Currently, more and more distributed systems are built based on the service oriented architecture (SOA), which can be implemented using Web service technologies. The benefits of SOA include better interoperability, scalability, maintainability and reuse. Service composition techniques allow creating new services by integrating existing services programmatically. They are instrumental in realizing the full potential of SOA. A typical service is often stateful: it offers multiple operations, and invoking these operations must be properly choreographed to achieve the desired result. The choreography rules can be captured and specified by a behavioral interface, also called service conversation protocol. A conversation protocol is important for a stateful service to be utilized properly. Incorporation of stateful services into a service composition poses interesting challenges, such as how to extend the current service description scheme to allow specification of conversation protocols, and how to utilize the service description correctly and efficiently. In this dissertation, we present a design and analysis framework for service oriented distributed systems. The cornerstone of the framework is a unified service specification model based on Colored Petri nets. The service specification model is suitable for both service compositions and conversation
protocols, and thus it supports incorporating partners exhibiting complex protocols. It also enables formal analysis of compositions: (1) verification of the correctness; (2) automatic derivation of the conversation protocol for the newly-created composite service. The framework consists of a modeling and analysis toolkit supporting the composition model, a translator between one of the standard composition languages and the model, a novel technique to derive conversation protocol automatically from composition, and a visual composition designer that integrates design and analysis components. The framework supports service composition involving partner services with complex conversation protocols.

**INDEX WORDS:** Service Oriented Architecture, Colored Petri Nets, Web Services, Web service Composition, Conversation Protocol.
A CPNETS-BASED DESIGN AND ANALYSIS FRAMEWORK FOR
SERVICE ORIENTED DISTRIBUTED SYSTEMS

by

XIAOCHUAN YI

B.E., Shanghai Jiao Tong University, 1996

A Dissertation Submitted to the Graduate Faculty of The
University of Georgia in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2005
A CPNET BASED FRAMEWORK FOR SPECIFICATION, DESIGN AND ANALYSIS OF
SERVICE ORIENTED DISTRIBUTED SYSTEMS

by

XIAOCHUAN YI

Major Professor: Krys Kochut
Committee: John A. Miller
           Eileen Kraemer
           Jaxk Reeves

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2005
ACKNOWLEDGEMENTS

I thank Dr. Kochut for his advisement and encouragement over the past five years. He has always generously given his time and has consequently had to sit through countless discussion of research ideas. He always did so with the patience of a great mentor. His broad knowledge and sharp insight have been the ideal that I have aspired to achieve in my study and future career; hopefully this dissertation is an evidence of that. Dr. Miller, Dr. Kramer, Dr. Cai, Dr. Arabnia and Dr. Sheth have also been great influences on me during my time at UGA. Finally, I thank Dr. Reeves for kindly giving his time to be on my committee.

Of course, I also owe a great deal of thanks to my family. I would have never even started my Ph.D. were it not for their enthusiastic support of my education. But, more importantly, they have always supported me in all things not related to this dissertation, and for that I am extremely grateful.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS..............................................................................................................................iv

LIST OF FIGURES .......................................................................................................................................vii

CHAPTER 1. INTRODUCTION ....................................................................................................................1

CHAPTER 2. DISTRIBUTED SYSTEMS AND SERVICE ORIENTED ARCHITECTURE . 4

2.1. Introduction......................................................................................................................................4

2.2. Web Services and Service Oriented Architecture ........................................................................6

2.3. Literature Review ..........................................................................................................................10

CHAPTER 3. MOTIVATING EXAMPLES ...............................................................................................15

3.1. Faulty BPEL4WS Code: Insurance Claim Processing.............................................................15

3.2. Incorporating Complex Partners: Composing Travel Agent Service .....................................16

CHAPTER 4. THE UNIFIED SERVICE SPECIFICATION MODEL..................................................20

4.1. Petri Nets and Their Variants.......................................................................................................20

4.2. Verification and Occurrence Graph ............................................................................................26

4.3. WSPNet ........................................................................................................................................28

4.4. Generating BPEL4WS Code ........................................................................................................29

4.5. Specification of Service Conversation Protocol .........................................................................29

4.6. Specification of Compositions .....................................................................................................34

CHAPTER 5. CONVERSATION PROTOCOL DERIVATION ............................................................37

CHAPTER 6. CPNETS REPRESENTATIONS OF BPEL4WS COMPOSITIONS ..................44

6.1. Parallel Split and Synchronization ............................................................................................47

6.2. Exclusive Choice and Simple Merge .........................................................................................48
LIST OF FIGURES

Figure 1. The model of basic Service Oriented Architecture ..............................................................6
Figure 2. Basic Web Service Protocols Stack ......................................................................................8
Figure 3. UML Activity Diagram of a faulty BPEL4WS process ..........................................................15
Figure 4. The problem of composing a Travel Agent service ............................................................16
Figure 5. A service whose conversation protocol can not be expressed with automata .............30
Figure 6. WSPNet for the conversation protocol of Airline service ................................................32
Figure 7. Conversation protocol in XML format: airlineProtocol.xml ............................................33
Figure 8. WSPNet for Travel Agent Service composition ..............................................................36
Figure 9. Service Conversation Protocol Derivation Algorithm ....................................................40
Figure 10. Service Transition Automaton ..........................................................................................41
Figure 11. When op is of request-response type .............................................................................41
Figure 12. When op is solicit-response type ....................................................................................42
Figure 13. When op is of notification type .......................................................................................42
Figure 14. When op is of one-way type ............................................................................................42
Figure 15. CPNet model for Sequence construct in BPEL ............................................................47
Figure 16. Representing parallel <flow> using CPNets .................................................................48
Figure 17. CPNet model for Exclusive Choice and Simple Merge .................................................50
Figure 18. CPNet model for BPEL4WS <while> construct ............................................................51
Figure 19. CPNet model for <pick> construct ..................................................................................52
Figure 20. CPNet model for serializable scope ..............................................................................54
Figure 21. UML Class Diagram for JCPNet .................................................................60
Figure 22. An abstract O-graph construction algorithm .............................................64
Figure 23. O-Graph construction algorithm in Java-like pseudo code ....................65
Figure 24. Get all bindings of a transition ..................................................................66
Figure 25. Set.xproduct .............................................................................................67
Figure 27. CPNet model of dining philosophers .........................................................68
Figure 28. CPNet model of distributed database ........................................................69
Figure 29. CPNet model of resource allocation scheme A ..........................................70
Figure 30. Occurrence graph of the CPNet model for resource allocation ..................72
Figure 31. Allocate all resources at the same time to a process. No deadlocks ..........72
Figure 32. Allocate resources (one resource to one process at a time) in the same other to each process. No deadlocks .................................................................73
Figure 33. A screen shot of WSPNet designer ..........................................................76
Figure 34. The initial design of insurance claim process ...........................................77
Figure 35. Generating CPNet OGraph analysis report ..............................................78
Figure 36. Occurrence graph of claim processing .....................................................79
Figure 37. A block-structured process design for insurance claim process ................81
Figure 38. OGraph Analysis report for black-structured process .............................82
Figure 39. The conversation protocol of the Airline service ......................................84
Figure 40. Incorporating a partner – choosing a partner ..........................................85
Figure 41. Incorporating a partner – partner protocol just incorporated ...................86
Figure 42. Incorporating a partner - completed .........................................................87
Figure 43. WSPNet model of the derived conversation protocol ...............................88
Figure 44. The first example of incorporation error: partner session is over, composition process hangs. ......................................................................................................................................90

Figure 45. Incorporation error detected by the designer .................................................................................91

Figure 46. The second example of incorporation error: invoking operation out of protocol synchronization .......................................................................................................................92

Figure 47. Out-of-synch incorporation error detected by the designer ..........................................................93
CHAPTER 1. INTRODUCTION

Service oriented architecture (SOA) is an architectural style that represents software functionality as services that can be discovered and invoked on the network. A service is a well-defined and self-contained function that does not depend on the state of other services. The functions performed by services can be anything from simple requests to sophisticated business processes. Services are often used as self-describing, platform-agnostic computational elements to support rapid, low-cost composition of distributed applications. Services also enable organizations to expose their functionalities programmatically over an intranet (or the Internet) using standard languages, protocols, and self-describing interfaces [Papazoglou 2003].

As a natural consequence of the evolution of the Web, the Web service paradigm is a collection of technologies including SOAP [Martin Gudgin 2003], WSDL [Erik Christensen 2001], UDDI [Luc Clement 2004]. It can be utilized to implement SOA. The Web service paradigm is suitable for constructing loosely coupled distributed systems, which resolves the problems of the CORBA or DCOM implementation of SOA. The goal of Web service is to provide a flexible framework facilitating universal interoperability as well as efficient integration of applications. The recently proposed standard Business Process Execution Language for Web Services (BPEL4WS) [Tony Andrews 2003] is an XML-based effort to create new Web services in the form of compositions of existing services. A BPEL4WS process is specified by referencing and interlinking portTypes defined in the WSDL definitions of partner services involved in the process. The Web service paradigm is the next step of development of distributed computing.
In order for Web services to realize their full potential and become widely accepted, researchers must address a number of important challenges. These challenges include various aspects of Web services composition, security of message exchanges, availability of transactional services, service description and discovery, and problems related to semantic Web services. Many of these challenges have already received some attention:

- Extending SOAP with security mechanisms and establishing a framework for secure interactions between Web services has been proposed in [IBM 2002] [Bob Atkinson 2002].
- Research efforts aimed at establishing a standard and interoperable way of achieving an acceptable and defined level of reliability at the SOAP messaging level, as well as a common vocabulary for describing reliable message exchange patterns have been reported in [Colleen Evans 2003].
- Traditional transactional models developed for distributed systems are not directly applicable to Web services, because Web services are loosely coupled applications. In addition, Web service operations are typically long running. Therefore, Web services need a specialized transaction model, as reported in [Luis Felipe Cabrera 2004].
- DAML-S (now called OWL-S) provides a comprehensive Web Service description, specifying its capability description in the service profile. A service capability matchmaking extension to the UDDI registry has been proposed in [M. Paolucci 2002]. The DAML-S process model provides a way to describe the conversation (interaction) protocol of the service process. In WSCL [Arindam Banerji 2002] the conversation protocol is defined using the Finite State Machine model, while WSCI and the abstract process in BPEL4WS describe the conversation
protocol using the workflow model [Tony Andrews 2003] [Assaf Arkin 2002]. The IBM conversation policy cpXML supports dynamic Web service conversations [J.E. Hanson 2002].

- Researchers have been involved in extending the vision of Semantic Web [T. Berners-Lee 2001] to the area of Web services. This novel research direction aims to develop methods for description, discovery, and composition of Web services based on semantic description of the services they offer. Initial research results in this area have been reported in [K. Sivashanmugam 2003] [M. Paolucci 2003]. Some latest research work, METEOR project, can be found in [Abhijit Patil, 2004].

- Other problems facing the researchers in the area of Web services are related to the lack of a clear methodology for designing Web services and their compositions. The current Web service composition languages do not offer any guidelines for locating and selecting component services [M. Paolucci 2003], satisfying the local and global functional and non-functional requirements [L. Zeng 2003], or how the business rules should be captured in the service composition [B. Orriens 2003].

This dissertation is organized as follows: in Chapter 2, we give an overview of distributed systems and service oriented architecture; in Chapter 3, we introduce several motivating examples regarding service description and composition; in Chapter 4, we propose the unified service specification model; in Chapter 5, we discuss the conversation protocol derivation algorithm; in Chapter 6, we describe the JCPNet toolkit and its application to service modeling and analysis; in Chapter 7, we discuss the translation between BPEL4WS and the unified service specification model; in Chapter 8, we present the visual composition tool and conduct several case studies; finally, Chapter 9 contains the concluding remarks and future directions.
CHAPTER 2. DISTRIBUTED SYSTEMS AND SERVICE ORIENTED ARCHITECTURE

2.1. Introduction

Distributed computing is the process of aggregating the power of several computing entities to run a single computational task collaboratively in a transparent and coherent way, so that they appear as a single, centralized system. A distributed system can be described as an application consisting of components running concurrently on different computers, which operate separately and are capable of communicating with each other. Andrew S. Tanenbaum has given a loose characterization: "A distributed system is a collection of independent computers that appears to its users as a single coherent system.” [Andrew S. Tanenbaum, 2001]. From the perspective of hardware, the computers constituting a distributed system are autonomous; from the perspective of software, the computers are treated by users as a single system. The World Wide Web is probably the largest distributed system at present. When you read a Web page, the Web browser running on your personal computer renders the content of the web page to you. Behind the scenes, the web browser communicates with various Web servers that provide Web pages. Sometimes, the browser accesses Web content through proxy servers, which cache the Web content of corresponding Web servers for performance and security purposes. During the process, domain name servers are used to locate a particular Web server.

