INTEGRATED BUSINESS PROCESS ADAPTATION TOWARDS FRICTION-FREE BUSINESS-TO-BUSINESS COLLABORATION

A Dissertation in
Business Administration
by
Zhe Shan

© 2011 Zhe Shan

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2011
The dissertation of Zhe Shan was reviewed and approved* by the following:

Akhil Kumar  
Professor of Information Systems  
Dissertation Advisor  
Chair of Committee

Terry P. Harrison  
Professor of Supply Chain and Information Systems

Zan Huang  
Assistant Professor of Supply Chain and Information Systems

Dennis K. Lin  
University Distinguished Professor of Statistics and Supply Chain Management

John E. Tyworth  
Professor of Supply Chain Management  
Chair of the Department of Supply Chain and Information Systems

*Signatures are on file in the Graduate School
ABSTRACT

One key issue in process-aware E-commerce collaboration is the orchestration of business processes of multiple business partners throughout a supply chain network in an automated and seamless way. Since each partner has its own internal processes with different control flow structures and message interfaces, the real challenge lies in verifying the correctness of process collaboration, and reconciling conflicts in an automated manner to make collaboration successful. The purpose of business process adaptation is to mediate the communication between independent processes to overcome their mismatches and incompatibilities. This dissertation proposes a new framework and efficient algorithms for creating integrated adapters for business process composition. For the control flow adaptation, we propose a structural analysis of patterns which we show is more efficient than existing approaches. Furthermore, we propose algorithms based on message pair analysis and integer programming to create an optimal adapter in both synchronous and asynchronous communication. These algorithms are designed to handle more general cases involving $M$ messages and $N$ processes ($M, N > 2$). For message adaptation, we identify a set of extendible message adaptation patterns to solve typical message mismatches. Finally we show algorithms to integrate individual message adaptation patterns with control flow adapters to create a complete adapter for multiple processes. All algorithms were implemented in a Java-based prototype system, and results of validation test and performance experiments are reported. We compare and discuss the insights gained about adapter creation in different scenarios.
# TABLE OF CONTENTS

List of Figures vi
List of Tables ix
Acknowledgements x

Chapter 1 Introduction 1
  1.1 Background 1
  1.2 Scenarios and Motivating Examples 4
  1.3 Overview of Our Approach 8

Chapter 2 Pattern-based Analysis for Process Composition 12
  2.1 Process Interaction Patterns and Control Flow Mediation 12
  2.2 A formal definition of compatibility 14
  2.3 Pattern compatibility matrix 17
  2.4 Algorithms for compatibility verification and adapter creation 19
    2.4.1 Process normalization 20
    2.4.2 Compatibility verification algorithm 21
    2.4.3 Adapter creation algorithms 22
  2.5 SP adapter creation algorithm 24
  2.6 A heuristic algorithm for a minimal adapter (PBA-MIN) 26
  2.7 Summary 29

Chapter 3 Pair-wise Analysis for SP Adapter Creation 31
  3.1 Communication Modes in Process Composition 31
  3.2 Pair-wise Interaction 32
  3.3 Detection of Incompatible Cases 35
  3.4 Summary 36

Chapter 4 General Case of Synchronous Communication 38
  4.1 Incompatible Cases in 3-process Interaction 38
  4.2 The MIG Approach to Create a Minimal Adapter 40
    4.2.1 Message interaction graph (MIG) and the general case 40
    4.2.2 Using MIG for adapter creation 45
    4.2.3 Optimal adapter creation 47
  4.3 Summary 55

Chapter 5 Adapter Creation for Asynchronous Communication 57

Chapter 6 Analysis of Message Mismatches and Design of Message Adaptation Patterns 64
  6.1 Mismatches in Messages 65
  6.2 Standard Patterns of Message Adaptation 66
LIST OF FIGURES

Figure 1-1. An example to illustrate the need for an adapter in both synchronous and asynchronous communication ................................................................. 5

Figure 1-2. A motivating example for message adapter .................................................. 7

Figure 1-3. Framework for business process adaptation .................................................. 9

Figure 2-1. Basic process patterns (and their corresponding names used in WSBPEL) .... 13

Figure 2-2. An example of interaction between processes involving parallel and choice patterns .............................................................................................. 14

Figure 2-3. Example to show how to combine two communicating processes into a single process for verification ................................................................. 16

Figure 2-4. An example to show how S, P and SP Patterns can interoperate ..................... 19

Figure 2-5. An example of process normalization .............................................................. 20

Figure 2-6. Compatibility verification algorithm ............................................................... 22

Figure 2-7. Pattern-based adapter creation algorithm (PBA) ............................................. 23

Figure 2-8. Making adapters for choice and loop structures .......................................... 24

Figure 2-9. SP adapter creation algorithm (SPA) ............................................................. 25

Figure 2-10: An example to illustrate the need for a smallest adapter ............................ 26

Figure 2-11. Minimal SP adapter creation algorithm (SPA-MIN) .................................... 29

Figure 3-1. Synchronous and asynchronous modes of communication between processes ... 32

Figure 3-2. Compatible cases of a message pair interaction for both types of communications ........................................................................................................ 33

Figure 3-3. Incompatible cases of a message pair interaction ......................................... 34

Figure 4-1. Incompatible cases involving 3 messages and 3 processes ............................. 39

Figure 4-2. Illustration of inter-process causal dependency in the two types of communication .............................................................................................................. 39

Figure 4-3. Incompatible cases involving 2 messages and 2 processes in synchronous communication ........................................................................................................... 41

