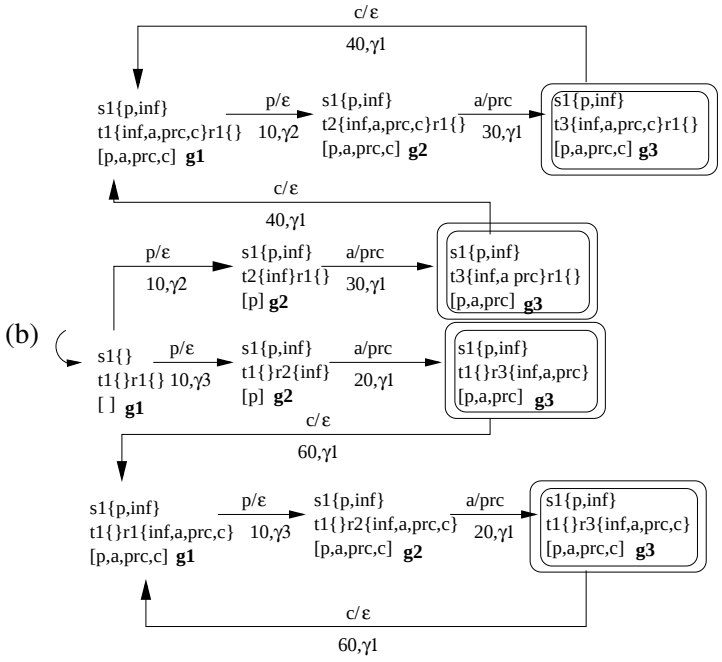
We call $\texttt{U} := (\|_n^{\vec{H}} A_n)^T$ to be the *universal* service automaton corresponding to the service-automata $\{A_n \mid 1 \leq n \leq N\}$.

In this definition the clauses are explained similar to the definition 15. Here, the third condition computes the overall cost of a transduced-closure transition as the sum of the costs of all the individual transitions in the sequence. This involves summation over all the computation costs of the transitions involved and the minimum possible communication cost to procure the inputs.

Example 18 Figure 5.11 depicts a part of the transduced closure automaton U obtained from $\|_n^{\vec{H}} A_n$ (Figure 5.10) of A_1, A_2 and A_3 (Figure 5.9). The history at the client site, h_0 is shown within [ ]. The dotted transitions correspond to the transitions obtained via the transduced-closure of a sequence of transitions. E.g., the transition $\texttt{s1\{\}t1\{\}r1\{\}[]} \xrightarrow[10,1]{\texttt{p}/\epsilon} \texttt{s1\{p,inf\}t1\{\}r2\{inf\}[p]}$ is obtained from the transduced-closure of $\texttt{s1\{\}t1\{\}r1\{\}} \xrightarrow[1]{\texttt{p/inf}} \texttt{s1\{p,inf\}t1\{\}r1\{\}} \xrightarrow[3]{\texttt{inf}/\epsilon} \texttt{s1\{p,inf\}t1\{\}r2\{inf\}}$. Note that the history of the source state of the second transition has `inf` available at site-1. Thus $\texttt{src}_2 = 1$. The cost of this transduced-closure transition is $c = 4 + 2$ (computation costs for the transitions involved)$+c(0, 1) + c(1, 3)$ (communication costs) $= 10$. Note that the ϵ output does not need to be communicated and hence the communication cost associated with it is not added.

(a)

Label	Sources	Label	Sources	Label	Sources
γ_1	-	γ_2	1	γ_3	1



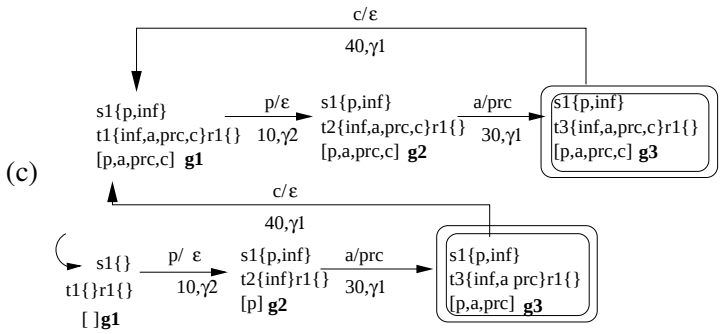


Figure 5.12: (a) Valuations of γ's, (b) Simulating synchronous product $A_0 \times \mathtt{U}$, (c) Mincost Choreography C.

5.5.3 Realizability of cyclic goal

A goal service is specified as an input/output-automaton, $A_0 = (S_0, S_0^0, S_0^F, I_0, O_0, \Delta_0)$. Note that $I_0 = \cup_{n=1}^{N} I_n$ and $O_0 = \cup_{n=1}^{N} O_n$ (see Definition 21), i.e., the inputs/outputs of the goal are the union of the inputs/outputs of the existing services. States in S_0^F designate the final states, reaching which signifies the completion of a task. In contrast, being at a non-final state signifies a pending task. Since no further execution is possible beyond a deadlocking state (which has no outgoing transition), it is natural to require that a deadlocking state is a final state (otherwise certain tasks can never be completed).

Example 19 The goal A_0 given in Figure 5.9 is simulated by the `U` automaton in the Figure 5.11. Thus A_0 can be realized by choreographing the services A_1, A_2, A_3 of Figure 5.9.

Since, the size of $A_0 \times$ `U` is usually smaller compared to the size of `U` (since size of A_0 is smaller compared size of `U`), we verify, whether $A_0 \sqsubseteq A_0 \times$ `U` or not.

Example 20 Figure 5.12(b) shows the simulating synchronous product of the goal automaton A_0 in Figure 5.9 and the universal service automaton U in Figure 5.11. It can be seen from inspecting $A_0 \times$ `U` and A_0 that $A_0 \sqsubseteq A_0 \times$ `U` holds. Here, $A_0 \times$ `U` has two paths from the start state both of which can yield choreographers. From the associated histories in the paths it can be seen that one path uses automaton A_2 for computing the transitions `inf/`ϵ, `a/prc` as well as `c/`ϵ and other path uses service A_3 for the same. In the following sections we introduce the algorithm for choosing the optimal solution.