Distributed system design and implementation is a complex task. Four goals must be achieved to make the effort worthwhile. (1) The primary goal is to enable users to access remote resources easily, and to share them with other users in a controlled way. (2) A distributed system should be
able to present itself to users and applications as a single computer system. In other words, a distributed system should be transparent – hide the fact that its constituent processes and resources are physically distributed across different computers. (3) The third goal of distributed system is called openness – the services of a distributed system should be offered according to standard rules that specify the syntax and semantics of those services. Services provided by a distributed system are generally specified though interfaces. (4) A distributed system should also be built to be scalable with respect to its size so that more users and resources can be added to the system; it should be geographically scalable when users and resources lie far distant from each other; it should be administratively scalable so that it is manageable while spanning multiple independent administrative organizations.

In order to build a large distributed system, we often decompose it into sub-systems that provide certain related set of services. The initial step of building a distributed system is called architectural design - identifying these sub-systems and establishing a framework for sub-system control and interaction. There are several architectural styles that have been used to build distributed systems, such as client-server, distributed object, peer-to-peer and three-tier model. Service Oriented Architecture (SOA) is a recently proposed architectural style that has promising applications to Internet based business applications. The basic model of Service Oriented Architecture is illustrated in Figure 1, where the two-headed arrows indicate the communication between the two parties at the ends is bi-directional.
SOA utilizes services as self-describing and platform-agnostic computational elements for developing applications/solutions. The functions provided by services can be anything from simple requests to complicated business processes. Services technologies allow organizations to implement and expose their core capabilities programmatically over the Internet using standard XML-based languages and protocols. Implemented services are deployed and published to public business registries, where a service can be discovered with a self-describing interface based on open standards, as well. The interface of the service specifies the syntax and semantics regarding how the service can be invoked. Furthermore, SOA also offers a scheme to support rapid, low-cost composition of distributed applications using existing services as fundamental building blocks [Papazoglou, 2003].

2.2. Web Services and Service Oriented Architecture

Even though SOA can be implemented using any service-based technology, currently most definitions of SOA identify the use of Web services (using SOAP and WSDL) as their
implementation. SOA often refers to a set of services residing on the Internet or an intranet using "Web services".

Web services provide a collection of protocols and standards for exchanging data between applications or systems. One advantage of Web services is that software applications written in different programming languages and running on different platforms can exchange data over the Internet in a manner similar to inter-process communication on a single operating system. The universal interoperability between applications (e.g., between Java and Perl, or Mac and Linux applications) is enabled by the open Web service standards. Two standardization committees, OASIS and the W3C are responsible for the architecture and standardization of Web services. Recently, the WS-I organization has been developing a series of profiles to advance Web service standards, so that the interoperability between various Web service implementations can be further improved.

The Web services protocol stack consists of a set of protocols based on XML (Extensible Markup Language). These protocols are used to implement SOA as illustrated in Figure 2.
In the above diagram, SOAP is short for “Simple Object Access Protocol”. SOAP specifies the runtime message that contains the service request and response. SOAP is a higher-level protocol than any particular transport protocol, thus it is can be implemented over HTTP or SMTP. WSDL is short for “Web Services Description Language”. WSDL describes a Web Service and the SOAP messages the service supports, which provides a programmatic interface specifying what the service does, paving the way for automation. UDDI is short for “Universal Discovery, Description and Integration”. UDDI is a cross industry standardization initiative for business service discovery, together with service registries, that facilitates standardized service publication and discovery process. We will discuss each standard in detail in the remainder of this section.

2.2.1. SOAP

SOAP [Martin Gudgin, 2003] is a standard for transmitting XML-based messages over computer networks, usually using HTTP. Currently, the SOAP specification is administered by the XML Protocol Working Group of the World Wide Web Consortium. As the foundation layer of the Web services protocol stack, SOAP provides a basic messaging framework that more advanced layers can build on.

Remote Procedure Call (RPC) [Andrew S. Tanenbaum, 2001] is the most common messaging pattern in SOAP. In this pattern a client node sends a request message to a server node, and the server sends a response message back to the client.

Because HTTP is the main transport protocol used in the Internet infrastructure, it has been chosen as the primary application layer protocol for SOAP. Because SOAP is not limited by firewalls, it provides a major advantage over other distributed protocols, such as Internet Inter-Orb
Protocol (IIOP) [OMG, 2004] and Distributed Component Object Model (DCOM) [Markus Horstmann, 1997], which are restricted by most firewalls.

Each SOAP message is structured with an SOAP *envelope*. Two additional sections, the header and the body of the message are contained in the envelope. The header carries the relevant information about the message, such as the date the message is sent and the authentication information. The header is not always required, but it must be included at the top of the envelope when it is present.

2.2.2. **WSDL**

WSDL is an XML format for describing Web services. The World Wide Web Consortium (W3C) has just released a draft for version 2.0 on May 11th, 2005, which will be a recommendation for an official standard to be endorsed by the W3C.

WSDL specifies the public interface to the Web Service, which provides an XML-based service description on how to communicate with the Web service. The service description includes the protocol bindings and message formats required to interact with the Web Services. The operations and messages supported by the service are described abstractly, which can be bound to a concrete network protocol and message format.

WSDL is used along with SOAP and XML Schema to provide Web services over the Internet. A client of the Web service reads the WSDL document to determine what operations are available on the Web site. Any particular data types involved in the WSDL document are constructed with XML Schema. The client can actually invoke one of the operations specified in the WSDL using the SOAP protocol.
2.2.3. **UDDI**

UDDI is a platform-independent, XML-based registry for businesses worldwide to list their services over the Internet. As an open industry initiative sponsored by OASIS, UDDI enables businesses to discover each other and provide description about the functionalities they provide over the Internet. A UDDI business registration consists of the following three components:

- **White Pages** - address, contact, and known identifiers;
- **Yellow Pages** - industrial categorizations based on standard taxonomies; and
- **Green Pages** - technical information about services exposed by the business.

UDDI has been designed to support interrogation via SOAP messages and to specify the URLs to WSDL documents.

2.3. **Literature Review**

Research in the area of Web services composition includes (i) composition specification languages, (ii) automatic composition techniques and tools, and (iii) composition analysis and verification techniques. BPEL4WS is one of the most visible efforts in the area of composition specification languages, which appears to be gaining the wide support from both industry and research communities. Service compositions defined in BPEL4WS can be deployed with the help of execution engines such as BPWS4J [Francisco Curbera 2004], Oracle BPEL Process Manager [Oracle 2005]. Other specification languages include WSCI [Assaf Arkin 2002], and BPEL4WS predecessors, WSFL [Leymann 2001] and XLANG [Thatte 2001]. Some progress on automatic composition of Web services (in the context of the Semantic Web) can be found in [K. Sycara 2003].

10
Verification of the Web services composition and its performance evaluation have been reported in [H. Foster 2003] and [S. Narayanan 2002]. [Xiang Fu, 2004] applies SPIN model checking to the verification of service compositions and conversations. One of the problems is the “realizability” of service composition: given conversation rules of a number of partner services, is it possible to create a composition using the partners. [Ferrara, 2004] discusses how BPEL4WS can be mapped to process algebra such as LOTOS. However, software tools have not yet been developed to support the translation between BPEL4WS and LOTOS, as well as the interpretation of the verification results. [H. Foster, 2003] uses Labeled Transition System Analyzer (LTSA) to verify service compositions, the properties to be verified includes service traceability and compatibility. In [Alin Deutsch, 2004], interactive Web applications powered by databases are considered as Web services, which provide access to information as well as transactions. The properties that can be verified about this type of Web services involve the sequences of inputs, actions, and states that may result from interactions with a user. Variants of Computation Tree Logic (CTL) [Michael Huth, 2004] has been used to capture these properties formally. The decidability issue of the formal verification has also been discussed.

In this paper, we focus on stateful services and related service description and composition problems. Services are stateful when they require both the consumer and the provider to share the same consumer-specific context. Being stateful reduces the overall scalability of the service provider due to the need to remember the shared context for each consumer. Being stateful also increases the degree of coupling between a service provider and a consumer, and makes switching service providers more difficult. However, stateful services are still needed in many practical scenarios, such as provision of customized services. Another scenario is to establish a session
between a consumer and a provider for efficiency reasons. For example, it is much quicker to share a session token rather than to send a security certificate along with every request [He 2003].

A stateful service usually provides many operations, and invocation of these operations must be choreographed. The choreography rules can be captured and specified by a conversation protocol. The conversation protocol can be thought of as the service behavioral interface, which complements the WSDL interface.

Compositions often involve incorporation of partner services exhibiting complex protocols, which poses a number of challenges: How to extend the current service description so that the service conversation protocol can be clearly specified? How can the incorporation be performed efficiently without violating the protocols?

A newly-constructed composite service can again be a stateful service. Unfortunately, its conversation protocol is often not easily known before the service is composed, due to the dependency on partner behaviors. However, the conversation protocol is necessary for the new service to be published and invoked appropriately. What is the conversation protocol of the newly-constructed composite service? We have devised an algorithm to derive the conversation protocol from the corresponding composition.

Our primary contribution is the creation of a design and analysis framework for service oriented distributed systems. The framework is based on Colored Petri Nets (CPNets) [Jensen 1996], and it consists of the following modules:

- A unified service specification model. The model is suitable for both compositions and conversation protocols. It eases the incorporation of partners with complex protocols into composition. It enables automated detection of errors hidden in compositions such as deadlocks, non-progressing loops, overflows, and violation of protocols. It also enables formal
verification of application specific service properties, such as the reachability of an expected state.

- An extended the current service description techniques to include the specification of service conversation protocols.
- A conversation protocol derivation algorithm and its implementation.
- Translation between BPEL4WS and the service specification model.
- A modeling and analysis tool supporting the unified service specification model (JCPNet tool). The tool implements various analysis algorithms related to our model.
- A visual composition tool supporting the framework. The tool puts various design and analysis functionalities together.

We have made a number of contributions to service composition, service description, and application-level protocol specification in the area of service oriented distributed systems and system modeling and analysis in the area of software engineering. Specifically, we have proposed a unified service specification model, which is more expressive for capturing the operation choreography rules of complex Web services than WSCL and possesses formal semantics that BPEL4WS lacks. We have identified a new research problem in the area of service composition: how to incorporate a partner service with complex operation choreography rules into a larger service composition efficiently and correctly. We have also developed our own supporting toolkit for CPNets and the unified service specification model. Compared to the well-known CPN tools, our toolkit has several advantages, such as its support for the use of the widely used Java programming language rather than Standard ML, and making the fundamental CPNets algorithms available in terms of Java APIs rather than as an external tool. The unified specification model and
the toolkit are especially suitable for specification of a service composition that incorporates complex partner services. Many design time composition errors, such as deadlocks and violations of partner choreography rules, can be automatically detected by the toolkit. Another important research problem, derivation of the conversation protocol from the service composition, has been identified and the derivation algorithm has been proposed and implemented. Finally, we have presented a comprehensive design and analysis framework featuring a graphical user interface for service oriented distributed systems. The framework consists of the modeling and analysis toolkit supporting the unified service specification model, a translator between the standard composition language and the model, a novel technique to derive the conversation protocol automatically from the composition, and a visual composition tool that integrates various design and analysis functionalities.
CHAPTER 3. MOTIVATING EXAMPLES

In this Chapter, we show the need to check the correctness of BPEL4WS compositions, and the importance of the conversation protocol of a service as its behavioral interface specification. We also introduce research issues regarding the incorporation of partner services exhibiting complex conversation protocols.