Figure 4-4. Incompatibility involving 3 messages and 3 processes ................................. 42
Figure 4-5. MIGs for the incompatible cases in Figure 4-3 and Figure 4-4.............. 43
Figure 4-6. A naïve algorithm for creating an adapter based on a MIG (MIG-Naïve) .... 47
Figure 4-7. An illustrative example of three interacting processes.......................... 48
Figure 4-8. MIG and detected cycles for the example in Figure 4-7.......................... 49
Figure 4-9. Greedy algorithm to select adaptation elements (MIG-Greedy)............. 50
Figure 4-10. An IP formulation for the example of Figure 4-7.................................. 52
Figure 5-1. An illustrative example for optimal adapter creation in async ................. 58
Figure 5-2. IP formulation for optimal adaptation message selection in async (Async-IP).... 59
Figure 5-3. An illustrative example for the Async-IP approach................................ 62
Figure 6-1. Illustrative example for message mismatch........................................... 64
Figure 6-2. Basic message adaptation patterns......................................................... 68
Figure 6-3. Tuple-to-Tuple with List-to-Element...................................................... 70
Figure 6-4. M-to-N pattern....................................................................................... 71
Figure 7-1. Illustrative example: Example 7-1.......................................................... 75
Figure 7-2. Output and input message forests ......................................................... 75
Figure 7-3. Illustrative example (Example 7-2)....................................................... 77
Figure 7-4. Illustrative example (Example 7-3)....................................................... 78
Figure 7-5. Message adapter for Example 7-2......................................................... 83
Figure 7-6. messageAdapters array......................................................................... 84
Figure 7-7. New composition scenario in Example 7-4............................................ 86
Figure 7-8. Communication graph in Example 7-4.................................................. 86
Figure 7-9. MIG analysis in Example 7-4................................................................. 87
Figure 7-10. Integrate adapters in Example 7-4....................................................... 87
Figure 8-1. Architecture of BPCT............................................................................ 90
Figure 8-2. Performance comparison of the MIG-greedy and MIG-IP algorithms....... 100
Figure 8-3. Optimizing the adapter for message size .................................................. 103
Figure 8-4. Groups of security profile by flexibility ................................................... 105
Figure 8-5. Adding security and privacy constraints .................................................. 105
Figure A-1. Johnson’s Algorithm ........................................................................... 141
Figure B-1. Example to illustrate the slimming of adapter process ......................... 142
Figure B-2. Updates of the relationship table for the example in Figure B-1 ............. 144
LIST OF TABLES

Table 2-1. A pattern compatibility matrix................................................................. 18

Table 3-1. Rules for detecting Type A and B incompatibilities for synchronous communication (2 messages m_i & m_j; 2 processes P & Q)......................................................... 36

Table 3-2. Rules for detecting Type C and D incompatibilities for asynchronous communication (2 messages m_i & m_j; 2 processes P & Q)......................................................... 36

Table 6-1. Message mismatches in the case of Figure 6-1........................................ 65

Table 7-1. Output message for Proc 1........................................................................... 79

Table 7-2. Input messages for Proc 2............................................................................ 79

Table 7-3. Output messages for Proc 2........................................................................... 79

Table 7-4. Input messages for Proc 1............................................................................ 79

Table 7-5. Output message elements for Proc1.............................................................. 79

Table 7-6. Input message elements for Proc2............................................................... 80

Table 7-7. Output message elements for Proc2.............................................................. 80

Table 7-8. Input message elements for Proc1............................................................... 80

Table 7-9. Matching table for Example 7-2................................................................. 80

Table 7-10. Updated matching table for Example 7-2.................................................. 81

Table 8-1. Results for the validation test...................................................................... 93

Table 8-2. Results of performance experiments......................................................... 98

Table 8-3. Adapter size comparison in synchronous and asynchronous communication...... 101

Table 9-1. Comparison of the verification approaches for process composition............ 114

Table 9-2. Comparison of the formal methods............................................................ 130

Table 10-1. Comparison of adapter creation in Sync vs. Async communication........... 136
I am deeply indebted to my dissertation advisor, Akhil Kumar, for his guidance and patience during all years of my study in the Ph.D. program. Through the numerous meetings, I have not only been able to push my dissertation forward, but also learned how to appreciate and do good research. I would never have been able to finish my dissertation without his guidance.

I am also very grateful to Terry Harrison, the member of my committee, for exposing me to important concepts and algorithms, for discussing and answering all my questions, and for passionately encouraging me to further my work. I would also like to thank my other committee members: Zan Huang and Dennis Lin, for taking the time to read my work, for their insightful comments, and for being supportive throughout the time.

My special thanks to Saurabh Bansal and Christopher Craighead for providing useful suggestions, and other support for part of my dissertation. I would also like to thank Susan Xu for her continuing encouragement and advice, not only for research, but also for life.

I would like to thank Jiayi Li, Valery Pavlov, Kaan Kuzu, Zheng Jia, Jianfeng Guo, Mrinmay Deb, Paolo Letizia, Cheng Nie, and Bernardo Quiroga Gomez, who as good friends, were always willing to help and give their best suggestions. My research would not have been possible without their helps. And, it would have been a lonely journey without their companion.

My heartfelt thanks to Beth Bower, Teresa Lehman, Alice Young, and Janie Jones for all their support and for making going to office everyday a warm and pleasant experience.

Last but not least, I would like to thank my parents for their unconditional love and support, without whom I could not have achieved almost anything that I have achieved so far.
Chapter 1

Introduction

1.1 Background

Organizations rely on effective supply chains to successfully compete in the global market. Successful supply chain management integrates individual functions within an organization or across different organizations. Supply chain business process integration involves close collaboration among supply chain partners. Business collaboration seeks to organize entire business processes throughout a value chain of multiple parties. In recent years, the emphasis in supply chain collaboration has shifted from matching the inputs and outputs of the processes to gaining a deeper understanding of the process semantics of individual players. Moreover, market forces demand an ability to make quick changes on-the-fly in interacting with supply chain partners. This variability has significant effect on the design and configuration of the processes. Therefore, analysis of the correctness and robustness of business collaboration has become one key determinant of supply chain success.

During the past decades, information technology has enabled organizations to successfully operate business processes. Many process languages and systems have been developed. Some prevailing ones include XML Process Definition Language (XPDL) [Workflow Management Coalition 2008], Business Process Modeling Notation (BPMN) [Grosskopf et al. 2009], and Yet Another Workflow Language (YAWL) [van der Aalst and ter Hofstede 2005]. With the emergence of service computing and cloud technologies [Zhang et al. 2007], more and more enterprises have migrated their platform into a service-oriented architecture to achieve more agility [Zhao et al. 2007]. As an example of an implementation approach, Web services [W3C
2009] are loosely coupled, distributed reusable components that encapsulate discrete functionality and are accessible over standard XML-based internet standards such as UDDI [OASIS UDDI Specifications TC 2004], WSDL [W3C Web Services Description Working Group 2001], and SOAP [W3C XML Protocol WG 2007]. Based on this technology, the functionality of a business process or operation can be provided in a unified interface, so that they can be searched, and then reused or consumed in an automatic and flexible way. Furthermore, to break the limit of simple operation models in WSDL, advanced Web Services languages such as Business Process Execution Language (WS-BPEL) [OASIS WSBPEL TC 2007] and Web Services Choreography Description Language (WS-CDL) [W3C 2005] provide XML based process definitions to specify communication protocols. These technologies lead to the possibility of automated, "on-the-fly" composition of multiple business processes in inner- or cross-organization interaction.