5.6 Optimum Decentralization

For optimal decentralization, we need to find out the composition where the overall communication and computation costs are minimized. This entails finding the optimal subautomaton as there might be several of them simulating the same sequence of inputs/outputs. Note that the cost of an automaton will be different than in Definition 18, since simply adding costs of transitions would amount to an infinity trace for paths having loops. The objective of optimization is to choreograph the existing services in such a way that regardless of the history of evolution, any pending task is completed in a minimal cost

Algorithm 3 (MinCost Choreography)

Initialization: $(k = 0)$

$\texttt{cost}^k(s_0, s_u) =$

$$\begin{cases} 0 \quad \text{if } (s_0, s_u) \text{ is a deadlocking state} \\ \underset{s_0 \xrightarrow{i/o} s_0' \in \Delta_0}{\texttt{MAX}} \left[\underset{(s_0,s_u) \xrightarrow[\bar{c},\bar{\gamma}]{i/o} (\bar{s_0},\bar{s_u}) \in \Delta_\times}{\texttt{MIN}} \left\{ \begin{array}{l} \{\bar{c} + \texttt{cost}^k(\bar{s_0}, \bar{s_u}) \mid (\bar{s_0}, \bar{s_u}) \notin S_0^F \times S_U\} \\ \cup \{\bar{c} \mid (\bar{s_0}, \bar{s_u}) \in S_0^F \times S_U\} \end{array} \right\} \right] \\ \quad \text{otherwise} \end{cases}$$

$\texttt{cost}^k = \texttt{MAX}\{\texttt{cost}^k(s_0, s_u) \mid (s_0, s_u) \in S_0 \times S_U\}$

Iteration: $(k \geq 1)$

$\texttt{cost}^k(s_0, s_u) =$

$$\begin{cases} 0 \quad \text{if } (s_0, s_u) \text{ is a deadlocking state} \\ \underset{s_0 \xrightarrow{i/o} s_0' \in \Delta_0}{\texttt{MAX}} \left[\underset{(s_0,s_u) \xrightarrow[\bar{c},\bar{\gamma}]{i/o} (\bar{s}_0,\bar{s}_u) \in \Delta_\times}{\texttt{MIN}} \left\{ \begin{array}{l} \{\bar{c} + \texttt{cost}^k(\bar{s_0}, \bar{s_u}) \mid (\bar{s_0}, \bar{s_u}) \notin S_0^F \times S_U\} \\ \cup \{\bar{c} \mid (\bar{s_0}, \bar{s_u}) \in S_0^F \times S_U, \forall m < k : \texttt{cost}^m(s_0, s_u) \neq \texttt{cost}^m\} \\ \cup \{\infty \mid (\bar{s_0}, \bar{s_u}) \in S_0^F \times S_U, \exists m < k : \texttt{cost}^m(s_0, s_u) = \texttt{cost}^m\} \end{array} \right\} \right] \\ \quad \text{otherwise} \end{cases}$$

$\texttt{cost}^k = \texttt{MAX}\{\texttt{cost}^k(s_0, s_u) \mid (s_0, s_u) \in S_0 \times S_U, \forall m < k : \texttt{cost}^m(s_0, s_u) \neq \texttt{cost}^m\}$

Termination: If $\texttt{cost}^k < \texttt{cost}^{k-1}$, set $k := k + 1$ and repeat Iteration step; otherwise stop and output $\texttt{cost}^{k-1}$.

Figure 5.13: Algorithm for computing minimum cost choreography

(the worst cost between any state and its nearest reachable final states is minimized). With this in mind, we define the cost of a subautomaton of $A_0 \times \mathtt{U}$ as follows.

Definition 23 (Cost of an Automaton) Given an i/o -automaton with each of its transitions labeled by some cost, we define the cost of a path to be the sum of the costs of all the transitions in that path. We define the cost of a state to be the maximum cost among all paths originating at that state and terminating at a final state. The cost of an automaton is defined to be the maximum cost among all its states.

We refer to the cost of an i/o-automaton A as $\mathtt{cost}(A)$. From Definition 23, we define the minimum cost choreography automaton for realizing A_0 from $\{A_n \mid 1 \leq n \leq N\}$ as defined in Definition 19

5.6.1 Computing optimum cost

In this section, we introduce the algorithm for computing a subautomaton of $A_0 \times \mathtt{U}$ for obtaining the optimal choreographer. Given a state (s_0, s_u) of $A_0 \times \mathtt{U}$, a certain goal transition $s_0 \xrightarrow{i/o} s_0'$ may be simulated by multiple transitions of the form $(s_0, s_u) \xrightarrow[\bar{c},\bar{\gamma}]{i/o} (\bar{s}_0, \bar{s}_u)$ in $A_0 \times \mathtt{U}$, and the algorithm given below selects the least expensive option. This is done by associating a cost with each state $(s_0, s_u) \in S_0 \times S_U$ that represents the worst cost among all paths, rooted at the state and ending at final states, needed to simulate the state s_0. The cost of states are assigned iteratively, and recursively within each iteration (see Algorithm 3).

In Algorithm 3 (Figure 5.13), the cost of a deadlocking state (s_0, s_u) is set to be zero as there is no computation or communication cost is incurred from such a state. The computation of the cost of a non-deadlocking state (s_0, s_u) in the initialization phase ($k = 0$) can be understood in two steps.

1. In the first step (minimization), we identify the cheapest way of simulating a goal transition $s_0 \xrightarrow{i/o} s_0'$ originating at s_0. This corresponds to a transition of $A_0 \times \mathtt{U}$ labeled by i/o having the least sum of the cost of the transition and the cost of its destination state. If the destination state is a final state, then it does not contribute any cost since reaching it results in the completion of a task.

2. In the second step (maximization), we identify the worst cost of simulating a transition of the type $s_0 \xrightarrow{i/o} s_0'$ originating at s_0.

The maximum cost of any state as computed in the initialization phase is denoted $\texttt{cost}^0$. It labels the cost of each state with the least cost to reach the nearest final state, simulating a goal path. Note that for acyclic paths, the final state will be deadlocking having a cost zero. For cyclic paths, another set of tasks start from the final states and will give assign it a non-zero cost.

The kth iteration ($k \geq 1$) explores the possibility of reducing the overall cost by way of avoiding reaching the worst-costing final states of the earlier iterations. Doing this results in raising the cost of all the other states (since they no longer have access to the worst-costing final states of the earlier iterations), but the overall cost may still reduce since the costs of the worst-costing states of the earlier iterations no longer affect the overall cost. The worst-costing final states of the earlier iterations are avoided by simply setting their cost contribution to be infinity. This is the only difference between the 0th iteration, and the subsequent iterations. A new iteration is executed only if the current iteration results in a reduction in the overall cost compared to the previous iteration.

Remark 1 The cost computation in each iteration is recursive for each $k \geq 0$ (the formula for state cost given in Algorithm 3 can be viewed to be of form: $\texttt{cost}^k(\cdot,\cdot) = fn(\{\texttt{cost}^k(\cdot,\cdot)\})$), and we seek to compute the least-fixed point of the recursion. Note the least-fixed point exists since the mapping given by $fn(\{\texttt{cost}^k(\cdot,\cdot)\})$ is non-decreasing and hence monotone. The least fixed point computation for any iteration $k \geq 0$ can be accomplished by initially assigning a zero cost to all the states, and next repeatedly updating their cost values by using the formula for $\texttt{cost}^k(\cdot,\cdot)$ given in Algorithm 3. The updates will be repeated at most $|S_0| \times |S_U|$ times, by which time they will either converge (to finite values), or continue to rise and in which case the overall cost will diverge (to infinity eventually). In either case, the recursive computation of the kth iteration can be terminated. The algorithm continues to a next iteration if $k = 0$ or the overall cost of the kth iteration is smaller than that of the previous iteration.