3.1. Faulty BPEL4WS Code: Insurance Claim Processing

The UML activity diagram shown in Figure 3 illustrates a BPEL4WS process handling insurance claims [W.M.P. van der Aalst 2002]. Two checks are carried out in parallel following the reception of a claim. Based on results of these two checks, either a rejection letter is sent (when at least one of checks fail) or a payment is made (when both checks succeed).

Figure 3. UML Activity Diagram of a faulty BPEL4WS process
If the results of both check_policy and check_claim are OK, then activity pay will be executed, and the process terminates successfully. This is the only scenario where the process is completed without any failure. If check_policy is not okay (NOK), and check_claim is NOK, then activity sendRejection will be executed twice. This is because the control flows will come to an or-join, which becomes active whenever one of its pre-condition is satisfied. If check_policy is NOK, and check_claim is OK, then join failure will occur at activity pay.

This example illustrates that errors may occur when we write BPEL4WS programs. What are the criteria of correctness? How to check the correctness of a BPEL4WS process? Some research work has been done in the area of workflow verification [Aalst 2000] [H.M.W. Verbeek 2001]. The challenge here is to represent the workflow of a BPEL4WS process with a suitable formal model.

3.2. Incorporating Complex Partners: Composing Travel Agent Service

![Diagram of Travel Agent Service](image)

Figure 4. The problem of composing a Travel Agent service.

Consider the problem of creating a new Travel Agent Web service through composition, utilizing an Airline partner service and a Credit Card partner service shown in Figure 4, where “Interact” is
an individual invocation and “Converse” involves the incorporation of a stateful service (a sequence of choreographed invocations). A similar problem is discussed in [Assaf Arkin 2002]. For simplicity, we will assume that the Credit Card partner service offers only a single operation, and therefore has a simple conversation protocol. However, the Airline partner service provides several related operations, which must be invoked according to a complex conversation protocol. Assume that the Airline service provides five different operations: checkSeatAvailability, reserveSeats, cancelReservation, bookSeats, and notifyExpiration, each of which performs a single airline travel related task. The operations must be invoked by a client according to the following synchronization rules:

- checkSeatAvailability must be the first operation to be invoked;
- reserveSeats may only be invoked if a client has already invoked checkSeatAvailability and that the requested seats are indeed available; the reservation is held only for a certain amount of time;
- bookSeats or cancelReservation may be invoked, but only if the seats have been reserved (by a successful invocation of reserveSeats) and the reservation has not yet expired;
- if neither bookSeats nor cancelReservation has been invoked by the client within a specified amount of time, the Airline service will itself invoke notifyExpiration to inform the client that the reservation has expired.

WSDL is used to publish a Web service in terms of its ports (addresses implementing this service), port types (the abstract definition of operations and exchanges of messages for each individual operation), and bindings (the concrete definition of which packaging and transportation protocols such as SOAP are used to inter-connect two conversing endpoints). Currently, WSDL
does not address synchronization rules regarding the operations (conversational behavior). However, a conversation protocol describing the conversational behavior of a stateful service is necessary for the service to be published and invoked correctly. The conversation protocol can be thought of as the behavioral interface of the service.

Now let us assume that we intend to create a Travel Agent Web service offering a value-added service to travelers by incorporating the Airline service and the Credit Card service, according to the following business process scenario.

A traveler planning on taking a trip submits a *TripOrder* to her travel agent, hoping to get an *Itinerary* proposal (*getItinerary*). The *TripOrder* contains information such as the departure and destination, departure date and time, and return date and time (for a round trip), the number of maximum connections, the number of travelers, and the other specification regarding the trip.

The Travel Agent finds the best itinerary to reach the destination, based on the traveler's criteria such as the cheapest fare, availability of flights, or frequent flyer miles accumulated by the traveler. Before the *Itinerary* can be proposed to the traveler, the Travel Agent invokes the Airline service to verify the availability of seats (*checkSeatsAvailability*). In case the seats are not available, the Travel Agent notifies the traveler and waits for the traveler to submit a modified *TripOrder*.

If seats are available, the proposed *Itinerary* is sent to the traveler for confirmation. She then decides to *reserve* the seats for the *Itinerary* and gives the Travel Agent her contact information so that the Airline service will be able to send her an e-Ticket.

Next, the Travel Agent interacts with the Airline service to electronically finalize the reservation (*reserveSeats*). Let us assume that the Airline holds such a reservation for one day (or \(D\) units of time), and that if a *BookRequest* is not received within one day, the seats are released
and the Travel Agent is notified. The Travel Agent sends a ReserveResult message to the traveler as an acknowledgement.

At this point, the traveler can either book or cancel the reservation. If she decides to book the trip, she sends a BookRequest to the Travel Agent containing her credit card information. The Travel Agent then invokes the Credit Card service to charge the traveler for the trip. If the Credit Card service approves the charge, the Travel Agent invokes operation bookSeats of the Airline service to finally book the seats. As a result, the Airline service books the seats for the proposed Itinerary, and issues an e-ticket to the traveler. The Airline service also sends a Statement together with a detailed description of the Itinerary to the traveler. On the other hand, if the credit card charge is rejected, the Travel Agent notifies the traveler and waits for alternate payment information.

The composition of the Travel Agent Web service described above requires incorporation of the Airline partner service which has a relatively complex conversation protocol. Creating a composition like this is an error-prone task, since the partner’s conversation protocol may be violated in the composition. If the composition and conversation protocols can be specified using a unified verifiable model, then we can formally analyze the composition.
CHAPTER 4. THE UNIFIED SERVICE SPECIFICATION MODEL

We have already seen the need for formal specification models for compositions and conversation protocols. Basically, a composition is a way to implement a service, and the conversation protocol is the behavioral interface of the service. Therefore, we present a unified specification model for compositions and for conversation protocols. The model enables efficient incorporation of partner services exhibiting complex conversation protocols, and formally verifies whether or not conversation protocols are violated. CPNets is a graphical formal model suitable for modeling concurrency, synchronization and communication, and it has been used with good success in specification of workflow systems [Aalst 1998] and communication protocols [J. Billington 1999] [Jensen 1996]. Therefore, our unified specification model (WSPNet) is based on CPNets. Some previous work has been done in [Krys J. Kochut 2004] and [X. Yi 2004].

4.1. Petri Nets and Their Variants

The concept of Petri nets was first proposed in Carl Adam Petri’s dissertation [Petri, 1966]. Since their introduction in the 1960s they have evolved impressively. An international conference on Petri nets theory and application, ICATPN, is held annually. At present several versions of this graphical modeling language exist, which have widespread application in specification, verification, and performance analysis of distributed parallel systems including communication networks. A review of the history of Petri nets and an extensive bibliography is presented in [Murata, 1989].

Essentially, Petri nets are a graphical and mathematical model – a type of directed bipartite graph extended with token firing rules. A Petri net is a collection of places (represented by ellipses
or circles), transitions (straight bars), edges (oriented arcs), and tokens, which are the marker of a place. Each edge connects a place to a transition or vice versa. Places can be considered as conditions on the system control states and transitions as actions. When one or more tokens fill all input places of a transition, the transition is enabled. An enabled transition can fire, which consumes tokens in the input places then deposit tokens into the output places. When more than one transitions are simultaneously enabled, which transition is fired first is non-deterministic. The initial marking of a Petri net determines the execution sequence: if for any initial assignment of tokens the behavior of the net does not stall, the net is called “live”. Otherwise, it is possible to find all the blocking situations, which are also called dead markings.

There exist several variants of the generic Petri nets introduced above, from the simplest Petri nets with boolean token to high-level Petri nets where tokens are structured values. For each class there is a set of ad hoc supporting software tools. The relation between high-level and generic Petri nets is analogous to the relation between high-level programming languages and assembler code: the expressive capabilities are be the same, but the superior abstraction of the high-level Petri nets produces simpler designs.

A high-level Petri net is a Petri net extended with ‘color’, ‘time’ and ‘hierarchy’. The first variant is called Colored Petri nets. Since tokens often represent objects (e.g., resources, goods and humans) in a distributed system, the attributes of these objects need to be captured. For example, if a token represents a truck, then we may need to represent the capacity, registration number and location of the truck. These attributes are not readily captured by tokens in a generic Petri net, and the generic Petri net model has been extended with colored or typed tokens. In a colored Petri net every token is similar to a variable in Java programming language: it has a value which is referred to as ‘color’. The set of all the possible values a token can assume is called a color set. In our
project we apply Colored Petri nets (CPNet) model to service oriented distributed systems. We have more detailed discussion of CPNet later in this chapter.

The second variant is called timed Petri nets. For many real systems it is important to specify the temporal behavior of the system such as durations and delays. There are many approaches to introduce time into the generic Petri net. One of them is to use a timing mechanism where time is associated with tokens and transitions determine delays.

Although timed colored Petri nets allow for a concise specification of many complex systems, precise specifications for many real distributed systems tend to be rather large and complex. This motivates the introduction of the third variant of Petri nets, hierarchical Petri nets, which provides a hierarchy construct called “system”. A system is an aggregate of places, transitions and possibly subsystems [Aalst, 1994].

Along with their supporting tools, Petri nets are a promising model for describing and studying information processing which are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. With its graphical nature, Petri nets can be used as a visual aid similar to flow charts and UML activity diagrams. In addition, Petri nets token firing rules can be used to model and analyze dynamic and concurrent system activities. As a mathematical model, it is possible to formulate state equations, algebraic equations, and other mathematical models governing the behavior of distributed systems.

Now, we give a more detailed presentation of CPNets. We start with an informal description of the structural and behavioral aspects of CPNet, followed by the formal definition of CPNet. CPNet is a directed bipartite graph, which has two different types of nodes (vertices): place and transition. An arc (edge) connects from a place to a transition; or from a transition to a place. Data manipulation in CPNets is supported by CPN-ML, which is an extension of Standard ML. A
CPNet has a declaration of data types (called color sets), token variables, constant values and functions in CPN-ML. A token variable can be bound to an element of the color set which the variable belongs to. A token is defined as an element of a particular color set. A token variable bound to a value is a token. Furthermore, the declared variables, constants and functions are used to construct CPN-ML expressions, which are used to inscribe places, transitions and arcs of the CPNet. Each place $p$ is mapped to a color set $C(p)$, and $p$ contains a collection of tokens which is a multi-set over $C(p)$. The state of $p$ containing tokens is called a marking of $p$. The distribution of various tokens all over the places in a CPNet is called a marking of the CPNet. An arc is inscribed with an expression, which evaluates to a multi-set over the color set of its input place when all the variables in the expression are bound to values. A transition has a boolean expression called the guard. The collection of variables which occur in the input arc expressions and the guard expression is called the variables of the transition. The pairing $(t, b)$ of a transition $t$ and a binding $b$ of the variables of $t$ is called a binding element. A binding element (or transition $t$) is enabled when each input place of $t$ contains enough tokens the input arc expression “asks for”. An enabled transition may fire. When the transition fires, firstly the tokens in each input place are consumed according to the input arc expression, secondly new tokens are added to each output place according to the output arc expression. The firing of a transition is deemed as an atomic event, there is no interim marking corresponding to the state when the input tokens have been consumed and output tokens have not yet added to output places.

Now we formally define non-hierarchical CPNets as many-tuples. Please note that the purpose of formal definition here is to give a mathematically sound and unambiguous definition of CPNet. We will always specify any particular net in terms of a CPNet diagram. Before we state the definition of CPNet, we introduce the following notations:
The set of all elements in T is denoted by the type name T itself.

The type of a variable v is denoted by Type (v).

The type of an expression expr is denoted by Type(expr).

The set of variables in an expression expr is denoted by Var(expr).

A binding of a set of variables V is denoted by associating with each variable v ∈ V an element b(v) ∈ Type(v).

The value obtained by evaluating an expression expr in a binding b is denoted by expr<b>. Var(expr) is required to be a subset of the variables of b, and the evaluating is performed by substituting for each variable v ∈ Var(expr) the value b(v) ∈ Type(v) determine by the binding.

With the above notations, we are ready to define CPNets. We use B to denote the boolean type {false, true}, and having standard operations from propositional logic. Assume Vars is a set of variables, we use Type(Vars) to denote the set of types {Type(v) | v ∈ Vars}. Some motivations and explanations of the individual elements of the definition immediately follow the definition.