While such standardization efforts simplify interoperability, they still do not guarantee that business processes can always be connected without conflicts. Dissimilar design practices and organization models lead to differences in the communication interfaces of business processes in interaction. The traditional approaches based on process re-design or re-engineering cannot fulfill these new requirements due to their long lifecycle and high cost. By comparison, creating adapters to enable conflict-free interaction is a much more efficient and lower cost solution. In recent years, this field has started to attract more attention from researchers [Brogi and Popescu 2006; Canal et al. 2008; Kumar and Shan 2008; Seguel et al. 2010]. However, the existing approaches in literature have a number of limitations: 1) Most of them adopt a formal model, such as Petri Nets, process algebra, etc., in their analysis. These state transition systems suffer from a state explosion problem when analyzing large-scale cases. 2) They generate either a full (all messages are adapted) or non-optimal adapter; thus their solution is not the most efficient one. 3) Most of them can only handle the cases involving two processes, instead of more general cases involving multiple processes, which are very common in real-world business scenarios. 4)
They only consider the number of messages in adaptation, and ignore other factors, such as message size, and security and privacy constraints, which are very important for the design of the adaptation solution.

Moreover, the mismatches among multiple processes are categorized into two classes, i.e. signature (message format) and protocol (control flow) [Kongdenfha et al. 2009; Motahari Nezhad et al. 2007; Wang et al. 2008]. However, most of the adaptation research in literature focused on either the control-flow aspect [Brogi and Popescu 2006; Canal, et al. 2008; Kumar and Shan 2008; Seguel, et al. 2010; Shan et al. 2010; Yellin and Strom 1997] or the message aspect [Dong et al. 2004; Dumas et al. 2006; Wang and Stroulia 2003]. There is very limited research involving integration of both control flow and message aspects in inter-process adaptation. In [Kongdenfha, et al. 2009; Motahari Nezhad, et al. 2007; Wang, et al. 2008], the authors tried to propose comprehensive solutions covering both aspects, but there are many limits: 1) The control-flow adaptation and message adaptation are still treated in an independent way, and their interplay is not recognized. 2) The explicit specification of the adapter process is missing, which makes it difficult to verify the correctness of the process composition at design stage. 3) They didn't intend to minimize the adaptation effort among interacting processes. 4) Their implementations require special system extension or component, which are not available in most of existing process engine products.

Consequently, in this dissertation we address the problem of how to develop a new framework and efficient algorithms to ensure that independent business processes (or composite Web services) can be composed "correctly". Our notion of correctness is soundness [van der Aalst 1997]. Methods for checking a single process are well-known [Kiepuszewski et al. 2000; Verbeek et al. 2001]. For the control flow adaptation, we propose a structural analysis of patterns which we show is more efficient than existing approaches. Furthermore, we propose algorithms based message pair analysis and integer programming to create an optimal adapter in both
synchronous and asynchronous communication. These algorithms are designed to handle more general cases involving $M$ messages and $N$ processes ($M, N > 2$). For message adaptation, we identify a set of extendible message adaptation patterns to solve typical message mismatches. Finally we show algorithms to integrate individual message adaptation patterns with control flow adapters to create a complete adapter for multiple processes.

In next section, we introduce several examples to illustrate the scenarios in which adaptation enables successful process composition.

1.2 Scenarios and Motivating Examples

To motivate the need for an adapter, consider for instance Figure 1-1 (a) which shows snippets from a flight booking example involving a client and an agent, each running its own process. The client first sends the date and city information ($date \ info$ and $city \ info$) in sequence to the agent. Then, the client sends another message containing his customer number ($customer \ #$), while the agent sends the flight information ($flight \ info$). Subsequently, the client receives the $flight \ info$ and sends the passenger information ($passenger \ info$), for one passenger at a time. Correspondingly, the agent receives the $customer \#$ and then the $passenger \ info$. If the flight tickets are available for booking, the agent sends the $ticket$ and $itinerary$ to the client. Otherwise, it sends out the $booking \ failure$ notification. Thus, the $date \ info$ is received after sending the $city \ info$, even though it is sent earlier in the client process. It shows that, assuming synchronous communication and no buffering of messages, these two processes can work together only if an adapter process is added as shown in Figure 1-1 (b) to ensure proper operations. Note that in the absence of an adapter, the processes would get stuck at $Send \ date \ info$ and $Receive \ city \ info$, and would not be able to proceed. Therefore, we can create an adapter (Adapter 1 in Figure 1-1 (b)) to
buffer *date info* and *customer #* from the client, and send them to the agent later. The creation of Adapter 1 will be discussed in more detail later.

![Figure 1-1. An example to illustrate the need for an adapter in both synchronous and asynchronous communication](image)

The example in Figure 1-1 (a) highlights the main idea behind compatible composition and the need for adapter creation. Another factor we need to consider in the adapter creation is the communication mode between the two processes. Generally there are two communication modes, i.e. **synchronous** and **asynchronous**. In synchronous communication, a *send* activity blocks its process until its corresponding/dual *receive* activity gets the message. However, in an asynchronous communication the *send* activity does not block after sending a message, and the process continues. Such a communication mode requires that a process execution engine have a
buffer mechanism to store incoming messages. The discussion of the composition compatibility and adapter creation in Figure 1-1 (a) and (b) is based on a synchronous communication mode. In an asynchronous communication mode, usually a buffer mechanism is implemented by one or multiple FIFO queues for incoming messages. The process execution engine may use one single queue to buffer all messages sent to a specific process. Or, it may equip multiple queues, one for each different message or message type. These communication modes matter to the compatibility verification and adapter creation of the process composition.