When there are no cycles between an initial or a final state and its nearest reachable final states (this will be the case for example when all cycles in the goal service possess a final state), the recursive computation of the kth iteration ($k \geq 0$) can be performed alternatively by maintaining a working-set

of states. The initial working-set will consist of the one-step backward reachable states of the final states, and each time the cost of all the states in the working set is computed using the formula of Algorithm 3, the working set will be enlarged to include its one-step backward reachable states. The recursive computation of the kth iteration will end when all states are included in the working-set. It can be concluded that the kth iteration will require $O(|S_0| \times |S_U|)$ number of computations. Further since in each iteration at least one final state is avoided (by forcing its cost contribution to be infinity), there can be at most $O(|S_0^F| \times |S_U|)$ number of iterations. Thus the overall computational complexity of Algorithm 3 is $O(|S_0| \times |S_0^F| \times |S_U|^2)$.

Remark 2 While Algorithm 3 works for general services and goals, a *finite-cost* optimum solution will exist if all cycles in the goal service either possess a final state or cost zero to simulate. (Otherwise the optimal cost will be ∞, and in which case any decentralized choreographer is an optimum one.)

Theorem 4 Given $A_0 \times \mathtt{U}$, where A_0 and $\mathtt{U}$ are goal and universal service automata respectively, Algorithm 3 in Figure 5.13 terminates with the cost equaling that of a mincost choreography subautomaton C of $A_0 \times \mathtt{U}$.

Proof: The proof is based on the facts that (a) in any iteration $k \geq 0$, the cost of a state is optimum under the constraint that the worst-costing final states of the earlier iterations are forced to remain unreachable, and (b) the iteration terminates when forcing the unreachability of the worst-costing final state of the last iteration ends up raising the overall cost.

Now we argue the correctness of assertion (a). In the kth iteration any state (s_0, s_u), that is not forced to become unreachable, must simulate all transitions defined at s_0. Hence the cost of (s_0, s_u) is the worst over all transitions defined at s_0 that must be simulated, which accounts for the maximization operation in the formula of Algorithm 3. For each transition defined at s_0, there may be multiple options to simulate it, and the least cost option is selected by the algorithm by choosing an appropriate successor $(\bar{s}_0, \bar{s}_u)$ such that the cost of the successor together with the cost of the transition from (s_0, s_u) to $(\bar{s}_0, \bar{s}_u)$ is minimized, which accounts for the minimization operation in the formula of Algorithm 3. If the successor $(\bar{s}_0, \bar{s}_u)$ is a final state, its cost contribution is zero if its reachability is allowed, and is infinity otherwise. Accordingly, the cost of (s_0, s_u) is the worst over all transitions at s_0 that must be

simulated, and for each such transition, best among all successors that can be reached to simulate that transition (where the best cost is given by the least combined cost of the transition together with the cost of the successor state). It is clear that for any iteration $k \geq 0$, the formula given in Algorithm 3 (in form of mapping $\texttt{cost}^k(\cdot,\cdot) = fn(\{\texttt{cost}^k(\cdot,\cdot)\})$) is a correct formula. Then the assertion (a) stated above follows from the fact that we compute the least-fixed point of the mapping (see Remark 1), and hence any cost value smaller than $\texttt{cost}^k(s_0, s_u)$ is not possible (given that $\texttt{cost}^k(s_0, s_u)$ must satisfy the formula of Algorithm 3 as established above).

(b) In the kth iteration, a state (s_0, s_u) becoming unreachable, means the cost valuation of that state is assigned from c to ∞. Consider the case, where there are other ways to simulate s_0. Then, in the $k+1$th iteration, the cost valuation of (s_0, s'_u) viz. c' and would contribute to the calculation of the overall cost of the choreography automaton. Clearly, $c' < c$, because (s_0, s_u) was made unreachable instead of (s_0, s'_u) in the previous iteration. Since, other cost valuations not depending on the new assignment remain the same, $\texttt{cost}^{k+1} < \texttt{cost}^k$. Hence iterations will continue. Since, there are a finite number of services, there will be finite number of ways to simulate a goal state s_0. Hence, after finite number of iterations, a state (s_0, s_u) will be forced to be unreachable, where there are no other ways to simulate s_0. Then the cost contribution of that node becomes ∞ and hence the Algorithm 3 will render the cost of the root node as ∞. This cost, viz. ∞ being higher than the previous iteration cost, will terminate the algorithm. This concludes the proof. ∎

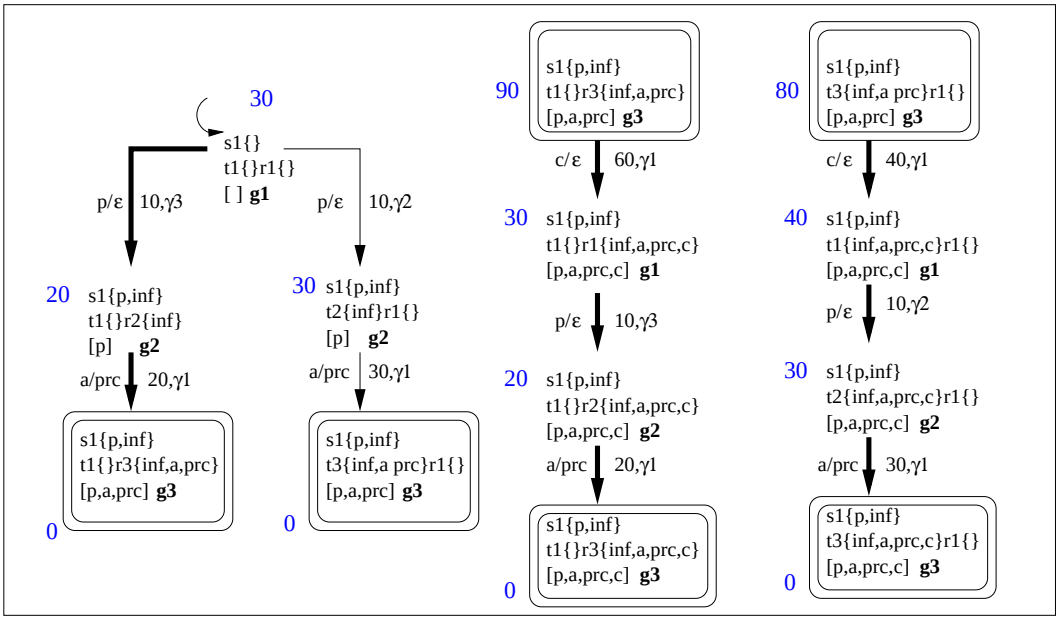
Example 21 Figure 5.14 depicts the computation of Algorithm 3 as applied to our running example. The simulating synchronous product $A_0 \times U$ is shown in Figure 5.12(b). As can be seen from Figure 5.9, the goal service possesses a cycle, and the cycle contains a final state. So in this case, the cost of completing a task starting from any state is finite (since the paths from any state to its nearest final states are cycle-free).