Definition 1. (CPNet) A CPNet is a tuple CPN = (Σ, P, T, A, N, C, G, E, I) satisfying the requirements below:

(i) Σ is a finite set of non-empty types, called color sets.

(ii) P is a finite set of places.

(iii) T is a finite set of transitions.

(iv) A is a finite set of arcs such that: P ∩ T = P ∩ A = T ∩ A = Ø.

(v) N is a node function. It is defined from A into P × T ∪ T × P.

(vi) C is a color function. It is defined from P into Σ.

(vii) G is a guard function. It is defined from T into expressions such that:
\[ \forall t \in \mathbb{T}: [\text{Type}(G(t)) = B \land \text{Type}(\text{Var}(G(t))) \subseteq \Sigma]. \]

(viii) E is an arc expression function. It is defined from A into expressions such that:

\[ \forall a \in A: [\text{Type}(E(a)) = C(p)_{\text{MS}} \land \text{Type}(\text{Var}(E(a))) \subseteq \Sigma] \text{ where } p(a) \text{ is the place of } N(a). \]

(ix) I is an initialization function. It is defined from P into closed expressions (without variables) such that:

\[ \forall p \in P: [\text{Type}(I(p)) = C(p)_{\text{MS}}]. \]

(i) The set of color sets determines the types, operations and functions that can be used in the net expressions (i.e., arc expressions, guards, initialization expressions, color sets, etc.). If desired, the color sets (and the corresponding operations and functions) can be defined by means of a many-sorted sigma algebra (as in the theory of abstract data types). We assume that each type has at least one element.

(ii) + (iii) + (iv) The places, transitions and arcs are described by three sets P, T and A which are required to be finite and pairwise disjoint. In contrast to classical Petri nets, we allow the net structure to be empty (i.e., \( P \cup T \cup A = \emptyset \)). The reason is pragmatic. It allows the user to define and syntax-check a set of color sets without having to invent a dummy net structure. We require the sets of places, transitions and arcs to be finite, so that a number of technical problems such as the possibility of having an infinite number of arcs between two nodes.

(v) The node function maps each arc into a pair where the first element is the source node and the second the destination node. The two nodes have to be of different kind (i.e., one must be a place while the other is a transition). We allow a CP-net to have several arcs between the same ordered pair of nodes (and thus we define A as a separate set and not as a subset of \( P \times T \cup T \times P \)).
(vi) The **color** function $C$ maps each place, $p$, to a type $C(p)$. Intuitively, this means that each token on $p$ must have a data value that belongs to $C(p)$.

(vii) The **guard** function $G$ maps each transition, $t$, into a boolean expression where all variables have types that belong to $\Sigma$. We shall allow a guard expression to be missing, and consider this to be a shorthand for the closed expression true.

(viii) The **arc expression** function $E$ maps each arc, $a$, into an expression of type $C(p(a))_{\text{MS}}$. This means that each evaluation of the arc expression must yield a multi-set over the color set that is attached to the corresponding place. We shall also allow a CPN diagram to have an arc expression $\text{expr}$ of type $C(p(a))$, and consider this to be a shorthand for $1'(\text{expr})$. Intuitively, this means that the arc expression evaluates to a color in $C(p(a))$ which we then consider to be a multi-set with one element. Finally, we shall allow an arc expression to be missing as a shorthand for empty.

(ix) The **initialization** function $I$ maps each place, $p$, into a closed expression which must be of type $C(p)_{\text{MS}}$, i.e. a multi-set over $C(p)$. Analogously to (viii), we shall also allow an initial expression to be missing as a shorthand for empty [Jensen, 1996].

4.2. **Verification and Occurrence Graph**

In the context of hardware and software systems, the term “formal verification” refers to the act of proving or disproving the correctness of a system with respect to a certain formal specified property, using formal methods.

Systems are modeled by mathematical models for formal verification. Mathematical system models include Finite State Machines (FSM), Labeled Transition Systems (LTS) and their compositions, Petri nets, timed automata and hybrid automata, cryptographic protocols,
combinatorial circuits, digital circuits with internal memory, and abstractions of general software components. The properties to be verified are often described in temporal logics, such as Linear Temporal Logic (LTL) or Computational Tree Logic (CTL).

Usually, formal verification is carried out algorithmically. The main approaches to implementing formal verification include state space enumeration, symbolic state space enumeration, abstract interpretation, abstraction refinement, process-algebraic methods, and reasoning with the aid of automatic theorem provers such as HOL or Isabelle.

In this dissertation, we analyze service oriented distributed systems using a state-based enumeration method based on CPNet. Most state space methods investigate the reachable states by constructing an occurrence graph (also known as reachability graph, O-graph for short). Basically, an O-graph is a directed graph consisting of all the states the system can reach from its given initial state. Each node of the graph represents a state of the system, and each directed arc is labeled with the action that leads to the next state. An O-graph provides an interleaving semantics of a system: two or more actions occurring simultaneously are treated as an un-ordered sequence. We state the formal definition of the O-graph as follows. Readers who are interested in the complete coverage of CPNet O-graph are referred to [Jensen, 1996].

Definition 2. The occurrence graph of a CP-net, also called the O-graph, is the directed graph OG = (V, A, N) where:

(i) V = [M0>, V is the set of all markings that can be reached from Marking M0.

(ii) A = {(M1, b, M2) ∈ V × BE × V | M1 [b> M2}

(iii) ∀ a = (M1, b, M2) ∈ A: N(a) = (M1, b, M2)
The O-graph has a node for each reachable marking and an arc for each step that occurs – with a single binding element. The source node of the arc is the start marking of the step, while the destination node is the end marking.

When we have a CPNet where all variables (in the arc expressions and guards) have finite color sets, it is straightforward to prove that the O-graph is finite iff all place instances are bounded.

4.3. WSPNet

Definition 3. WSPNet (Web Service Process Net) is an extension of CPNet used to specify service compositions and conversation protocols. WSPNet is defined as a tuple \((N, \Omega, S)\) where:

- \(N\) is the underlying CPNet.
- \(\Omega\) is a set of WSDLOperation’s. Each \(WSDLOperation\) is defined as a tuple \((name, type, inputPlace, outputPlace)\), where \(name\) is a string that identifies the operation; \(type\) indicates the type of operation, which evaluates to an element in the enumerated set of \\\n\{\text{ONE\_WAY, REQUEST\_RESPONSE, SOLICIT\_RESPONSE, NOTIFICATION}\}; \(inputPlace\) refers to a place in the CPN which represents to the input of the corresponding operation, similarly \(outputPlace\) refers to a place in the CPN which represents the output of the corresponding operation.
- \(S\) refers to a special place in the CPN, such that a token in the place indicates the corresponding service is ready to start a new instance, and a token put back into the place indicates the instance is terminated.
The workflow of the composition and the choreography of operations can be expressed with the underlying CPNet $N \Omega$ corresponds to the collection of operations defined in WSDL. Place $S$ simply provides a way to mark the beginning and end of the service.

We have developed a modeling and analysis package, JCPNet tool [Xiaochuan Yi 2005], to support verification of WSPNets as well as CPNets. More detailed description of the development of JCPNet tool is given in Chapter 5.

4.4. Generating BPEL4WS Code

Since BPEL4WS is the de-facto standard of Web service composition language, the final implementation of new compositions developed using our design and analysis framework specified in BPEL4WS. However, the initial design of BPEL4WS processes are created using WSPNet model. We have developed a designer to support visual design of initial BPEL4WS processes. The designer allows a user to verify the initial design using the O-graph analysis. Once the initial design passes the verification, the designer further allows the user to generate a skeleton BPEL4WS code from the WSPNet model automatically. Finally, the skeleton BPEL4WS file can be loaded into the more implementation oriented tools such as the Oracle BPEL editor, where complete execution details are added to the skeleton.

4.5. Specification of Service Conversation Protocol

A service conversation protocol specifies how the operations provided by the service should be choreographed to achieve a desired result. It is important for the service to be properly utilized. It can be thought of as the behavioral interface of the service, which complements the abstract interface defined in WSDL.
WSCL [Arindam Banerji 2002] allows the specification of conversation protocols based on the Finite State Machine (FSM) model. However, concurrency and communication can not be modeled by automata without proper extension. For example, the WSPNet model shown in Figure 5 specifies a service which offers two request-response operations, and operation 2 must be invoked the same number of times as operation 1 has been invoked. From the perspective of computation models, the service accepts the non-regular language (context free) \( \{a^n b^n \mid n \geq 1\} \). A conversation protocol like this can not be expressed by WSCL, which uses automata to specify conversation protocols.

Figure 5. A service whose conversation protocol can not be expressed with automata
With the emergence of BPEL4WS, conversation specification languages such as WSCL should be subsumed into the overall process specification [S. Chatterjee 2004]. An abstract BPEL4WS process can be used to define conversation protocols. However, the formal semantics of BPEL4WS is not yet well defined. We contend that the specification of conversation protocol should take composition into consideration, thus partners exhibiting complex protocols can be incorporated efficiently. Furthermore, protocol violation should be able to be automatically verified.

Our solution is representing the conversation protocol as a WSPNet, the general guidelines are as follows:

(1) Each operation is represented by a transition. An inputPlace (if exists) connects to the transition, and represents the reception and buffering of inbound messages for the operation. The transition also connects to an outputPlace (if exists), which represents buffering and transmission of outbound messages for the operation. The transition also has one input place which stands for the pre-condition of the operation, and one output place which stands for the post-condition of the operation.

(2) Messages exchanged by the service and its clients are modeled by tokens. Small-sized color sets are used to capture the protocol-relevant feature of a message.

(3) The synchronization rules of the conversation protocol are captured by connecting the transitions (each of them representing an operation) with places, arcs, and dummy transitions used only for control flow purposes.

For instance, the conversation protocol of the Airline service in the Travel Agent example is represented by a WSPNet shown in Figure 6.
Declarations:

color Msg = with TripOrder | ReserveReq | ReserveRes | BookReq | BookRes | CancelReq | CancelRes | Timeout | CheckReq | Itinerary;
color BOOL = bool with (no, yes);
color Start = unit with go timed;

var q: Msg; var st: Start; var a: BOOL; val T = 99999;

Figure 6. WSPNet for the conversation protocol of Airline service

The conversation protocol in XML format is listed in Figure 7.
Our approach to extend the current WSDL only service description with conversation protocol is similar to what has been done in [Dorothea Beringer 2001], namely include the conversation protocol as an additional overview document in the business entity in the UDDI registry. Both the URL to WSDL document and the URL to the conversation protocol in XML format will be specified in the overview documents URL of the corresponding business entity:

Figure 7. Conversation protocol in XML format: airlineProtocol.xml.

```xml
<convProtocol modelType = "WSPNet">
  <operations>
    <operation transition="check" inputPlace="p1a" outputPlace="pola"/>
    <operation transition="reserve" inputPlace="p12a" outputPlace="p2a"/>
    <operation transition="book" inputPlace="p13a" outputPlace="p3a"/>
    <operation transition="cancel" inputPlace="p14a" outputPlace="p4a"/>
    <operation transition="timeout" inputPlace="" outputPlace="p5a"/>
  </operations>
  <start place="begin"/>
  <transitions>
    <transition name="check"/>   <transition name="reserve"/>
    <transition name="book"/>   <transition name="cancel"/>
    <transition name="timeout"/>
  </transitions>
  <places>
    <place name="begin"/>   <place name="p1a"/>   <place name="p12a"/>
    <place name="p13a"/>   <place name="p14a"/>
    <place name="p15a"/>   <place name="p2a"/>   <place name="p3a"/>
    <place name="p4a"/>   <place name="p5a"/>
    <place name="p1a"/>   <place name="p2a"/>   <place name="p3a"/>
  </places>
  <arcs>
    <arc source="begin" target="check"/>   <arc source="check" target="p1a"/>
    <arc source="p1a" target="reserve"/>   <arc source="reserve" target="p2a"/>
    <arc source="p2a" target="book"/>   <arc source="book" target="cancel"/>
    <arc source="p2a" target="timeout"/>   <arc source="timeout" target="begin"/>
    <arc source="cancel" target="begin"/>   <arc source="timeout" target="begin"/>
  </arcs>
</convProtocol>
```
Listing 1. Include conversation protocol and WSDL in UDDI registry

An alternative approach to just extend the WSDL only service description is to propose an additional tag `<convProtocol/>` to W3C, for example:

```xml
<convProtocol>
</convProtocol>
```

Listing 2. Proposal of a new tag in WSDL for conversation protocol

4.6. Specification of Compositions

A composition can be thought as an automated workflow supported by Web services. Workflow has been modeled using standard Petri Nets [Aalst 1998]. A thorough analysis of the workflow patterns occurring in BPEL4WS is conducted in [P. Wohed 2002]. We will discuss how to represent each workflow pattern in BPEL4WS using CPNets constructs in Chapter 6. For example, the Travel Agent service composition given in the motivating examples can be specified by a WSPNet shown in Figure 8, where Airline partner service (protocol shown in Figure 6) has been incorporated in the composition.
Once a composition is specified as a WSPNet, the correctness of the composition can be formally verified. For the WSPNet model of a correct composition, there should not be any extra dead markings (other than the expected termination), or any non-progressing cycles; all places should be bounded. The violation of the conversation protocol during incorporation of a partner leads to extra dead markings.