Consider again the example in Figure 1-1 (a). Suppose each process instance has a single queue to buffer incoming messages. When the client sends date info and city info in sequence, these two messages are received and buffered in a sequential order in the agent queue. This order is different from the order in which the agent receives these two messages. In order to avoid the possible deadlock, we need to create an adapter to receive date info and city info from the client process, and then send city info before date info to the agent process. Afterwards, when the client sends out the customer #, the message is buffered in the agent queue. Hence the successful transmission of this message does not block the transmission of the flight info. Therefore, we do not need Adapter 1 in the asynchronous communication mode. Nevertheless, we do need an adapter for ticket and itinerary. This is because the agent sends ticket and itinerary in parallel, which means they can reach and be buffered in the client queue in any order. However, in the client process Receive ticket is executed before Receive itinerary. This requires that ticket should be placed before the itinerary in the client queue. In order to avoid the possible deadlock, we need to create an adapter to receive ticket and itinerary in parallel from the agent process, and then send ticket before the itinerary to the client process. The complete adapter for asynchronous communication is shown in Figure 1-1 (c). This example shows the impact of communication modes on the adapter creation problem.
In the example in Figure 1-1, we assume that the messages are well matched. However, in real cases different processes may have different definitions on message interfaces. To motivate the need for message adaptation, consider for instance Figure 1-2 that shows snippets of two processes from a travel service. A customer might wish to communicate with a flight order service offered by an airline company by sending passenger name (name) and flight number (FL#) in a tuple \( T_1 \), and credit card number (CC#) and expiration date (expire) in another tuple \( T_2 \). She wishes to execute the two activities in parallel using the \textit{parallel} pattern labeled P in Figure 1. In general, the parallel pattern contains two (or more) branches that can be interleaved. The interface of the flight service, on the other hand, requires all the information in a single tuple \( T \). Therefore, without the information required by its interface in the right format, the service cannot fulfill the request from the client. In this case, the messages provided in the customer process are not compatible with the messages required in the service process. We call this type of interface incompatibility as \textit{message mismatch}. Moreover, the activity structure in customer process (send \( T_1, T_2 \) in parallel (P)) is not compatible with the structure in the service process (receive a single tuple \( T \)). We call this interface incompatibility \textit{control flow mismatch}. A possible adapter is shown in the center of Figure 1-2. And, it will be discussed in more detail later.

Figure 1-2. A motivating example for message adapter
In next section, we give an overview for the major contributions of this dissertation.

1.3 Overview of Our Approach

The purpose of business process adaptation is to mediate the communication between client and service processes to overcome the mismatches in the control flow and the message aspects. Our goal in this dissertation is to develop a framework and techniques for creating highly flexible, dynamic, general purpose adapters that can be written on-the-fly to allow interaction between multiple processes. This framework can integrate two major dimensions of adaptation, i.e. message adaptation and control flow adaptation. In our framework (Figure 1-3), three forms of adaptation are:

- **Message adaptation** transforms the format (data type) of messages between a service consumer and a service provider while leaving the protocol as is (except for the indirection via the adapter).

- **Control flow adaptation** mediates the message exchange protocol by altering the order of messages and/or changing the number of messages, without changing the format of the messages.

- **Integrated adaptation** combines both message adaptation and control flow adaptation, which is the focus of this work.
In the control flow aspect, we aim at developing an efficient algorithm to generate an optimal adapter that reconciles differences in control flow of two (or more) interacting processes to make them compatible. In this part, we assume that message format mismatches do not exist between processes. We start with a novel approach based on structural analysis of process patterns (Chapter 2), which is more efficient than existing approaches. Then, we decompose the interaction of two processes to the interaction of each message pair. By exhaustively checking all the communication scenarios for one message pair, we identify incompatibility cases for both synchronous and asynchronous communication (Chapter 3). Finally, we extend this analysis into more complicated cases involving $M$ messages and $N$ processes ($M, N > 2$). Further, to create an optimal adapter for synchronous messages we introduce the idea of an adapter creation algorithm based on message interaction graphs (MIG) to capture all dependencies among messages exchanged among multiple processes (Chapter 4). For asynchronous messages, we propose a new algorithm based on another optimization model (Chapter 5). In both algorithms, we detect all incompatible cases, identify potential adaptation elements, and integrate them into a complete adapter.
Turning to the message adaptation aspect, our approach is based on building and using message adaptation patterns (Chapter 6). At message adapter creation time we can detect any necessary patterns or their combination, and apply them to invoke a variety of message transformation. Another interesting feature of our approach is that the patterns are extendible. Thus, a user may add more message transformation patterns and define their relation with existing ones. Also we show how message adapters can be created on the fly for matching parts of two services where basic adapter patterns do not exist (Chapter 7). Having analyzed the issues in adapter creation from the control flow and message flow perspectives separately, we then show how these two types of adaptation techniques can be combined to create an integrated adapter (Chapter 7).

This dissertation contributes to the state-of-the-art of the field in the following major aspects,

1. The pattern-based control flow adaptation approach decomposes the adaptation effort into a number of adapter creation problems in sub pattern levels. It avoids the state explosion problem in analyzing large-scale cases.

2. The compatibility verification and adapter creation approach introduced in this work can handle general cases involving any number of processes and messages, which are very common in real-world business scenarios.

3. The adapter creation algorithms generate an optimal adapter in both synchronous and asynchronous communication. They minimize the communication effort, and also protect privacy and security of business parties at a maximum level. The performance experiments show that the size of generated adapters by our algorithms decreases more than 70% on average compared with the existing SET algorithm in literature.
4. The extended MIG-IP method can consider not only the number of adapted message, but also other factors, such as message size, and security and privacy constraints. This makes it possible to design a complete adaptation solution.

5. We recognize the interplay between control-flow adaptation and message adaptation, and treat them in an integrated way. Our approach generate an explicit adapter process covering both control flow and message adaptation, which makes it possible to verify correctness and improve performance of a process composition at design stage.

Consequently, the plan of this dissertation is as follows. Chapter 2 presents the pattern-based analysis for compatibility verification and adapter creation. Chapter 3 introduces the idea of pair-wise analysis and gives results. Chapter 4 and Chapter 5 show how we can extend this analysis to $M$-message and $N$-process cases, and create an optimal adapter for both communication modes. Chapter 6 describes some basic message adaptation patterns. Then, Chapter 7 gives our integrated adapter creation algorithms in detail. Chapter 8 presents our prototype implementation, and the results of validation tests and performance experiments. Chapter 9 presents a comprehensive comparison on the existing verification methods for Web services composition, and also discusses related works in adapter creation and message adaptation. Chapter 0 concludes this dissertation, and presents my research output.
Chapter 2

Pattern-based Analysis for Process Composition

In this section, we propose a novel approach based on pattern analysis to create a control flow adapter for two processes.

2.1 Process Interaction Patterns and Control Flow Mediation

Web services [W3C 2009] are loosely coupled, distributed reusable components that encapsulate discrete functionality and are accessible over standard XML-based internet standards such as UDDI [OASIS UDDI Specifications TC 2004], WSDL [W3C Web Services Description Working Group 2001], and SOAP [W3C XML Protocol WG 2007]. In general, atomic services are provided by multiple different providers, and a user wishing to use several related services would compose them into a process that coordinates the order in which the different services are invoked to create a composite service.