Notice that $A_0 \times U$ has one initial state, and two final states. Upon starting from the initial state, when a final state is reached a certain task of the goal is completed. If the final state reached is non-deadlocking, a new task begins from this state and its completion occurs when a subsequent final state is reached. Thus $A_0 \times U$ can execute three distinct tasks, starting from either the initial state or one of the final states. In order to simplify the illustration of the algorithm, the graph of $A_0 \times U$ is accordingly

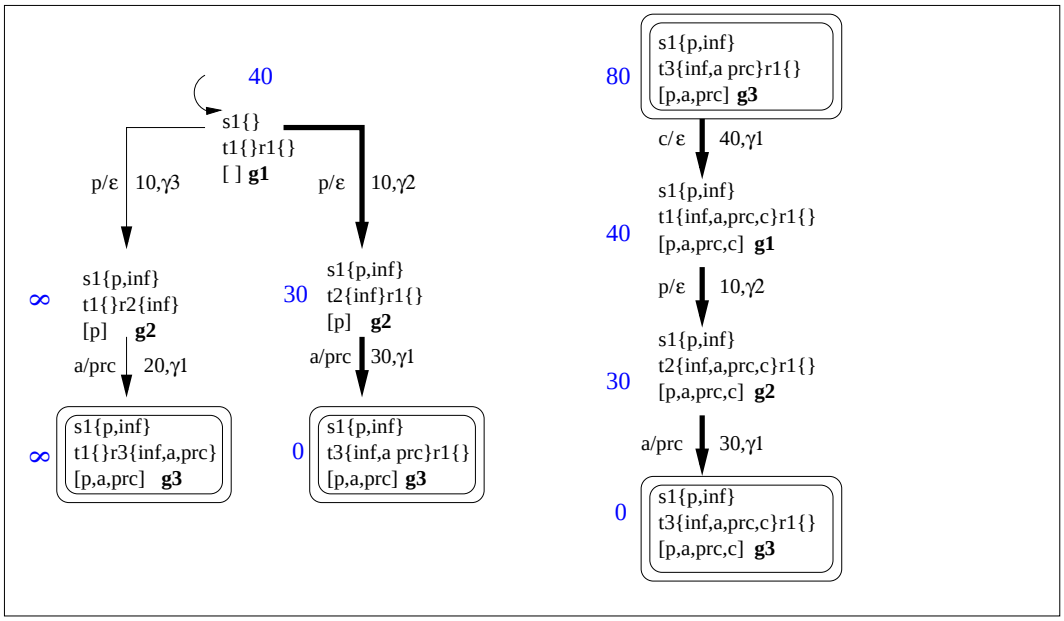
split into three subgraphs (that are trees), one for each task, as shown in the top half of Figure 5.14.

For $k = 0$, the cost of each state is computed using the formula of Algorithm 3 following the method discussed in Remark 1. The leaf nodes of the trees are final states representing completion of tasks. Since no further computation or communication is required once a task is completed, the leaf nodes are associated with zero costs. (Note in Algorithm 3, the fact that the cost contribution of leaf nodes is zero is implicitly encoded in the second term of the union appearing inside the minimization operation, in which any state cost is not included.) The state `s1{p,inf}t1{}r2{inf}[p]` is one-step backward reachable from the final states or the leaf nodes. Applying the formula given in Algorithm 3, we obtain the cost of `s1{p,inf}t1{}r2{inf}[p]` as 20. Similarly the cost of `s1{p,inf}t2{inf}r1{}[p]` is obtained to be 30. Next at the initial state `s1{}t1{}r1{}[]`, Algorithm 3 picks the least cost option to simulate the transition $\texttt{p}/\epsilon$ and labels the cost of the initial state as $20 + 10 = 30$ (the other option costs $30 + 10 = 40$ which is more expensive). The thickened edges represent the least expensive options. Also, in the 0th iteration, the cost of each final state, which represents the cost of executing another task starting from the corresponding final state, is computed. This is depicted by the cost labels of the other two trees of the initialization step. The two final states bear the costs of 90 and 80 respectively. Thus the overall cost is given by $\texttt{cost}^0 = \texttt{MAX}(30, 90, 80) = 90$.

For $k = 1$, the worst-costing node having the cost of 90 is avoided by forcing its cost contribution as a leaf node to be ∞ (as opposed to zero in the initialization step). Also since this node is to be avoided, it can no longer act as a restarting point of a task, and hence there are only two trees to consider in Iteration 1. The cost of each states is computed as before. In this iteration, the initial state has the cost of 40 since it is forced to select the the only available option to simulate the transition on $\texttt{p}/\epsilon$. This choice is depicted by the thickened edges. While the cost of the initial node rises, the overall cost drops and is given by $\texttt{cost}^1 = \texttt{MAX}(40, 80) = 80$. A reduction in the overall cost leads to execution of the next iteration ($k = 2$) (not shown in the figure). In this iteration the worst-costing state having the cost of 80 is avoided by forcing its cost contribution as a leaf node to be ∞. Also this node cannot serve as a restarting point of a task, and so in Iteration 2 there is only one tree (rooted at the initial state) to consider. As both its leaf nodes have a cost contribution of ∞, it is easy to see that $\texttt{cost}^2 = \infty$. Since $\texttt{cost}^2$ exceeds $\texttt{cost}^1$, Algorithm 3 terminates with the optimum solution provided by Iteration 1.



Initialization (k=0)



Iteration 1 (k=1)

Figure 5.14: Iterations of Algorithm 3

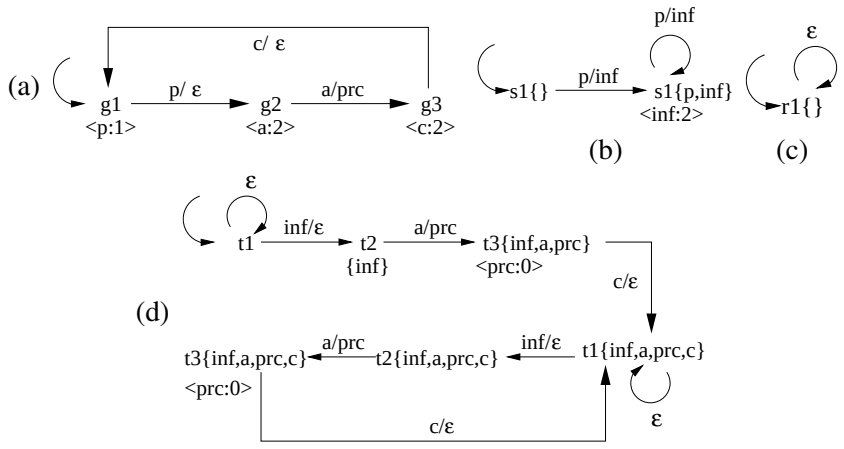


Figure 5.15: Site specific Choreography Automata: (a) C_0, (b) C_1, (c) C_3 and (d) C_2

Example 22 For our running example, the automaton C, shown in Figure 5.12(c), represents the optimum choreographer (from among the two candidate choreographers embedded in Figure 5.12(b)). One choreography scheme simulates the transition on $\mathtt{p}/\epsilon$ using A_1 and A_2 while the other scheme simulates the same transition using A_1 and A_3. Note that the two schemes cost differently, and our algorithm selects the least expensive option by selecting A_2..