We have developed a visual composition tool to support design of composite services involving complex partners. The visual design can be stored into BPEL4WS as well as WSPNet. We have also developed an open source software tool JCPNet [X. Yi 2005]. JCPNet supports formal analysis of CPNets and WSPNets including state space analysis. An advantage of JCPNet is that the analysis algorithms can be utilized flexibly in terms of Java APIs rather than solely relying on external tools such as CPN tools [CPN_Group 2005].
Declarations of taComposition.cpn:
color Msg = with TripOrder | ReserveReq | ReserveRes | BookReq | BookRes | CancelReq | CancelRes | Timeout | CheckReq | It timed;
color Start = with go timed;
var q:Msg; var st: Start;
val T = 1024;

Figure 8. WSPNet for Travel Agent Service composition
CHAPTER 5. CONVERSATION PROTOCOL DERIVATION

As we have mentioned in the introduction, a composite service usually offers multiple operations, which must be choreographed according to its conversation protocol. The conversation protocol is needed for service publication, and it must be consistent with the composition. In this Chapter, we present an algorithm that derives a conversation protocol from the composition model automatically. Along with the WSDL interface, the derived conversation protocol is published to a UDDI registry where the URL of the conversation protocol and the URL of WSDL interface are referred by UDDI tModel.

The basic idea of the derivation algorithm is simplifying the WSPNet service composition model, while preserving its externally observable interaction behavior. Now we discuss the derivation algorithm formally. Suppose a service $S$ offers a finite set of operations, denoted by $\Omega(S)$, that can be invoked. Assume service $S$ is realized by a composition, which is represented by WSPNet $N$, and the initial marking of $N$ is denoted as $M_0$. Note $M_0$ corresponds to the state when the composite service and all of its partners are all ready to start. If placing a token in the inputPlace of an operation $\alpha \in \Omega(S)$ at a certain marking $M$ leads to a live marking, then $\alpha$ is called invokable at marking $M$. A terminal node of the occurrence graph of $(N,M)$ (no outgoing arcs) represents a dead marking of $N$.

We introduce the definitions of two properties of reactive services:

Definition 4. Perpetual Service

A service is called perpetual if there exists a certain amount of time $T$, such that observing the service for duration of $T$, we can always find at least one operation that can be invoked.
Definition 5. Restorable Service

A service is called *restorable* if there exists a natural number \( L \), such that the service can eventually return to its initial state (marking) after less than \( L \) invocations.

The above two properties are reasonable requirements for most real world online services: a service is perpetual means that it never goes out of service permanently; a service is restorable implies that each instance of the service process can be terminated after a reasonable amount of time.

Since the O-graph of an unbounded CPNet or a CPNet using infinite color sets is infinite, we only consider a composition (denoted by \( N \)) satisfying the following conditions:

1. \( N \) is always bounded from any bounded initial marking \( M \).
2. All color sets used in \( N \) are finite.

Therefore, the occurrence graph of \((N, M)\) is finite.

Theorem 1. If service \( S \) is perpetual and restorable, its composition model \( N \) is bounded, all color sets used in \( N \) are finite, and the initial marking of \( N \) is \( M_0 \), then the conversation protocol of \( S \) can be represented by a finite automaton \( A = (Q, \Sigma, \delta, q_0, F) \) as follows:

\[
q_0 = M_0 \quad \text{and} \quad F = \{M_0\}.
\]

This indicates that the service is restorable.

\( \Sigma \) is a set of events, each of which is the invocation of one operation in \( \Omega(S) \). The event is initiated by clients of the service. For simplicity reasons, each event is denoted by the symbol of the corresponding operation.

Each state in \( Q \) corresponds to a dead marking of \( N \). There exist certain operations of \( \Omega(S) \) that can be invoked at this state.
For any $M \in Q$ and any operation $\alpha \in \Omega(S)$ that is invokable at $M$, if invocation of $\alpha$ leads to marking $M' \in Q$, then $\delta(M, \alpha) = M'$ is a transition function of $A$.

Proof of Theorem 1:

Remember the conversation protocol of a service is defined as choreography rules of its operations. Consider the composition model $N$ with initial marking $M_0$, based on CPNet O-graph construction algorithm given in Volume 2 of [Jensen 1996], we can explore all possible choreographies of its operations as follows:
Let $q_0 = M_0$, and $F = \{M_0\}$.
Let $\Sigma = \Omega(S)$.

Let the initial marking of $N$ be $M_0$.
Let $Waiting$ and $Trace$ be two sets which are initially empty.
Construct the occurrence graph $G$ for $(N,M_0)$.

For each terminal node of $G$ {
  Add the node to $Waiting$. Add the node to $Trace$.
}

While $Waiting$ is not empty {
  Remove a marking $M_1$ from $Waiting$.
  Find all the invokable operations of $N$ at $M_1$.
  For each invokable operation $\alpha$ at $M_1$ {
    Add a token into $inputPlace(\alpha)$, get a new marking $M'_1$.
    Construct the 0-graph $G'$ of $(N,M'_1)$.
    For each terminal node $M_2$ of $G'$ {
      Set current marking of $N$ to $M_2$.
      Empty $outputPlace(\alpha)$, the marking of $N$ changes to $M'_2$.
      If $M'_2$ has not yet occurred in $Trace$ {
        Add $M'_2$ to $Trace$.
        Add $M'_2$ to $Waiting$.
        Found a service transition function $\delta(M_1,\alpha) = M'_2$.
      }
      Else {
        Session Over.
        Found a service transition function $\delta(M_1,\alpha) = M_0$.
      }
    }
  }
  Set the current marking of $N$ to $M_1$.
}// End For each invokable operation
Q = Trace

Figure 9. Service Conversation Protocol Derivation Algorithm.

Because each occurrence graph of $(N,M')$ in the above algorithm is finite ($N$ is bounded and uses finite color sets) and $Q = Trace$ is finite (the number of states of the service is finite), the above algorithm terminates within finite number of steps and generates an automaton $A$.

The derivation algorithm has been implemented as a method getSTAutomaton of class WSPNet in package JCPNet [Xiaochuan Yi September 2004]. Furthermore, the service automaton
can be transformed into a WSPNet, which is often more intuitive and simpler than the automaton representation. Essentially, the automaton is a collection of transition functions. Considering the occurrence of operation \( \text{op} \) that leads the service to make a transition from \( M_0 \) to \( M_1 \) as follows:

![Figure 10. Service Transition Automaton](image1)

Depending on what type operation \( \text{op} \) is, the above automata construct can be transformed into the WSPNet construct as follows, where \( M_0 \) corresponds to place \( \text{pre} \) containing a token and place \( \text{post} \) is empty, while \( M_1 \) corresponds to the token in place \( \text{pre} \) has been consumed and place \( \text{post} \) containing a token.

![Figure 11. When \( \text{op} \) is of request-response type](image2)
When the WSPNet representing a service composition uses infinite color sets, the conversation protocol of the service can not be represented by an automaton. Therefore, the O-Graph-based derivation algorithm is no longer applicable. Currently, we are investigating a
mathematical reduction of WSPNet, with the goal to simplify a WSPNet without changing its externally observable behaviors.
CHAPTER 6. CPNETS REPRESENTATIONS OF BPEL4WS COMPOSITIONS

In order to capture all of the essential elements of Web services composition, we evaluate the composition constructs of BPEL4WS. In BPEL4WS, both the Web service composition and the conversation protocol are specified as workflows consisting of activities. There are two types of activities: basic activity and structured activity.

Basic activities include `<invoke>`, `<receive>`, `<reply>`, `<assign>`, `<wait>`, `<terminate>`, `<empty>` and `<throw>`. The first three types of basic activities are message exchange activities, which correspond to the operations defined in the WSDL port type. Primary attributes of these activities include partner, port type and operation involved in the message exchange. The activity `<invoke>` is used to invoke one of the operations offered by a partner service; activity `<receive>` is used to receive a message from a partner; and the activity `<reply>` is correlated to a previous `<receive>`, and is used to send a response back to the partner where message is received from. The fourth type of basic activity, `<assign>`, is used for data manipulations. The fifth type of basic activity, `<wait>`, is used to set a delay in process. The sixth type, `<terminate>`, is used to immediately terminates a process instance. Analogous to the empty statement of the C programming language, the seventh type, `<empty>`, does nothing in a process instance. The eighth type, `<throw>`, is used to signal an internal fault/exception.

Basic activities may be combined into larger structured activities using the following constructs:

- `<sequence>`, for sequential execution,
• `<flow>`, for parallel execution,

• `<switch>`, for conditional choice,

• `<pick>`, to respond to an external event,

• `<while>`, for iteration.

In addition, the activities within a `<flow>` construct may be connected by `<link>`’s to specify an almost arbitrary process control flow. The `<scope>` construct provides a context for a collection of activities with the associated fault handlers. The overall composition is enclosed within the `<process>` construct.

In order to determine if an activity is ready to start executing, we need to consider the following process-internal conditions:

• The semantics of the enclosing construct, such as process, scope, or any of the structured activities.

• If the activity is the target in a link construct, the joinCondition attribute of the activity, and the transitionCondition attribute associated with the link.

• The completion of a `<receive>` or a synchronous `<invoke>` activity depends on whether the expected message has been received. The reception of a message is considered an external condition to the process.

A Web service composition defined in BPEL4WS can be deployed to a BPEL4WS execution engine, which creates process instances that have access to a collection of process variables, partners (bound to actual component services at execution time), and message properties (similar to constants in a programming language, used for message correlation).
[P. Wohed 2002] has evaluated the expressiveness of BPEL4WS against twenty workflow patterns [W. M. P. Van Der Aalst 2003]. He presented a thorough study of how BPEL4WS is used to express various types of control flow. In the remainder of this Chapter, we will look at various workflow patterns in BPEL4WS and discuss how to represent them using CPNets model.

To represent BPEL4WS using CPNets, basically we represent activities with CPNets transitions. The control flow relations between activities specified in BPEL4WS are captured by CPNets token firing rules. As we shall see the following discussion, BPEL4WS is an expressive language and its process constructs can be used quite flexibly. We consider the following most used workflow patterns in BPEL4WS [P. Wohed 2002] and discuss how to represent each of them using CPNets.