A composite service, say, one described in WSBPEL, is a process, which interacts with other processes [Benatallah et al. 2005; Brogi and Popescu 2006; Motahari Nezhad, et al. 2007; Nakajima 2002]. A process is composed of basic patterns or building blocks like sequence (SEQ or S), parallel (AND, A or P), choice (XOR or X) and loop (L) as shown in Figure 2-1. These patterns can be combined to create processes in WSBPEL. The parallel pattern contains two (or more) branches that can be interleaved, while the choice pattern contains multiple branches of which only one is selected at run time. Finally, the loop pattern contains one forward branch and one backward branch that may be repeated multiple times. For two interacting loop patterns, their iteration counters should be equal, or their loop conditions should be identical. Each branch may be a single activity, or in general, it may represent a child process.
As an example, consider Figure 2-2 where each process is composed of two basic patterns: parallel and choice which are in turn connected with one another in a sequence. After the products have been ordered by the retailer, the retailer process waits to receive an invoice and also the goods in any order. Similarly, the manufacturer process may send the invoice and goods in any order. Upon receiving the goods and invoice, the retailer may make the payment either by check or by visa. The manufacturer process will receive the corresponding payment. Also, note that the Verify and Approval steps are internal steps in each process and they involve no interaction with the other process. They are disregarded in verification.

This figure shows that the two processes can interoperate correctly because: (1) each pattern in the retailer process has a matching pattern in the manufacturer process and vice versa; (2) every send action in a pattern has a corresponding receive action in its corresponding pattern; (3) there is no deadlock.

In this dissertation we assume that each process is well structured [Kiepuszewski, et al. 2000], i.e. each split for a pattern has a corresponding join, and also that the patterns are properly nested inside one another. For the discussion of control flow adaptation in Chapter 2, Chapter 3, Chapter 4 and Chapter 5, it is assumed that the messages in interacting processes are well matched, i.e. each send activity in process 1 has a corresponding receive activity in process 2, and
vice versa. These two activities are dual with each other. Additionally, we assume that each send or receive activity is unique and is not repeated in the process.

Figure 2-2. An example of interaction between processes involving parallel and choice patterns

2.2 A formal definition of compatibility

Creating or modifying a complex process that combines parallel and conditional routing is a task subject to errors. A careless design or change of an activity can cause a deadlock or livelock. A deadlock occurs if at some unexpected point in the business process it is no longer possible to make any progress for a certain case (process instance, i.e. an execution of a process) that is being handled. A livelock occurs if it is possible to make continuous progress for a certain case, however, without progressing towards successful completion and without ending in a deadlock (e.g., an endless loop). The widely accepted notion for correctness of a single business process is soundness, which is introduced in [van der Aalst 1997]. This notion expresses the minimal requirements any process should satisfy. Informally, it can be defined as follows,
Definition 2-1. A business process is *sound* or *correct* if it satisfies the following conditions [Verbeek, et al. 2001],

- **Option to complete**: It should always be possible to complete a case (process instance) that is handled according to the process. This condition guarantees the absence of deadlocks and livelocks.

- **Proper completion**: It should not be possible that the business process signals completion of a case while there is still work in progress for that case.

- **No dead tasks**: For every activity, there should be an execution of the business process that executes it. This restriction means that every task has a meaningful role in the business process.

Our formal notion of compatibility is based on combining two processes (and in general $N$ processes) into a single process while maintaining the correct synchronous communication semantics between them. Then we check the combined process for correctness using existing techniques [Eshuis and Kumar 2010; Sadiq and Orlowska 2000]. First we show how to combine two processes by a construction procedure.

**Procedure 2-1: (Process Combination Construction)**

1) Add a new node *Start* and connect it to an AND-Split node.

2) Connect the AND-split node to the start nodes *start1* and *start2* of the two processes.

3) Add a new node *End* and connect an AND-Join node to it.

4) Connect the end nodes *end1* and *end2* of the two processes to the AND-Join node.

5) For each *receive* activity (rcv$_i$), insert an AND-Split (AND-S$_i$) node into its in-link.

6) For each *send* activity (snd$_i$), insert an AND-Join (AND-J$_i$) node into its in-link.

7) Add a link from AND-S$_i$ to AND-J$_i$. ■

Figure 2-3 illustrates how to apply this procedure for the example of Figure 1-1. Figure 2-3 (a) shows the two (abbreviated) individual processes from Figure 1-1, and Figure 2-3 (b)
shows the combined process. In general, this approach can be applied to combine \( n \) processes into a single process. Next, we define the compatibility of process composition.

**Definition 2-2:** Multiple processes are **compatible** if the combined process resulting from applying the construction procedure above is correct.

![Diagram](image)

Figure 2-3. Example to show how to combine two communicating processes into a single process for verification

**Lemma 2-1:** The construction procedure maintains the send-receive semantics of synchronous communication between two processes.

**Proof:** In synchronous communication between process 1 and 2, a send activity in process 1 can only be performed if a receive activity in process 2 is ready (and vice versa), i.e. it should be possible to perform both activities in parallel. By adding an AND-Split node just before a receive activity and connecting it to the corresponding send activity, our construction ensures that the receive activity must be ready before the send activity is performed. Hence, the send and receive pair of activities can be synchronized. ■
In general, this approach can be extended to compose multiple processes. Each process in our approach is represented as an AND/OR graph. Techniques for checking correctness of such graphs are discussed in [Eshuis and Kumar 2010; Sadiq and Orlowska 2000]. Therefore, the same techniques can be applied to check the combined process graph which is also an AND/OR graph. If the combined process graph is correct, it means that no adapter is required for these communicating processes. Otherwise, an adapter can be created. To check the compatibility between the processes and the created adapter, we can combine them using the same construction approach and then check the newly combined process for correctness. If this process is correct it means that the adapter is correct.

2.3 Pattern compatibility matrix

The basic patterns described above, like sequence, choice, parallel and loop, can combine to create composite patterns. A pattern consisting of only sequence and parallel structures is called an SP pattern. When processes interact, clearly one basic requirement for compatibility is that each send in pattern 1 must have a corresponding receive in pattern 2, and vice versa; else the two processes cannot interoperate. However, as noted earlier this rule by itself does not guarantee interoperability. Our goal here is to look for ways to check interoperability at the structural level first before conducting an analysis at the message level. Hence, we introduce the notion of pattern compatibility. Table 2-1 is a pattern compatibility matrix showing compatibility between patterns in different processes. When processes interoperate, the patterns inside them must also interoperate according to the rules in this matrix, i.e. they should have corresponding or matching patterns.
Below we give a simple result to capture the essence of this matrix.