5.6.2 Synthesizing optimum choreographers

Starting from a subautomaton C of $A_0 \times \mathtt{U}$ representing an optimum choreography scheme, our objective is to synthesize choreographers at each site such that the transduced closure of their product replicates C. In addition to normal i/o-behavior at each transition, a site-specific choreographer also needs to record information regarding the i/o's that must be sent from one choreographer to another for minimal cost communication. Site-specific choreographers are obtained using the mechanism described in Algorithm, 2.

Example 23 Figure 5.15(a, b, c, d) shows the various site-specific choreographers as obtained by applying Algorithm 3. The labeling of states as implied by the E_n function is shown within $<>$. In Figure 5.15(a), we obtain the choreographer C_0 at the client-site. C_0 communicates $\mathtt{p}$ to the choreographer C_1 at site-1, and $\mathtt{a}$ and $\mathtt{c}$ to the site-2 choreographer C_2 from the initial and the next successor

states, respectively. Other choreographers can be explained in similar fashion. They realize the goal in with minimum cost (as per cost tables in Figure 5.8(b)) as the choreographers are obtained from the minimum cost automaton in Figure 5.12(c).

CHAPTER 6. Conclusion and Future Work

6.1 Concluding Remarks

We have reduced the problem of verifying the existence of a choreographer to a simulation problem over i/o automata. The solution relies on the construction of appropriate universal-service automaton, $\mathtt{U}$-automaton or $\mathtt{U}^T$-automaton, as the case may be. One of the future avenues of research is to investigate applicability of local, on-the-fly algorithms to solve the problem. Such algorithms will explore the state-space of the universal-service automaton as and when needed and will stop exploration whenever the proof of existence of choreographer is obtained.

The implementation was done with a local and on-the-fly technique for verifying the existence choreographer and synthesizing it. This is achieved by logical encoding of choreographer-based composition problems and evaluating the logic program in a goal-directed fashion. The results prove that our technique is promising and can be used effectively in practical settings. One of the future avenues of research is to develop heuristics to further assist and guide local exploration of the composite services.

Then we addressed the problem of decentralizing the choreography of web services for achieving cost benefits. We have systematically formulated this problem and presented algorithms for finding an optimal decentralized scheme of choreography. The state space exploration of the Universal Automaton has been reduced by taking the simulating synchronous product with the goal. An algorithm for finding an optimum decentralized choreography scheme is presented that for every possible configuration minimizes the worst cost over all behaviors required to realize the goal service. Another algorithm computes the site-specific choreographers by distributing the optimum choreography scheme computed by the first algorithm. The paper presents a first automated technique for solving the optimum decentralized choreography problem, in which the goal service is allowed to have cycles. Thus, in the architectural framework shown in Figure 1.2, our scheme fits as the engine of the composition and as

constraints we considered optimizing a cost model.

6.2 Future Directions

Automated composition of Web Services is still to mature for wide acceptance in the industry. In future the prototype of optimization implemented can be applied to real scenarios with the possible additional handling of standard programming control constructs such as *for,if-then-else,while*. Also, this would entail conversion of standard composition languages such as WS-BPEL, WS-CDL to our automata models and reconversion of the composition result back to the standard language modes.

Based on the composition schemes as necessary in future, our theoretical approach can be extended for further advancements. The automata model can be strengthened with guard conditions emulating control flows. Our work assumes an exact match of inputs and outputs in the user specifications with the services available in the repository, whereas the user may want outputs which can be any subset of the available outputs from the services. The inputs provided can be more than what is required, in which case a subset of user inputs has to be matched. Another direction along the same lines is the partial realization of the user's goal when it cannot be met fully. Optimal partial realization of the goal would address the problem of constructing a composition which is least dissimilar with Goal, than other realizations possible from given services. A complete automation would also require semantic matching of service interfaces, as well as considering other aspects for communication such as security, service level agreements, which have also been a much pursued topic of research.

APPENDIX

Implementation Results

The optimal decentralization tool accepts a set of service specifications and the goal specification and if a choreographer exists gives as output the optimal decentralized realization of the goal.

The choreography scheme with acyclic goals first implements the composition algorithm which calculates the synchronous product of the Universal Automaton and the Goal. All the i/o automata are stored in graph data structures, classes are defined for states, and each state maintains transition labels leading to successors. We explore the each goal transition and search for its realization in the product in a depth-first manner. It maintains the distributed history for each state in a Set data structure along the realization of the transition. Once, all transitions of the goal are explored and realized, it is concluded that a choreographer exists. Note that, the product holds the intermediate states interleaved for the transduced realization of a transition. This is essential for extraction of the choreographer by projecting the interleaving to various service sites.

Then the optimization Algorithm 1 is applied to extract the optimal branch of the product. For this, we find the critical path determining the cost of the minimum cost choreography automaton. Maintaining the cost of the automaton, the other optimal paths are marked so that all goal transitions are simulated. The costlier paths to simulate are eliminated. This trims the automaton to yield the minimum cost choreography scheme.

Finally, applying Algorithm 2, we get the projection of the minimum cost choreography automaton on different sites. This yields the site-specific choreographers. Since the intermediate states for realizing a goal transition are stored in the graph for the automaton, extracting the site-specific automata amounts to finding the transitions specific to the concerned site in the optimal realization graph.