The <sequence> tag, or <flow> tag with linear <link>’s, can be used to express Sequence workflow pattern, WP1 in [P. Wohed 2002].

```
1 <sequence>
2    activityA
3    activityB
4 </sequence>
```

```
1 <flow>
2   <links>
3     <link name="L"/>
4   </links>
5    activityA
6     <source linkName="L"/>
7     ...
8    activityB
9     <target linkName="L"/>
10  </flow>
```

Listing 3. BPEL4WS <sequence>

Listing 4. BPEL4WS sequential <flow>

The <sequence> or sequential <flow> construct can be represented as follows:
6.1. **Parallel Split and Synchronization**

The `<flow>` tag is primarily used to construct Parallel Split and Synchronization patterns, WP2 and WP3 in [P. Wohed 2002] respectively.

```xml
<sequence>
  <flow>
    activityA1
    activityA2
  </flow>
  activityB
</sequence>
```

Listing 5. Parallel Split and Synchronization using BPEL4WS `<flow>`, example 1.

```xml
<flow name="F">
  <links>
    <link name="L1"/>
    <link name="L2"/>
  </links>
  activityA1
  <source linkName="L1"/>
  activityA2
  <source linkName="L2"/>
  activityB
  joinCondition="L1 AND L2"
</flow>
```

Figure 15. CPNet model for Sequence construct in BPEL
Listing 6. Parallel Split and Synchronization using BPEL4WS <flow>, example 2

The parallel <flow> in BPEL4WS can be represented by CPNets model as follows:

![CPNets model diagram]

Figure 16. Representing parallel <flow> using CPNets

6.2. Exclusive Choice and Simple Merge

Exclusive Choice indicates a point in the process where, based on a decision or process control data, one of several branches is taken. Simple Merge indicates a point in the process where two or more alternative branches come together without synchronization, under the assumption that none of the alternative branches is ever executed in parallel. The BPEL4WS <switch> tag defines an ordered list of the <case> tag. A <case> corresponds to a possible activity which may be executed. The condition of a case is a Boolean expression on variables. As shown in Listing 7, <switch> tag can be used for Exclusive Choice (WP4) and Simple Merge patterns, WP4 and WP5 in [P. Wohed
An alternative BPEL4WS construct expressing Exclusive Choice and Simple Merge is `<flow>` with conditional `<link>`'s shown in Listing 8.

```
1 <switch>
2    <case condition="C1">
3       activityA1
4    </case>
5    <case condition="C2">
6       activityA2
7    </case>
8 </switch>
9 activityC

Listing 7. BPEL `<switch>` construct
```

```
1 <flow>
2   <links>
3     <link name="L1"/>
4     <link name="L2"/>
5     <link name="L1s"/>
6     <link name="L2s"/>
7   </links>
8   <empty>
9     <source linkName="L1" transitionCondition="C1"/>
10     <source linkName="L2" transitionCondition="C2"/>
11   </empty>
12 activityA1
13   <target linkName="L1">
14     <source linkName="L1s"/>
15     activityA2
16   </target linkName="L2">
17     <source linkName="L2s"/>
18     activityC
19     joinCondition="L1s OR L2s"
20     <target linkName="L1s">
21     <target linkName="L2s"/>
22 </flow>

Listing 8. BPEL `<flow>` construct with conditional `<link>`'s
```
The CPNet model for Listing 7 or Listing 8 is given as follows, where the branch conditions such as c1 and c2 are abstracted away when we do not have enough details about the condition expressions.

![Figure 17. CPNet model for Exclusive Choice and Simple Merge](image)

### 6.3. Iteration

BPEL `<while>` tag can be used to construct iterations, as shown in Listing 9.

```
1 <process>
2    <while cond="C1">
3       activityA
4    </while>
5 </process>
```

Listing 9. Iteration using BPEL4WS `<while>` construct

The `<while>` construct can be represented by CPNets as follows:
6.4. Deferred Choice

Deferred Choice, WP16 in [P. Wohed 2002], indicates a point in a process where one among several alternative branches is taken according to the information which is not necessarily available when this point is reached. This differs from the normal exclusive choice, in that the choice is not made immediately when the point is reached, but instead several alternatives are offered, and the choice between them is delayed until the occurrence of some event. This pattern is realized through BPEL4WS <pick> construct. The semantics of pick, i.e. awaiting the receipt of one of a number of messages and continuing the execution according to the received message, captures the key idea of this pattern, namely a choice is not made immediately when a certain point (i.e., the pick activity) is reached, but delayed until receipt of a message.

```
1 <pick>
2   <onMessage Msg1>
3       activityA1
4   </onMessage>
5   <onMessage Msg2>
6       activityA2
7   </onMessage>
8   <onMessage Msg3>
9       activityA3
```
Listing 10. Deferred Choice using <pick> construct

The <pick> construct can be represented by CPNets as follows:

Figure 19. CPNet model for <pick> construct

6.5. Interleaved Parallel Routing

Interleaved Parallel Routing, WP17 in [P. Wohed 2002], represents a workflow pattern where a set of activities is executed in an arbitrary order. Each activity in the set is executed exactly once. The order between the activities is decided at run-time: it is not until one activity is completed that the decision on what to do next is taken. In any case, no two activities in the set can be active at the same time. It is possible to capture this pattern in BPEL4WS using the concept of serializable scope (see Listing 11). A serializable scope is an activity of type scope whose
containerAccessSerializable attribute is set to “yes”, thereby guaranteeing concurrency control on shared variables.

```xml
<flow>
  <scope name=S1 containerAccessSerializable="yes">
    <sequence>
      write to variable C
      activityA1
      write to variable C
    </sequence>
  </scope>
  <scope name=S2 containerAccessSerializable="yes">
    <sequence>
      write to variable C
      activityA2
      write to variable C
    </sequence>
  </scope>
</flow>
```

Listing 11. Serializable scope in BPEL4WS

A CPNet model for serializable scope is shown as follows:
We have developed a program called BPEL2CPN to translate BPEL4WS code to WSPNet/CPNet. The translator is implemented by the dom.BPEL2CPN class in JCPNet package. The translator compiles a BPEL4WS file to an intermediate textual format, from where the corresponding WSPNet can be generated. Apache Xerces-J DOM parser [Apache 2005] is utilized. The translation scheme is based on recursive descent parsing [Alfred V. Aho 1986]. The translator eases the task of verifying existing BPEL4WS files.
CHAPTER 7. JCPNET: A VERIFICATION TOOLKIT

The model based approach is crucial to the analysis of system design. Colored Petri Nets (CPNet) have been used as software models with good success. They can be used to complement UML diagrams by providing formal modeling and analysis of dynamic behaviors of distributed systems. In order to enable CPNet model to be utilized more flexibly and widely, a new software tool JCPNet has been developed [Xiaochuan Yi September 2004]. This chapter discusses the background, motivation and development of JCPNet, and its applications to specification and analysis of distributed systems.

7.1. Introduction

System models provide abstractions of a physical system that allow engineers to reason about the system. All forms of engineering rely on models to understand complicated systems. The uses of models include predicting system qualities, reasoning about specific properties when aspects of the system are changed, and communicating key system characteristics among various stakeholders. System models can be developed as a precursor to implementation of the physical system, or they can be derived from an existing system as an aid to understanding its behavior [Brown 2005].

A major advantage of formal modeling is that it allows for formal reasoning. All state-based formal reasoning techniques involve examining every possible state of a system. Such techniques are automatic, thus they can be applied by less trained personnel, for analysis and error detection as well as verification. Therefore they are one of the most promising formal reasoning techniques [Glen Lewis 2001].
Colored Petri Nets model (CPNet) [Jensen 1996] [Lars M. Kristensen 1998] is a high-level Petri net extended with token color, time and hierarchy. It is a graphical formalism suitable for modeling systems where concurrency, synchronization, and communication are key characteristics. CPNets and Petri nets have already been applied successfully to communication protocols, data networks, embedded systems, business process/workflow modeling, and manufacturing systems. Moreover, the use of CPNets spans over many phases of system development ranging from requirements elicitation to design, validation, and implementation [Lars Michael Kristensen 2003]. The primary reason for the success of CPNets is because CPNet has a graphical representation, a well-defined semantics allowing formal analysis, and supporting software tools such as CPN Tools [CPN_Group 2005].

In our research project on Service-Oriented Distributed Systems, we focus on research issues regarding service composition involving stateful partner services, such as specification of behavioral interfaces (also called conversation protocols) and compositions. Our underlying specification model is based on CPNet. In certain situations, the conversation protocol of a composite service is not easily known before the composition is finished. Therefore we have devised an algorithm that derives the conversation protocol from the corresponding composition automatically. The derivation algorithm is based on a series of CPNet occurrence graph constructions. Unfortunately, the source code of CPNet occurrence graph construction algorithm has not been publicly available, which forced us to create our own data structures representing the CPNet model and implement the core CPNet algorithms.

In order to enable CPNet to be more widely and easily used, the programming language used for CPNet data structures and algorithms must be a widely used programming language. The programming language should also support convenient GUI development and XML processing, so
that CPNet models can be efficiently visualized and exchanged in XML documents. Initially C++ has been considered due to the execution speed of compiled C++ programs. Eventually, we turned to Java because class reflection mechanism is provided by Java and its data structures are more flexible. These features are keys to efficient representation of CPNet.

The CPNet model incorporates certain capabilities of programming language for richer data manipulation, which have enhanced the expressiveness of CPNet model and reduced the model size considerably compared to the standard Petri Net model [Murata 1989]. The programming language adopted in CPNet is CPN-ML, an extension of function programming language Standard ML [Robin Milner 1990]. CPNet elements such as color sets, token variables, arc expressions, transition guard expressions are specified in CPN-ML. Our contention is that the programming language incorporated in CPNet model specification should not be confined to one particular language such as SML. Java is one of the most popular programming languages. It is platform independent and provides an SDK for efficient GUI development and powerful XML processing. Last but not least, Java is expressive enough to specify CPNet inscriptions, as well.

Currently, the source code of CPNet data structure and algorithms is not freely available in a commonly used programming language such as Java or C++. By providing the CPNet data structure and algorithms in Java, we hope more researchers and programmers will get a chance to look at CPNet model and actually use it. Furthermore, basic CPNet data structures and algorithms can be integrated as APIs, which is more flexible than solely relying on external tools such as CPN tools.

Driven by the above motivations, we have developed a new tool supporting CPNet called JCPNet. JCPNet provides a general representation of CPNet model in Java classes and the implementation of a collection of common CPNet algorithms such as occurrence graph
construction algorithm, described in Volume 2 of [Jensen 1996]. An occurrence graph is also known as a reachability graph or state space. JCPNet also provides a number of example applications: arithmetic adder, dining philosophers, distributed database, resource allocation, and specification and analysis of service oriented distributed systems.

The chapter is organized as follows: in Section 2, we present concepts regarding CPNet and an overview of key data structures and algorithms in JCPNet. In Section 3, we discuss the importance of state space analysis and CPNet occurrence graph construction algorithm. In Section 4, we present several examples to show the application of JCPNet to distributed systems modeling and analysis. Finally, Section 5 contains concluding remarks and discussion of future direction.

7.2. **Key Data Structures for CPNets**

In this section, we give an informal description of concepts regarding CPNet structure and behavior. The description is helpful to understand the formal definition of CPNet and our implementation of JCPNet. Readers who are interested in the formal definition of CPNet structure and behavior should refer to Volume 1 of [Jensen 1996].

CPNet is a directed bipartite graph, which has two different types of nodes (vertices): place and transition. An arc (edge) connects from a place to a transition; or from a transition to a place. Data manipulation in CPNets is supported by CPN-ML, which is an extension of Standard ML. A CPNet has a declaration of data types (called color sets), token variables, constant values and functions in CPN-ML. A token variable can be bound to an element of the color set which the variable belongs to. A token is defined as an element of a particular color set. A token variable bound to a value is a token. Furthermore, the declared variables, constants and functions are used to construct CPN-ML expressions, which are used to inscribe places, transitions and arcs of the
CPNet. Each place p is mapped to a color set C(p), and p contains a collection of tokens which is a multi-set over C(p). The state of p containing tokens is called a marking of p. The distribution of various tokens all over the places in a CPNet is called a marking of the CPNet. An arc is inscribed with an expression, which evaluates to a multi-set over the color set of its input place when all the variables in the expression are bound to values. A transition has a boolean expression called the guard. The collection of variables which occur in the input arc expressions and the guard expression is called the variables of the transition. The pairing (t,b) of a transition t and a binding b of the variables of t is called a binding element. A binding element (or transition t) is enabled when each input place of t contains enough tokens the input arc expression “asks for”. An enabled transition may fire. When the transition fires, firstly the tokens in each input place are consumed according to the input arc expression, secondly new tokens are added to each output place according to the output arc expression. The firing of a transition is deemed as an atomic event, there is no interim marking corresponding to the state when the input tokens have been consumed and output tokens have not yet added to output places.