**Result 2-1**: The following compatibility rules (see Table 2-1) must be observed between interacting patterns of different processes:

1. An SP pattern from process 1 must match with an SP pattern from process 2.
2. An X pattern from process 1 must match with an X pattern from process 2.
3. An L pattern from process 1 must match with an L pattern from process 2.

**Proof Sketch.** We argue in terms of action sets, i.e. all the atomic actions in a pattern. An S and P pattern can match each other if they contain the same action set because all actions are executed. X cannot match with S, P or L patterns because only one branch of an X can be executed, and so the action set of the executed branch will not be the same as the action set of the other patterns. Finally L can only match with an L because its action set can be repeated multiple times. Hence, Table 2-1 is valid. Moreover, this is a necessary condition for process compatibility.

**Example 2-1**: In Figure 2-2, the two parallel patterns in the retailer and manufacturer processes match because the *send invoice* and *receive invoice* activities correspond to each other (as do, *send products* and *receive products*). Similarly, the choice patterns also match because their branches have corresponding matching branches.

The next example shows how two SP patterns can interoperate with each other.

**Example 2-2**: Figure 2-4 shows process snippets to illustrate interoperation between a sequence and SP patterns. After receiving the retailer's *order*, the manufacturer *sends* the *invoice*...
and the ship information (shipping info) in parallel; however, the retailer can receive them in sequence only as shown. Thus, an adapter must be created. The adapter contains two activities in sequence, receive ship info and send ship info.

![Figure 2-4. An example to show how S, P and SP Patterns can interoperate](image)

The compatibility rules reduce the search space considerably, and in the next section we apply these rules to develop algorithms to verify that two processes can interoperate, and create an adapter for them if necessary.

### 2.4 Algorithms for compatibility verification and adapter creation

Now we apply the above rules to design algorithms to first verify if two processes are compatible and then to generate an adapter if one is required. Here we first introduce the process normalization algorithm, which is executed on the original processes before they are passed to the compatibility verification and adapter creation algorithms. Then we present the verification algorithm and the adapter creation algorithms.
2.4.1 Process normalization

The motivation of the process normalization is to first make the processes structurally similar for verification. The original processes may contain both private activities, which are internal within a company, and public activities, which are communications with external parties. When checking compatibility, we only need to consider the public activities. Therefore, we first delete all private activities in both processes. After this deletion, the structure of processes may be changed. Hence, we need to reorganize the corresponding patterns according to this procedure:

Procedure 2-2: (Process Normalization)

1) Delete private or internal activities.
2) If a non-loop parent is left with only one child, then simplify by removing the parent.
3) Merge immediately nested identical, non-loop patterns (e.g. if parent and child are both X)
4) If two or more branches of an X pattern are identical, then merge them into one.

Example 2-3. Figure 2-5 shows an example of the normalization procedure. In this example, it first deletes the internal activities $t1$, $t2$ and $t3$; then removes the single-child parallel structure; and last, combines two nested choice structures into one.

Figure 2-5. An example of process normalization
2.4.2 Compatibility verification algorithm

To verify if two processes are compatible, first we need to check whether the two sets of activities, one for each process respectively, can communicate with each other, i.e. for each activity of a process, its dual (or corresponding) activity exists in the other process. Recall that the dual of a *send* activity is a *receive*, and vice-versa. After that, we need to check whether there is any deadlock in the communications between two processes. Then, we turn to choice and loop patterns. We handle these structures by decomposing them. A choice contains multiple branches, each of which can be viewed as a subprocess. Similarly, a loop contains forward and backward branches (or subprocesses) that are repeated 0 or more times depending upon the loop condition. As discussed above, a choice in one process must have a matching choice pattern in the other process, and a loop should likewise have a matching loop pattern. In this algorithm, the handling for choice and loop are similar. We find all the loop or choice patterns in each process, and then check whether their corresponding branches are compatible too. Therefore, the algorithm will be executed recursively until all loop and choice patterns are verified. Figure 2-6 gives the full algorithm. This algorithm is called twice, with the arguments P1 and P2 interchanged the second time. If it returns 'True' both times, it means that P1 and P2 are compatible.

As part of the verification algorithm, the deadlock detection algorithm (*deadlock_exists*) first transforms processes P1 and P2 into a directed graph. In addition to the existing links in P1 and P2, we create links from each *send* activity to its corresponding *receive* activity. In this directed graph, we check for cycles, and their absence means P1 and P2 are deadlock free. In general, this idea can be extended to any number of processes.
Figure 2-6. Compatibility verification algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01:</td>
<td>foreach (activity t1 in P1)</td>
</tr>
<tr>
<td>02:</td>
<td>if (not exists t2 in P2 s.t. t2 == dual(t1))</td>
</tr>
<tr>
<td>03:</td>
<td>return(False);</td>
</tr>
<tr>
<td>04:</td>
<td>if (deadlock_exists(P1, P2))</td>
</tr>
<tr>
<td>05:</td>
<td>return(False);</td>
</tr>
<tr>
<td>06:</td>
<td>if ((P1.type == 'X'</td>
</tr>
<tr>
<td>07:</td>
<td>if (not exists P2.branch1', P2.branch2' s.t.</td>
</tr>
<tr>
<td>08:</td>
<td>P2.type = P1.type &amp;&amp; P1.loop_count = P2.loop_count</td>
</tr>
<tr>
<td>09:</td>
<td>&amp;&amp; verify(P1.branch1, P2.branch1')</td>
</tr>
<tr>
<td>10:</td>
<td>&amp;&amp; verify(P1.branch2, P2.branch2'))</td>
</tr>
<tr>
<td>11:</td>
<td>return(False);</td>
</tr>
<tr>
<td>12:</td>
<td>else</td>
</tr>
<tr>
<td>13:</td>
<td>{</td>
</tr>
<tr>
<td>14:</td>
<td>foreach (child X1 in P1)</td>
</tr>
<tr>
<td>15:</td>
<td>if (not exists X2 in P2 s.t. verify(X1,X2))</td>
</tr>
<tr>
<td>16:</td>
<td>return(False);</td>
</tr>
<tr>
<td>17:</td>
<td>foreach (child L1 in P1)</td>
</tr>
<tr>
<td>18:</td>
<td>if (not exists L2 in P2 s.t. verify(L1,L2))</td>
</tr>
<tr>
<td>19:</td>
<td>return(False);</td>
</tr>
<tr>
<td>20:</td>
<td>}</td>
</tr>
<tr>
<td>21:</td>
<td>return(True);</td>
</tr>
</tbody>
</table>

2.4.3 Adapter creation algorithms

Besides verifying that two processes are compatible, we also need to check whether they need an adapter to achieve successful communication. In this section, we introduce the pattern-based algorithm (PBA) for adapter creation in synchronous communication. The adapter creation algorithm works bottom up and is discussed next (see Figure 2-7).