The service automata and the goal are encoded similar to what is described in 4.2.

```
service_trans(s, (a,b), s'),c.
```

In the above service transition, `s` signifies the start state, `a`, the input of the transition, `b`, the output, `s'`, the destination state and `c` the computation cost. The communications cost `C` between site `A` and `B` are listed as `(A,B=C)`. Here is the encoded input of the example used earlier.

```
%% service 1 Address
service_trans(s1, (nP,aP), s1),1.
start_service(s1).
end_service

%% service 2 Address
service_trans(t1, (nP, aP), t1),1.
service_trans(t1, (nC, aC), t1),1.
start_service(t1).
end_service

%% service 3 Route
service_trans(r1, (aP, eps), r2),1.
service_trans(r2, (aC, r), r3),1.
start_service(r1).
end_service

%% goal
service_trans(g1, (nP, eps), g2).
service_trans(g2, (nC, r), g3).
start_goal(g1).
end_goal

start_communication_costs
(0,1=5)(0,2=10)(0,3=20)
(1,0=5)(1,2=10)(1,3=15)
(2,0=10)(2,1=10)(2,3=15)
(3,0=20)(3,1=15)(3,2=15)
```

```
end_communication_costs
```

The output is as follows:

```
A0 X U:

<2> t1{[]}s1{[aP, nP]}r2{[aP, eps]}g2{[eps, nP]}{42} -nC(from:0)/aC(1)->
    <5> t1{[aC, nC]}s1{[aP, nP]}r2{[aP, eps]}g2g3{[eps, nP, nC]}

<4> t1{[aP, nP]}s1{[]}r2{[aP, eps]}g2{[eps, nP]}{47} -nC(from:0)/aC(1)->
    <7> t1{[aC, aP, nP, nC]}s1{[]}r2{[aP, eps]}g2g3{[eps, nP, nC]}

<8> t1{[aC, aP, nP, nC]}s1{[]}r3{[aC, aP, r, eps]}g3{[r, eps, nP, nC]}{94}

<6> t1{[aC, nC]}s1{[aP, nP]}r3{[aC, aP, r, eps]}g3{[r, eps, nP, nC]}{89}

<1> t1{[]}s1{[aP, nP]}r1{[]}g1g2{[nP]}{6} -aP(from:1)/eps(1)->
    <2> t1{[]}s1{[aP, nP]}r2{[aP, eps]}g2{[eps, nP]}

<3> t1{[aP, nP]}s1{[]}r1{[]}g1g2{[nP]}{11} -aP(from:2)/eps(1)->
    <4> t1{[aP, nP]}s1{[]}r2{[aP, eps]}g2{[eps, nP]}

<7> t1{[aC, aP, nP, nC]}s1{[]}r2{[aP, eps]}g2g3{[eps, nP, nC]}{58} -aC(from:2)/r(1)->
    <8> t1{[aC, aP, nP, nC]}s1{[]}r3{[aC, aP, r, eps]}g3{[r, eps, nP, nC]}

<5> t1{[aC, nC]}s1{[aP, nP]}r2{[aP, eps]}g2g3{[eps, nP, nC]}{53} -aC(from:2)/r(1)->
    <6> t1{[aC, nC]}s1{[aP, nP]}r3{[aC, aP, r, eps]}g3{[r, eps, nP, nC]}

<0> t1{[]}s1{[]}r1{[]}g1{[]}{0} -nP(from:0)/aP(1)->
    <1> t1{[]}s1{[aP, nP]}r1{[]}g1g2{[nP]}
<0> t1{[]}s1{[]}r1{[]}g1{[]}{0} -nP(from:0)/aP(1)->
    <3> t1{[aP, nP]}s1{[]}r1{[]}g1g2{[nP]}

Optimal Choreography:
```

```
<5> t1{[aC, nC]}s1{[aP, nP]}r2{[aP, eps]}g2g3{[eps, nP, nC]}{53} -aC(from:2)/r(1)->
    <6> t1{[aC, nC]}s1{[aP, nP]}r3{[aC, aP, r, eps]}g3{[r, eps, nP, nC]}

<1> t1{[]}s1{[aP, nP]}r1{[]}g1g2{[nP]}{6} -aP(from:1)/eps(1)->
    <2> t1{[]}s1{[aP, nP]}r2{[aP, eps]}g2{[eps, nP]}

<6> t1{[aC, nC]}s1{[aP, nP]}r3{[aC, aP, r, eps]}g3{[r, eps, nP, nC]}{89}

<2> t1{[]}s1{[aP, nP]}r2{[aP, eps]}g2{[eps, nP]}{42} -nC(from:0)/aC(1)->
    <5> t1{[aC, nC]}s1{[aP, nP]}r2{[aP, eps]}g2g3{[eps, nP, nC]}

<0> t1{[]}s1{[]}r1{[]}g1{[]}{0} -nP(from:0)/aP(1)->
    <1> t1{[]}s1{[aP, nP]}r1{[]}g1g2{[nP]}

Decentralized Choreographers:

<t1:> {[]} -nC/aC(1)-> <t1:3> {[aC, nC]}
<t1:3> {[aC, nC]}

<s1:> {[]} -nP/aP(1)-> <s1:3> {[aP, nP]}
<s1:3> {[aP, nP]}

<r1:> {[]} -aP/eps(1)-> <r2:0> {[aP, eps]}
<r2:0> {[aP, eps]} -aC/r(1)-> <r3:0> {[aC, aP, r, eps]}
<r3:0> {[aC, aP, r, eps]}

<g1:1> {[]} -nP/eps()-> <g2:2> {[eps, nP]}
<g2:2> {[eps, nP]} -nC/r()-> <g3:> {[r, eps, nP, nC]}
<g3:> {[r, eps, nP, nC]}
```

BIBLIOGRAPHY

[1] The XSB Logic Programming System, Dept. of Computer Science, SUNY at Stony Brook, Available from *http://xsb.sourceforge.net*.

[2] Universal Description, Discovery and Integration, *www.uddi.org*.

[3] Web Service Description Language, *www.w3.org/TR/wsdl*.

[4] Web Service Use Cases, *http://www.w3.org/TR/ws-arch-scenarios*.

[5] Simple Object Access Protocol, *www.w3.org/TR/soap*.

[6] Web Services Modeling Ontology, *www.wsmo.org/*.

[7] Semantic Web enabled Web Services, *www.swws.org/*.

[8] Web Service Ontology *http://www.daml.org/services/owl-s/*

[9] Fensel, D., Bussler, C. The Web Service Modeling Framework WSMF. Eletronic Commerce: Research and Applications. Vol. 1. pages 113-137, 2002.

[10] K. S. May Chan, Judith Bishop and Luciano Baresi, Survey and Comparison of Planning Techniques for Web Services Composition, Technical Report, February, 2007.

[11] J. Peer, Web Service Composition as AI Planning - a Survey, Technical Report, University of St. Gallen, 2005.

[12] J. Rao, Semantic Web Service Composition via Logic-based Program Synthesis, PhD Thesis, Norwegian University of Science and Technology, 2004.

[13] R. Hull and J. Su, Tools for Composite Web Services: A Short Overview, journal of ACM SIGMOD Record, 34,2, June, 2005.

[14] Saayan Mitra, Ratnesh Kumar and Samik Basu, Automated Choreographer Synthesis for Web Services Composition Using I/O Automata, IEEE International Conference on Web Services, pages 364-371, 2007.

[15] Saayan Mitra, Samik Basu and Ratnesh Kumar, Local and On-the-fly Choreography based Web Service Composition, IEEE/ACM International Conference on Web Intelligence, pages 521-527, 2007.