The UML Class Diagram of JCPNet is shown in Figure 21, which presents an overview of data structures related to CPNets. With the class diagram, now we describe each class in details. Please note the term “attribute” in UML has the same meaning as the term “class field” in Java, and the term “operation” has the same meaning as the term “class method” in Java.
Class Declaration

An object of class *Declaration* realizes the declaration of color sets, token variables, constant values, and expressions that will be used in the CPNet. *Declaration* is an abstract class, it must be extended for each specific CPNet.

Sometimes, token variables from different input arcs of a transition are not independent of each other. For example, in the CP-net of Distributed Database (see Fig. x), transition *receive* has two input token variables: *r* and *mes* (*mes* is pair of (*s*, *r*)), and any binding applied to transition *receive* should always keep *r* equal to the *r* of *mes*. The *match* method can be overridden to enforce such a specific constraint when necessary.

Class Node
CPNet is essentially a directed bipartite graph which contains two different types of nodes: places and transitions. However, a place and a transition still share some common attributes, which is captured by class Node to encourage reuse.

Class Place extends Node

The attribute currentMarking represents the current marking of the place, while attribute initialMarking represents the initial marking of the place. Operation Place() is the constructor of class Place.

Class Transition extends Node

The attribute vars represents the set of variables of the transition, including any variable which occurs in the guard expression and any input arc expressions of the transition. Even though the information attribute vars carries is implied by the guard expression and input arc expressions, we insist that the attribute vars should be specified explicitly for simplicity and efficiency purposes. Usually vars is a very small set.

Each token variable can be bound to any token in the corresponding input place. Operation getBindings() generate the all possible combinations of variable bindings. Operation isEnabled() indicates whether the transition is enabled with a binding of vars. If any input token is time stamped and the time stamp has not reached the current time, then even the transition is enabled, it is not yet ready to fire. Operation isReady() indicates the readiness of the transition. When a transition is enabled and ready, it can be fired. The firing of the transition will consume tokens from each input place according to the binding and input arc inscriptions, and generate tokens according to binding and the output arc inscriptions then add tokens into each output place.

Class Arc
An arc connects its source node to its target node. A node is either a place or a transition. The source node and target node of an Arc should never be of the same type. Otherwise an IllegalArcException will be thrown.

Class CPNet

The attribute places represents the finite set of places, while Attribute transitions represents the finite set of transitions. The operation getOGraph() constructs the occurrence graph of the CPNet with the current marking. The operation getPlaceByLabel() and getTransitionByLabel() are used to retrieve the reference of a place or transition by its label. The operation print() display the structure of the CPNet. The operation printMarking() is a static method used to display the content of a particular marking of the CPNet.

Class OGraphNode extends Node

Each node of an occurrence graph represents a reachable marking from the initial marking of the CPNet. Attribute id is used to refer to a node for convenience reasons. In depth-first search of graphs [Thomas H. Cormen 2001], nodes are colored to indicate their state during the search. Each node is initially white. It is grayed once upon it has been discovered in the search, then it is blackened when all its adjacent nodes have been examined (also called the node is finished). A depth-first search generates a depth-first forest. During the search, each node is also time stamped. There are two time stamps: attribute dtime records when the node is first discovered (and grayed), and attribute ftime records when the search finishes examining all output arcs of the node. Attribute parent refers to the parent of the node in the search tree.

Class OGraphArc
When an enabled binding element fires, the state of the CPNet transfers from one marking (denoted by attribute *source*) to another marking (denoted by attribute *target*). An object of class OGraphArc represents the transferring between two reachable markings.

Class OGraph

Attribute *nodes* represent the finite set of nodes of the occurrence graph. Attribute *net* refers to the CPNet the occurrence graph belongs to.

Operation *DFS_Visit*(root) performs a depth-first search visit starting from OGraphNode root. Operation *isReachable* (node1, node2) checks if the marking corresponding to node2 is reachable from the marking corresponding to node1. Operation *isHome*(home) examines if home represents a home marking.

7.3. **CPNet Occurrence Graph**

We have already indicated the importance of state-based formal reasoning, which is supported by state space methods. Most state space methods investigate the reachable states by constructing an occurrence graph (also known as reachability graph, O-graph for short). An O-graph is a directed graph consisting of all the states the system can reach from its given initial state. Each node of the graph represents a state of the system, and each directed arc is labeled with the action that leads to the next state. An O-graph provides an interleaving semantics of a system: two or more actions occurring simultaneously are treated as an un-ordered sequence. Readers who are interested in the formal definition of O-graph are referred to Volume 2 of [Jensen 1996].

Now we discuss O-graph construction algorithm, which is the key to the state space method using CPNet. When we have a CPNet where all variables (in the arc expressions and transition guards) have finite color sets, the O-graph is finite iff (if and only if) all places are bounded. Below
we recapture the abstract algorithm to construct the O-graph given in Volume 2 of [Jensen 1996]. The algorithm halts iff the O-graph is finite. Otherwise the algorithm continues for ever, producing a larger and larger subgraph of the O-graph.

```
Waiting := Ø
Node(M0)
repeat
    select a node M1 ∈ Waiting
    for all (b, M2) ∈ Next (M1) do begin
        Node(M2)
        Arc (M1, b, M2)
    end
    Waiting := Waiting – M1
until Waiting = Ø
```

Figure 22. An abstract O-graph construction algorithm

*Waiting* is a set of nodes. It contains those nodes for which we have not yet found the successor. *Node (M)* is a procedure that creates a new node M and adds M to *Waiting*. If M is already a node, the procedure has no effect. Analogously, *Arc (M1, b, M2)* is a procedure that creates a new arc with source *M1*, binding element *b*, and destination *M2*. If it is already in existence, the procedure has no effect. We use *Next (M1)* to denote the set of all possible “next moves”: binding element and the next reachable marking from marking *M1*.

Now, we present our implementation of the algorithm in Figure 23, Figure 24 and Figure 25, which show a number of additional key points which are not obvious in the abstract algorithm. For the complete source code refer to CPNet.java in JCPNet.
// method getOGraph of class CPNet
public OGraph getOGraph()
{
    M = getCurrentMarking();
    Vector waiting = new Vector();
    int id = 1; // integer id of an OGraphNode
    // create a empty OGraph og, og.net refer to the CPNet
    OGraph og = new OGraph(deco, this);
    // create node1. node1.marking = marking, node1.id=id
    OGraphNode node1 = new OGraphNode(marking, id++);
    og.nodes.addElement(node1);
    waiting.addElement(node1);
    OGraphNode node2 = null;

    while (!waiting.isEmpty()) {
        node1 = (OGraphNode) waiting.firstElement();
        M1 = node1.marking;
        setCurrentMarking(M1);
        for (each transition t) {
            bindings = t.getBindings();
            for (each binding b of t) {
                binding = (Vector) bindings.elementAt(j);
                if (t.isEnabled(binding)) {
                    t.fire(binding);
                    M2 = getCurrentMarking();
                    setCurrentMarking(M1);
                    if (M2 not yet occur in og) {
                        node2 = new OGraphNode(M2, id++);
                        og.nodes.addElement(node2);
                        waiting.addElement(node2);
                    }
                    if ((node1, (t, b), node2) not yet in og) {
                        Vector be = new Vector();
                        be.addElement(t.label);
                        be.addElement(binding);
                        new OGraphArc((node1, be, node2));
                    }
                } // end if t.isEnabled
            } // for each binding
        } // for each transition
        waiting.removeElement(node1);
    } // end while

    return og;
} // end getOGraph

Figure 23. O-Graph construction algorithm in Java-like pseudo code.
// method getBindings of class Transition
public Vector getBindings()
{
    Place place = null;
    Arc arc = null;
    Iterator it = null;
    Vector[] values = new Vector[vars.length];
    Object v = null;
    Vector bindings = null;
    Vector p = null;
    // for each variable of the transition
    for (int i = (vars.length - 1); i >= 0; i--) {
        values[i] = new Vector();
        // find the input arc with inscription vars[i] // vars[i] can be bound to any token in the input place
        for (int j = 0; j < inputArcs.size(); j++) {
            arc = (Arc)inputArcs.elementAt(j);
            if (arc.inscription.equals(vars[i])) {
                place = (Place)arc.source;
                it = place.currentMarking.iterator();
                while (it.hasNext()) {
                    v = it.next();
                    values[i].addElement(v);
                }
                break;
            }
        }
    }
    if (vars.length == 1) {
        bindings = new Vector();
        for (int j = 0; j < values[0].size(); j++) {
            Vector binding = new Vector();
            binding.addElement(values[0].elementAt(j));
            bindings.addElement(binding);
        }
    } else {
        if (i == (vars.length - 1)) p = values[i];
        else {
            // generate the combination of all possible variable binding
            bindings = Set.xproduct(values[i], p);
            p = bindings;
        }
    }
    return bindings;
} // end getBindings

Figure 24. Get all bindings of a transition
7.4. Examples of Applications

We present four example applications in this section. “Dining philosophers” and “Distributed database” have also been discussed and analyzed in Volume 1 and 2 of [Jensen 1996], here we use them to test the correctness of our implementation of CPNet O-graph construction algorithm. “Resource allocation” is a classical problem in operating system, which is used to illustrate how JCPNet can be used to detect deadlocks automatically. JCPNet has also been applied intensively in the specification, design and analysis of service-oriented distributed systems.

JCPNet has several advantages over CPN tools: (1) the content of each node in an O-graph can be found more conveniently, while in CPN tools you have to write a number of state space queries; (2) the O-graph can be generated automatically in the input format of GraphViz [AT&T 2005] for visualization; (3) JCPNet also enables CPNet modelers to utilize CPNet algorithms as Java APIs rather than solely relying on external tools.

Figure 25. Set.xproduct

```java
public class Set {
    public static Vector xproduct(Vector a, Vector b) {
        // a is Vector of Object
        // b is Vector of Object or Vector of Objects
        // return Vector of Vector
        Vector product = new Vector();
        for (int i = 0; i < a.size(); i++) {
            for (int j = 0; j < b.size(); j++) {
                Vector item = new Vector();
                item.addElement(a.elementAt(i));
                if (b.elementAt(j) instanceof Vector) {
                    Vector v = (Vector) b.elementAt(j);
                    for (int k = 0; k < v.size(); k++) {
                        item.addElement(v.elementAt(k));
                    }
                } else item.addElement(b.elementAt(j));
                product.addElement(item);
            }
        }
        return product;
    }
}
```
7.4.1. Dining Philosophers

Dining philosophers is a classical problem of concurrency and conflicts, which are captured in the CPNet model in Figure 3. The model is implemented by `DiningPhilosophers.java` in JCPNet.