We first discuss the treatments of the choice and loop patterns. We start with the innermost loop or choice patterns XL1 and XL2 in each process (line 2), and create an adapter for this structure (line 5-12). Then we find the next innermost loop or choice structure (line 16) and find an adapter for it, and so on. While matching choice patterns, we create individual adapters for each unique pair of branches by invoking CreateSPAdapter (the SP adapter creation algorithm to be described in Section 2.5) (line 8), and then combine these adapters in a choice pattern themselves (see Figure 2-8 (a)) (line 11). For loop patterns, forward and backward branches are respectively matched, and the created adapters are placed inside a loop as shown in
Figure 2-8 (b). After an adapter is created for a choice or loop pattern, this subprocess is replaced by a single block with a new block id in the parent process (\texttt{BlockReplace} function) (line 13-14). The matching block id's and their adapters are stored in an array (\texttt{blockAdapter}) for later reference (line 15). This procedure is repeated for all the choice and loop patterns, until an adapter is created for the outermost one (line 3). Thus, the remaining process will be left with only sequence and parallel patterns (and blocks with ids) to which the SP adapter algorithm is applied again and a new adapter is returned (line 18). If the returned adapter is null, the two processes are compatible without the need of adapter. Note that function \texttt{FindMatchedBranchPair} matches corresponding branches of the two (sub)processes represented by its arguments and returns one matching branch pair at a time. The match must ensure that each activity in one branch has a dual in the other branch.

```plaintext
CreateAdapter(Proc P1, Proc P2) // Pattern-based adapter creation algorithm (PBA)
01: blockAdapter[][]={} //initialize the block adapter array
02: Find innermost matched X/L patterns XL1 and XL2;
03: while ((XL1,XL2)! = null)
04: {
05:   Adapter={}; branchAdapterSet={}; //initialize for pattern adapter
06:   foreach ((b1,b2)= FindMatchedBranchPair(XL1,XL2)) //each branch pair (b1, b2)
07:     {
08:       branchAdapter = CreateSPAdapter(b1,b2);
09:       branchAdapterSet = branchAdapterSet ∪ branchAdapter;
10:     }
11:   if (XL1.type == 'X') Adapter = Xor(branchAdapterSet); //merge Xor
12:   if (XL1.type == 'L') Adapter = Loop(branchAdapterSet); //merge Loop
13:   block1 = BlockReplace(P1,XL1); //generate block1 to replace XL1 in P1
14:   block2 = BlockReplace(P2,XL2); //generate block2 to replace XL2 in P2
15:   blockAdapter[block1.ID][block2.ID] = Adapter; //store adapter in an array
16: Find next innermost matched X/L patterns XL1 and XL2;
17: }
18: return CreateSPAdapter(P1, P2, blockAdapter);
```

Figure 2-7. Pattern-based adapter creation algorithm (PBA)
2.5 SP adapter creation algorithm

For ease of presentation, we discuss the CreateSPAdapter algorithm separately. This algorithm is shown in Figure 2-9 and assumes that the two processes contain only S and P patterns. It works by simulating the execution of the two processes to create an adapter for S/P patterns. It starts with two lists of current activities (currList1 and currList2) that are ready to run in the two processes (p1 and p2), and a pool that is initially empty (line 2-6). It matches any send-receive pair or block pair in the current lists and removes them (line 9-11). If they are blocks, it adds their adapter which is already stored in an array to a set name blockAdapterSet (line 12). Then, if there is any entry in the pool that matches one in a current list, the pair is removed (line 16-18). If the entries are atomic then they are added to the adaptActivitySet set (line 20). If they are blocks, their adapter will be put in blockAdapterSet (line 22). Similarly, if there are any unmatched send activities or blocks in a current list, they are added to the pool when no activity is matched in a given round (line 25-27). The dual of a send entry, i.e. a receive, is also added to the adaptActivitySet (line 28). Then the entries in adaptActivitySet are combined in parallel and added in sequence to the adapter being created (line 31). The algorithm runs until the current lists are
empty (line 39). Last, the created adapter and all the adapters in blockAdapterSet are put inside a parallel pattern to make the final adapter, which is also a process itself (line 41).

```java
CreateSPAdapter(Proc P1, Proc P2, array blockAdapter[][]) //SP adapter creation algorithm (SPA)
01: //initialize structures
02: pool={}; adapter={}; doneList1={}; doneList2={};
03: currList1={child activities that are ready to execute in P1};
04: currList2={child activities that are ready to execute in P2};
05: adaptActivitySet={}; //set for activities to be added in this round
06: blockAdapterSet={}; //set will contain required block adapters
07: 08: repeat{
09: foreach (a1 in currList1, a2 in currList2) // match current lists
10: { match activities and blocks in currentlist1 with currentlist2;
11: if (match found) { remove from current list; add to done list;}
12: if (two blocks match) { add adapter for matching blocks to blockAdapterSet;}
13: }
14: 15: foreach (a1 in {currList1, currList2}, a2 in pool) //match w/pool
16: { match activities and blocks in currentlists with pool;
17: if (match found)
18: { remove them from currentlist and pool;
19: add to doneList1, doneList2;
20: add a2 to adaptActivitySet;
21: }
22: if (two blocks match) { add adapter for matching blocks to blockAdapterSet;}
23: }
24: 25: foreach (a in {currList1, currList2}) //Add unmatched to pool
26: { if ((a.mode == 'send')||(a.type != null))
27: { add a to pool; remove a from currList; add a to doneList;
28: add dual(a) to adaptActivitySet;
29: }
30: 31: adapter = Seq(adapter, And( adaptActivitySet ));
32: //put activities in adaptActivitySet in And pattern and append
33: 34: adaptActivitySet = {}; // reinitialize adaptActivitySet now
35: update currList1 with successors of doneList1 in P1;
36: update currList2 with successors of doneList2 in P2;
37: doneList1={}; doneList2={}; //reinitialize doneList1, doneList2
38: while (currList1 != null || currList2 != null); // end outer loop
39: 40: } adapter = And(adapter u blockAdapterSet);
41: //put the adapters for Blocks in Parallel with the current adapter
42: 43: return adapter;
44: 45: }
```

Figure 2-9. SP adapter creation algorithm (SPA)
2.6 A heuristic algorithm for a minimal adapter (PBA-MIN)

Above we described algorithm PBA for creating an adapter. Next, we modify this algorithm and improve it further by raising the question, what is the smallest adapter we can create? Let us reconsider the example of Figure 2-10. Adapter 2 is an improvement upon Adapter 1 because it only buffers ship info. After ship info is received by Adapter 2, the retailer process can receive the invoice from the manufacturer process. Then, the two processes transact payment. In the last step, the Adapter2 sends ship info to the manufacturer process. Thus, Adapter2 has only two activities in a sequence pattern, and is simpler than Adapter1.