[16] Saayan Mitra, Ratnesh Kumar and Samik Basu, Optimum Decentralized Choreography for Web Service Composition, IEEE International Services Computing Conference, pages 395-402, 2008.

[17] Saayan Mitra, Ratnesh Kumar and Samik Basu, A Framework for Optimum Decentralized Choreography, *Submitted to* IEEE International Conference on Web Services, 2009.

[18] Saayan Mitra, Ratnesh Kumar and Samik Basu, Choreography based Web Service Composition with I/O Automata, *Submitted to* Journal of Web Intelligence and Agent Systems, 2009.

[19] D. Berardi, D. Calvanese, G. Giacomo, R. Hull, M. Mecella, Automatic Composition of Transition-based Semantic Web Services with Messaging, Conference on Very Large Databases, pages 613-624, 2005.

[20] D. Berardi, G. Giacomo, M. Mecella and D. Calvanese, Composing Web Services with Nondeterministic Behavior, IEEE International Conference on Web Services, pages 909-912, 2006.

[21] T. Bultan, X. Fu, R. Hull and J. Su, Conversation Specification: A New Approach to Design and Analysis of e-service Composition, World Wide Web Conference, pages 403-410, 2003.

[22] X. Fu, T. Bultan and J. Su, Analysis of Interacting BPEL Web Services, World Wide Web Conference, pages 621-630, 2004.

[23] WS Business Process Execution Language, *www.oasis-open.org*.

[24] B. Srivastava and J. Koehler, Planning with Workflows - An Emerging Paradigm for Web Service Composition, ICAPS Workshop on Planning and Scheduling for Web and Grid Services, 2004.

[25] R. Akkiraju, B. Srivastava, A. Ivan, R. Goodwin and T. Syeda-Mahmood, SEMAPLAN: Combining Planning with Semantic Matching to Achieve Web Service Composition, IEEE International Conference on Web Services, pages 37-44, 2006.

[26] P. Traverso and M. Pistore, Automated Composition of Semantic Web Services into Executable Processes, International Semantic Web Conference, pages 380-394, 2004.

[27] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri and P. Traverso, MBP: a Model Based Planner, Workshop on Planning under Uncertainty and Incomplete Information, 2001.

[28] S. McIlraith and T.C. Son, Adapting Golog for Composition of Semantic Web Services, International Conference on the Principles of Knowledge Representation and Reasoning, pages 482-496, 2002.

[29] B. Carminati,E. Ferrari and P. Hung, Security Conscious Web Service Composition, IEEE International Conference on Web Services, pages 489-496, 2006.

[30] S. Narayanan and S. McIlraith, Simulation, Verification and Automated Composition of Web Services, International Conference on World Wide Web, pages 77-88, 2002.

[31] J. Rao, P. Kungas and M. Matskin, Logic-based Web services Composition: from Service Description to Process Model, IEEE International Conference on Web Services, pages 446-453, 2004.

[32] A. Deutsch, L. Sui and V. Vianu, Specification and Verification of Data-driven Web Services, ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems, pages 71-82, 2004.

[33] D. Berardi, D. Calvanese, G. Giacomo, M. Lenzerini and M. Mecella, Automatic Composition of e-services that export their behavior, International Conference on Service Oriented Computing, pages 43-58, 2003.

[34] C. Gerede, R. Hull, O. Ibarra and J. Su, Automated Composition of e-services: Lookaheads, International Conference on Service Oriented Computing, pages 252-262, 2004.

[35] M. Carman and C. Knoblock, Inducing Source Descriptions for Automated Web Service Composition, Workshop on Exploring Planning and Scheduling for Web Services, Grid, and Autonomic Computing, 2005.

[36] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam and Q. Sheng Quality Driven Web Services Composition, International Conference on World Wide Web, pages 411-421, 2003.

[37] J. Yang and M. Papazoglou, Web Component: A Substrate for Web Service Reuse and Composition, International Conference on Advanced Information Systems Engineering, pages 21-36, 2002.

[38] B. Medjahed, A. Bouguettaya and A. Elmagarmid, Composing Web services on the Semantic Web, The International Journal on Very Large Data Bases", 12(4), pages 333-351, 2003.

[39] W. Binder, I. Constantinescu and B. Faltings, Decentralized Orchestration of Composite Web Services, IEEE International Conference on Web Services, pages 869-876, 2006.

[40] G. Chafle, S. Chandra, V. Mann and M.G. Nanda, Decentralized Orchestration of Composite Web Services, International World Wide Web Conference on Alternate track papers and posters, pages 134-143, 2004.

[41] L. Zeng, B. Benatallah, G. Xie and H. Lei, Semantic Service Mediation, International Conference on Service-Oriented Computing, pages 490-495, 2006.

[42] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau and P. Traverso, Planning and Monitoring Web Service Composition, Conference on Artificial Intelligence:Methodology, Systems, Applications, pages 106-115, 2004.

[43] S. Thakkar, C. Knoblock and J. Ambite, A View Integration Approach to Dynamic Composition of Web Services, ICAPS Workshop on Planning for Web Services, 2003.

[44] M. Sheshagiri, M. desJardins and T. Finin, A Planner for Composing Services Described in DAML-S, International Conference on Automated Planning and Scheduling, 2003.

[45] S. McIlraith, T. Son and H. Zeng, Semantic Web Services, journal of IEEE Intelligent Systems, 16(2), pages 46-53, 2001.

[46] B. Benatallah, F. Casati, D. Grigori, H.R.M. Nezhad and F. Toumani, Developing Adapters for Web Services Integration, International Conference on Advanced Information Systems Engineering, pages 415-429, 2005.

[47] B. Benatallah, Q. Sheng and M. Dumas, The Self-Serv Environment for Web Services Composition, journal of IEEE Internet Computing, 7(1), pages 40-48, 2003.

[48] T. Bultan, J. Su, and X. Fu, Analyzing Conversations of Web Services, journal of IEEE Internet Computing, 10(1), pages 18-25, 2006.

[49] R. Hull, M. Benedikt, V. Christophides and J. Su, E-services: A Look Behind the Curtain, ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 1-14, 2003.

[50] M. Pistore, P. Roberti and P. Traverso, Process-level composition of executable Web Services: on-the-fly versus once-for-all composition, European Semantic Web Conference, pages 62-77, 2005.

[51] M. Trainotti, M. Pistore, G. Calabrese, G. Zacco, G. Lucchese, F. Barbon, P. Bertoli and P. Traverso, ASTRO: Supporting Composition and Execution of Web Services, International Conference on Service Oriented Computing, pages 495-501, 2005.

[52] X. Fu, T. Bultan and J. Su, WSAT: A Tool for Formal Analysis of Web Services", International Conference on Computer Aided Verification, pages 510-514, 2004.