![CPNet model of dining philosophers](image)

Declarations:
val n = 5;
color PH = index ph with 1..n;
color CS = index cs with 1..n;
var p: PH;
fun Chopsticks(ph(i)) = 1`cs(i) ++ 1`cs(if i=n then 1 else i+1);

Figure 26. CPNet model of dining philosophers

7.4.2. Distributed Database

Figure 27 shows the CPNet model of a distributed database where data replication is enabled and data consistency is enforced. The model is implemented by `DistributedDB.java` in JCPNet.
Declaration:
color DBM = index d with 1..n; color PR = product DBM * DBM;
color MES = subset PR by diff;
color E = with e;
fun diff(x,y) = (x<>y);
fun Mes(s) = PR.mult(1`s, DBM.all() – 1`s);
var s, r: DBM;

Figure 27. CPNet model of distributed database

7.4.3. **Resource Allocation**

The CPNet model show in Figure 28 shows the problem of allocating two resources r1 and r2 to two processes P1 and P2, only one resource is allowed to be allocated to a process at a time, and the two resources are allocated to each process in different order.
### The Ocurrence Graph of the CPNet Resource Allocation has 6 nodes
Node # 1 with marking:
Content of marking:
p11 : e
p21 : e
Resource1 : e
Resource2 : e
--------------------------------------
Input Arcs:
BE = [work1, [e]]
FROM Node # 4
BE = [work2, [e]]
FROM Node # 6
Output Arcs:
BE = [allocateR1ToP1, [e]]
TO Node # 2
BE = [allocateR2ToP2, [e]]
TO Node # 3
Node # 2 with marking:
Content of marking:
Input Arcs:
BE = [allocateR1ToP1, [e]]
FROM Node # 1
Output Arcs:
BE = [allocateR2ToP1, [e]]
TO Node # 4
BE = [allocateR2ToP2, [e]]
TO Node # 5
Node # 3 with marking:
Content of marking:
p11 : e
p22 : e
Resource1 : e
--------------------------------------

Input Arcs:
BE = [allocateR2ToP2, [e]]
FROM Node # 1
Output Arcs:
BE = [allocateR1ToP1, [e]]
TO Node # 5
BE = [allocateR1ToP2, [e]]
TO Node # 6
Node # 4 with marking:
Content of marking:
p13 : e
p21 : e
--------------------------------------

Input Arcs:
BE = [allocateR2ToP1, [e]]
FROM Node # 2
Output Arcs:
BE = [work1, [e]]
TO Node # 1
#ifndef5 is a TERMINAL node #>>>
Node # 5 with marking:
Content of marking:
p12 : e
p22 : e
--------------------------------------
Input Arcs:
BE = [allocateR2ToP2, [e]]
FROM Node # 2
BE = [allocateR1ToP1, [e]]
FROM Node # 3
Output Arcs:
Node # 6 with marking:
Content of marking:
p11 : e
The research allocation scheme leads to one deadlock, which has been detected automatically by the state space analysis reported in Figure 29. As we can see, node #5 indicates a deadlock which occurs when resource r2 has been allocated to process P1 and resource r1 has been allocated to process P2. The model is implemented by ResourceAllocation.java in JCPNet.

The deadlock can be avoided by (1) allocate all resources to a process at the same time (shown in Figure 30); or (2) allocate one resource to a process at a time, but allocate the resources to each process in the same order (shown in Figure 31).

Figure 30. Allocate all resources at the same time to a process. No deadlocks.
Figure 31. Allocate resources (one resource to one process at a time) in the same other to each process. No deadlocks.

7.4.4. Service Oriented Distributed System

As we have introduced in Chapter 3, a subclass of CPNet, WSPNet, has been created as the unified service specification model for service compositions and behavioral interface (conversation protocols). WSPNet supports incorporation of stateful partners, verification of service compositions, derivation of behavioral interfaces, and verification of application-level protocol
involving services. We will discuss the application of JCPNet tools in details in the following Chapters.
CHAPTER 8. WSPNET DESIGNER AND CASE STUDIES

We have developed a supporting tool for our design and analysis framework, WSPNet Designer, which integrates the following capabilities:

- Specification of a service compositions and conversation protocols visually using WSPNet model. These functions are provided in the “WSPNet” menu.
- Incorporation of a partner with complex conversation protocol into a larger composition. The function is provided in the popup menu inside an internal frame of the designer.
- Verification of the correctness of compositions. The function is provided in the “Analysis” menu.
- Automatic generation of the skeleton BPEL4WS code from corresponding WSPNet model. The function is provided in the “Tool” menu.
- Automatic derivation of service conversation protocol from the service composition model. The function is provided in the “Analysis” menu.
- Service discovery and publication involving complex conversation protocols. These functions are provided in the “Tool” menu.
- Loading an existing BPEL4WS files and automatically generate WSPNets for automatic verification and visualization. This function is provided in the “Tool” menu.

WSPNet designer is implemented in Java, using JGraph library, XML DOM parser, UDDI4J and JCPNet verification toolkit. A screen shot of the designer is shown Figure 32. We will illustrate how to use the designer in the following case studies.
8.1. Detect design errors in composition

In the first case study, we revisit the insurance claim processing problem we have introduced in Chapter 3. According to the description of the insurance claim processing workflow given in Chapter 3, a straightforward initial design of the BPEL4WS process can be specified with a WSPNet model as follows:
Figure 33. The initial design of insurance claim process

The above design has been created with WSPNet designer, which can generate a CPNet OGraph analysis report automatically as follows:
Figure 34. Generating CPNet OGraph analysis report

The OGraph analysis automatically generates a visualization of the OGraph (Figure 35) along with a detailed list of marking content which corresponds to each node of the OGraph (Listing 12).
Figure 35. Occurrence graph of claim processing

The Occurrence Graph CPNet claimProcessing.wsp has 26 nodes.
Node # 1 with marking:
Content of marking:
begin : e

Input Arcs:
Output Arcs:
BE = [receive, [e]]
TO Node # 2
Node 22 is a TERMINAL node.
Node # 22 with marking:
Content of marking:
end : e

c5 : e
Listing 12. The detailed CPNet OGraph Analysis report for the insurance claim process

Four dead markings have been detected by WSPNet designer. One of them corresponds to the successful termination of the insurance claim process. The other three dead markings correspond to the errors we have mentioned in Chapter 3. The strength of our approach is that the error detection is automated by WSPNet designer.

Now, we present a correct insurance claim process using block-structured design, which is shown as follows. The output place of transition “decide” starts a exclusive choice: if the results of both “checkPolicy” and “checkClaim” are OK, then transition “pay” will be fired, otherwise transition “sendLetter” will be fired.
In order to make sure the new process design is correct, we have conducted the following CPNet OGraph analysis, which shows that there exists only one dead marking – the successful termination of the Insurance Claim Processing.
Based on the verified WSPNet model, we can automatically generated the following BPEL4WS skeleton code using WSPNet designer.

```xml
<process>
  <sequence>
    <receive name="receive"/>
    <flow>
      <invoke name="checkPolicy"/>
      <invoke name="checkClaim"/>
    </flow>
    <assign name="decide"/>
    <switch>
      <case>
        <invoke name="pay"/>
      </case>
      <case>
        <reply name="sendLetter"/>
      </case>
    </switch>
  </sequence>
</process>
```

Listing 13. BPEL Skeleton Code for Claim Processing
Now we consider how to compose the Travel Agent Web service we have described in Chapter 3. We will go through the entire design and analysis cycle of composite web services, which includes

- specification of the conversation protocol of the Airline partner service using WSPNet model,
- incorporation of the Airline partner service into the larger Travel Agent service composition,
- verification of the newly composed Travel Agent service,
- automatic derivation of the conversation protocol from the Travel Agent service composition model,
- publishing the WSDL interface and conversation protocol of the Travel Agent service to a public UDDI business registry.

We have implemented a demo Airline partner service and published it to Microsoft UDDI business registry. Each of the URLs to the WSDL interface and conversation protocol can be found in the OverviewDoc of the corresponding tModel instance in the business entity. The conversation protocol of the Airline service can be specified using a WSPNet shown in Figure 38:
The WSPNet model of the Airline service conversation protocol is encoded and stored in an XML document, which is made available through the URL given in UDDI tModel instance – http://www.cs.uga.edu/~xyi/kyws/airlineProtocol.wsp.
The following three pictures illustrate how the Airline partner is incorporated into the larger Travel Agent service composition.

![Figure 39. Incorporating a partner – choosing a partner](image)

85
Figure 40. Incorporating a partner – partner protocol just incorporated
With the WSPNet composition model available, it is possible to verify if the Airline partner is incorporated correctly by detecting if there exist extra dead markings in the composition model, just as what we did for the Insurance Claim processing example. Furthermore, the conversation protocol of the composite service can be automatically derived from the WSPNet composition.
model. The derived conversation protocol is again represented with a WSPNet. The generateDot method of Class WSPNet in WSPNet designer package outputs a WSPNet into GraphViz dot file. GraphViz dot tool reads the dot file, carries out automatic layout for the WSPNet, and provides an automatic visualization of the conversation protocol as follows:

![WSPNet model of the derived conversation protocol](image)

Figure 42. WSPNet model of the derived conversation protocol
Finally, the URLs to the conversation protocol and WSDL interface of the Travel Agent service are specified in tModel instances. The complete service description is published to a public UDDI business registry using WSPNet designer. Now the Travel Agent service can be discovered and utilized widely.

8.3. Verification of Partner Incorporation

Now, we show two examples where the incorporation of a partner service into a larger composition goes wrong, and how these errors can be automatically detected by our toolkit.

In the first example, the travel agent composition tries to invoke operation “book” after “timeout” occurred in partner conversation protocol. Since the occurrence of operation “timeout” takes the token in place “p2a” away, a token will get stuck in place “pi3a”, as is shown in Figure 43.
Figure 43. The first example of incorporation error: partner session is over, composition process hangs.

This type of errors can be detected by running the protocol derivation on the service composition model. A series of CPNet OGraph analyses will be conducted as we do for conversation protocol derivation. This type of errors lead to dead markings where a token is stuck
in one of the input places of partner protocol. This example indicates that the incorporation must be carried out with care: once “timeout” occurs, any attempts to invoke “book” or “cancel” have to be reset. The incorporation error can be detected by the designer, as is shown in the following picture:

```
<<<<<<<<< receive3 fires >>>>>>>>
Content of marking:
p05ta : e
p31 : e

<<<<<<<<< invoke3 fires >>>>>>>>
Content of marking:
p05ta : e
p13a : e
................................

number of nodes in OGraph from M1p is 3
Terminal Node M2:
Content of marking:
p05ta : e
p13a : e
................................
Incorporation is NOT correct!
```

Figure 44. Incorporation error detected by the designer

In the second example, suppose the travel agent process tries to invoke operation "reserve" before "check" by mistake. Invoking these two operations in the above order violates the conversation protocol of the Foo airline partner service. Therefore, a token will get stuck in place
“pi2a” and the token in place “al_begin” will never be consumed when it is supposed to be, as is shown in the following picture.

Figure 45. The second example of incorporation error: invoking operation out of protocol synchronization.
The type of error can also be detected by running the protocol derivation on the service composition model. Where the series of OGraph analyses report a dead marking where one of the input places of the protocol contains a token, then it indicates that the protocol is not correctly incorporated into the composition. The following picture shows how the errors is detected by the designer:

```
<<<<<<<<<<< receive11 fires >>>>>>>>>>
Content of marking:
al_begin : e
plan : e
----------------------------------
Content of marking:
al_begin : e
plan : e
----------------------------------
Content of marking:
al_begin : e
plan : e
----------------------------------
<<<<<<<<<<< invoke1 fires >>>>>>>>>>
Content of marking:
al_begin : e
pi2a : e
----------------------------------
Content of marking:
al_begin : e
pi2a : e
----------------------------------

End of getOGraph
```

number of nodes in OGraph from M1p is 3
Terminal Node M2:
Content of marking:
al_begin : e
pi2a : e
----------------------------------
Incorporation is NOT correct!

Figure 46. Out-of-synch incorporation error detected by the designer
CHAPTER 9. CONCLUDING REMARKS AND FUTURE DIRECTIONS

In this dissertation, we have discussed four important topics in the field of service description and composition: achieving service dependability through the formal verification of critical programs used in service compositions, description of the service conversational protocol, incorporation of a partner exhibiting a complex conversation protocol into a larger composition, and automatic derivation of the conversation protocol from the service composition. Based on CPNet, we have introduced a unified service specification model and a framework for service design and analysis. We have shown how to extend the current service description with CPNet-based conversation protocols, how to incorporate a partner with a complex conversation protocol into a larger composition correctly. We have also proposed an approach to represent BPEL4WS programs with CPNets, which enables a formal verification of the correctness of compositions and an automatic derivation of conversation protocols. The models and approaches are supported by computer software tools we have developed, including the JCPNet verification toolkit and the WSPNet designer.

Currently, only a subset of BPEL4WS language has been mapped to CPNets model. Several advanced features of BPEL4WS, such as compensation handler and message correlation, have not yet been covered in this dissertation. Moreover, the conversation protocol derivation algorithm is just a starting effort in this direction. Because the derivation algorithm is based on CPNets occurrence graph analysis, the composition models that can be analyzed by the derivation algorithm are still limited. In the near future, we plan to investigate various mathematical CPNets
reduction techniques in order to design an algorithm which is able to simplify a wider variety of WSPNets.
REFERENCES


AT&T (2005). Graphviz - Graph Visualization Software.


S. Narayanan, S. M. (2002). Simulation, verification and automated composition of web services. WWW.