[53] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini and M. Mecella, A Tool for Automatic Composition of Services based on Logics of Programs, International Workshop on Technologies for E-Services, pages 80-94, 2004.

[54] G.J.Holzmann, The Model Checker SPIN, journal of IEEE Transactions on Software Engineering, 23(5), pages 279-295, 1997.

[55] R. Milner, A Calculus of Communicating Systems, Springer-Verlag New York, Inc., 1982.

[56] J. Pathak, S. Basu, R. Lutz and V. Honavar, Parallel Web Service Composition in MoSCoE: A Choreography-Based Approach, Proceedings of the European Conference on Web Services, pages 3-12, 2006.

[57] H.R.M. Nezhad, B. Benatallah, F. Casati and F. Toumani, Web services Interoperability Specifications, Journal of Computer, 39(5), pages 24-32, 2006.

[58] J. Peer, A PDDL based Tool for Automatic Web Service Composition, Workshop on Principles and Practice of Semantic Web Reasoning, pages 149-163, 2004.

[59] R.J. Waldinger, Web Agents Cooperating Deductively, International Workshop on Formal Approaches to Agent-Based Systems, pages 250-262, 2000.

[60] S. Lammermann, Runtime Service Composition via Logic-Based Program Synthesis, Ph.D. Dissertation, Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden, 2002.

[61] S.B. Mokhtar, N. Georgantas and V. Issarny, Ad Hoc Composition of User Tasks in Pervasive Computing Environments, International Workshop on Software Composition, pages 31-46, 2005.

[62] B. Mitchell and J. Hillston, Analysing web service composition with PEPA, Third Workshop on Process Algebras and Stochastically Timed Activities, pages 33-44, 2004.

[63] A. Bartoli, R. Jimnez-Peris, B. Kemme, C. Pautasso, S. Patarin, S. Wheater and S. Woodman, The ADAPT Framework for Adaptable and Composable Web Services, IEEE Distributed Systems Online, 2005.

[64] A. Martinez, M. Patiño-Martinez, R. Jiménez-Peris and F. Perez-Sorrosal, ZenFlow: a visual Web service composition tool for BPEL4WS, IEEE Symposium on Visual Languages and Human-Centric Computing, pages 181-188, 2005.

[65] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar and J. Miller, METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services, Journal of Information Technology and Management, 6(1), pages 17-39, 2005.

[66] D. Wu, B. Parsia, E. Sirin, J. Hendler and D. Nau, Automating DAML-S Web Services Composition Using SHOP2, International Semantic Web Conference, pages 195-210, 2003

[67] B. Orriëns, J. Yang and M. Papazoglou, ServiceCom: A Tool for Service Composition Reuse and Specialization, pages 355-358, 2003.

[68] L. Cabral, J. Domingue, E. Motta, T. Payne and F. Hakimpour Approaches to Semantic Web Services: an Overview and Comparisons in The Semantic Web: Research and Applications, Volume 3053, pages 225-239, 2004.

[69] H. Foster, S. Uchitel, J. Magee and J. Kramer, Model-based Verification of Web Service Compositions, IEEE Conference on Automated Software Engineering, pages 152-163, 2003.

[70] S.R. Ponnekanti and A. Fox, SWORD: A Developer Toolkit for Web Service Composition, World Wide Web Conference, 2002.

[71] M. Bravetti and G. Zavattaro, Towards a Unifying Theory for Choreography Conformance and Contract Compliance, International Symposium on Software Composition, pages 34-50, 2007.

[72] H. Yang, X. Zhao, C. Cai and Z. Qiu, Exploring the Connection of Choreography and Orchestration with Exception Handling and Finalization/Compensation, Formal Techniques for Networked and Distributed Systems, 4574, pages 81-96, 2007.

[73] M. Carbone, K. Honda and N. Yoshida, Structured Communication-Centred Programming for Web Services, European Symposium on Programming, pages 2-17, 2007.

[74] L. Penserini, A. Perini, A. Susi and J. Mylopoulos, From Stakeholder Intentions to Software Agent Implementations, International Conference on Advanced Information Systems Engineering, pages 465-479, 2006.

[75] M.G. Nanda, S. Chandra and V. Sarkar, Decentralizing execution of composite web services, ACM SIGPLAN Conference on Object-oriented programming, systems, languages and applications, pages 170-187, 2004.

[76] P. Muth, D. Wodtke, J. Weißenfels, A.K. Dittrich and G. Weikum, From centralized work flow specification to distributed work flow execution, Journal of Intelligent Information Systems, 10(2), pages 159-184, 1998.

[77] J. Biskup, B. Carminati, E. Ferrari, F. Muller and S. Wortmann, Towards Secure Execution Orders for Composite Web Services, IEEE International Conference on Web Services, pages 489-496, 2007.

[78] F. Goethals, M. Snoeck, W. Lemahieu and J. Vandenbulcke, Considering (de)centralization in a Web Services World, Second International Conference on Internet and Web Applications and Services, 22, 2007.

[79] U. Yildiz and C. Godart, Synchronization Solutions for Decentralized Service Orchestrations, Second International Conference on Internet and Web Applications and Services, 39, 2007.

[80] J. Balasooriya, Distributed Web Service Coordination for Collaborative Applications and Biological Workflows, Georgia State University, 2006.

[81] R. Fielding, Architectural Styles and the Design of Network-based Software Architectures, University of California at Irvine, 2000.

[82] J. Balasooriya, J. Joshi, S.K. Prasad and S. Navathe, Distributed Coordination of Workflows over Web Services and Their Handheld-Based Execution, International Conference on Distributed Computing and Networking, pages 39-53, 2008.

[83] L. Baresi, A. Maurino and S. Modafferi, Towards Distributed BPEL Orchestration, Electronic Communications of the EASST, 3, 2006.

[84] A. Barker, J. Weissman and J. van Hemert, Orchestrating Data-Centric Workflows, IEEE International Symposium on Cluster Computing and the Grid, pages 210-217, 2008.

[85] M. Zaremba, T. Vitvara and M. Moran, Towards Optimized Data Fetching for Service Discovery, European Conference on Web Services, pages 191-200, 2007.

[86] A. Barker, J. Weissman and J. van Hemert, Eliminating The Middleman: Peer-to-Peer Dataflow, International Symposium on High-Performance Distributed Computing, pages 191-200, 2008.

[87] Ricardo Jiménez-Peris and Marta Patiño-Martínez and Ernestina Martel-Jordán, Decentralized Web Service Orchestration: A Reflective Approach, ACM Symposium on Applied Computing, pages 494-498, 2008.

[88] X. Qiao and J. Wei, A Decentralized Services Choreography Approach for Business Collaboration, IEEE Services Computing Conference, pages 190-197, 2006.

[89] U. Yildiz and C. Godart, Centralized versus Decentralized Conversation-based Orchestrations, The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services, pages 289-296, 2007.