In order to get a minimal adapter, intuitively we should buffer as few messages as possible. Consequently, in each iteration of the SP adapter algorithm (Figure 2-9), if both processes are stuck, we can buffer only one send activity. After this send activity is buffered, if both processes are still "stuck" (i.e. they cannot proceed), we would buffer another send activity in the next iteration, and so on, until the processes can proceed.

To implement the above idea, we need to choose the "best" send activity to buffer, i.e. one which has the highest probability to resolve the block. First, we define the distance between
two activities simply as the number of arcs or steps in a shortest path connecting them (it is infinity if there is no path). In Figure 2-10 for example, we observe that when both processes are stuck, i.e. the retailer process stops at *send ship info* and the manufacture process stops at *send invoice*, the distance between *send ship info* and *receive invoice* (the dual of *send invoice*) is 1, while the distance between *send invoice* and *receive ship info* (the dual of *send ship info*) is 2.

Intuitively, if we buffer *ship info*, it only takes one step to free the "stuck" status of the manufacturer process, while it takes two steps to free the stuck status of the retailer process if we buffer the *invoice*. Therefore, we buffer the *ship info* first. To formalize this idea, we define the *minimum receive distance* next.

**Definition 2-3:** The **minimum receive distance** (MRD) of a *send* activity in a current list of a process is the minimum of the distances between this activity and each of the *receive* activities, which are the duals of the *send* activity/activities in the current list of the other process, i.e.,

\[
\text{MRD}(a_1) = \min(\text{Dist}(a_1, \text{Dual}(a_2))),
\]

where \((a_1 \in \text{currList}_1 \&\& a_1.\text{type}=='\text{send}' \&\& a_2 \in \text{currList}_2 \&\& a_2.\text{type}=='\text{send}') \)

\[\| (a_1 \in \text{currList}_2 \&\& a_1.\text{type}=='\text{send}' \&\& a_2 \in \text{currList}_1 \&\& a_2.\text{type}=='\text{send}').\]

The smaller the MRD of a *send* activity, the larger is the probability that the buffering of this activity can free the processes. Therefore, a reasonable heuristic (**Heuristic 1**) is to buffer the *send* activity \((a_1)\) with the smallest MRD when both processes are stuck and both current lists contain *send* activities, i.e.

\[
\text{selectedSend} = a_1, \text{s.t. } \text{MRD}(a_1) = \min(\text{MRD}(a)).
\]

In the example of Figure 2-10, both current lists contain at least one *send* activity. However, in some other cases, when two processes are stuck, it could also be that only one
current list (say, \textit{currList1}) contains all \textit{send} activities, while the other current list (say, \textit{currList2}) may contain only \textit{receive} activities. In such situations, to buffer the \textit{send} activity closer to the \textit{send} activities which are the duals of the \textit{receive} activities in \textit{currList2} is more likely to free the stuck processes. To formalize this idea, we define the \textit{minimum send distance} next.

\textbf{Definition 2-4}: The \textit{minimum send distance} (MSD) of a \textit{send} activity in the current list of a process is the minimum of the distances between this activity and each of the activities which are the duals of the \textit{receive} activities in the current list of the other process, i.e.,

\begin{align*}
\text{MSD}(a1) = \text{Min}(\text{Dist}(a1, \text{Dual}(a2)))
\end{align*}

where \((a1 \text{ in } \text{currList1} \&\& a1.\text{type}=='\text{send}' \&\& a2 \text{ in currList2} \&\& a2.\text{type}=='\text{Recv}' ) || (a1 \text{ in } \text{currList2} \&\& a1.\text{type}=='\text{send}' \&\& a2 \text{ in currList1} \&\& a2.\text{type}=='\text{Recv}')\). ■

The smaller the MSD of a \textit{send} activity is the greater is the probability that buffering this activity can free the stuck processes. Therefore, a second heuristic (\textbf{Heuristic 2}) is to buffer the \textit{send} activity (a1) with the smallest MSD when both processes are stuck and only one current list contains \textit{send} activities, i.e.

\text{selectedSend} = a1, \text{s.t. MSD}(a1) = \text{Min}[\text{MSD}(a)].

Our algorithm is shown in Figure 2-11. For brevity, we only list the treatment for the \textit{send} activities in both current lists, where the major difference from the SP adapter algorithm in Figure 2-9 lies. The algorithm maintains a variable \textit{isStuck} to indicate the status of the two processes. Any time both processes are stuck, the SP algorithm sets \textit{isStuck} to True. Then the above two heuristics are applied depending upon whether both current lists contain \textit{send} activities (Heuristic 1), or only one contains \textit{send} and the other only \textit{receive} activities (Heuristic 2). In the two cases the \textit{send} activity with the minimum MRD or the minimum MSD, respectively, is selected for adding to the adapter, and the variable \textit{isStuck} is set to False.
There is a subtle semantic difference between the adapters created by PBA and PBA-MIN algorithms. The adapter created by PBA-MIN allows a \textit{send} to get blocked if the corresponding \textit{receive} of the other party is not ready, while ensuring there is no deadlock. For example, in Figure 2-10 while using Adapter 2 the manufacturer may get blocked while trying '\textit{send Invoice}' if the retailer has not performed the '\textit{send Ship Info}' activity first. However, this issue does not arise in Adapter 1 created by PBA because the '\textit{send invoice}' activity is also buffered in the adapter and so this \textit{send} will not block. This is the cost of creating a smaller adapter.

CreateMinSPAdapter(Proc P1, Proc P2, array blockAdapter[][]) 
// Minimal SP adapter creation algorithm (SPA-MIN) 
01: if (isStuck) 
02: { 
03: sendList1 = getSendActivity(currList1); 
04: sendList2 = getSendActivity(currList2); 
05: if ((sendList1 is empty)||(sendList2 is empty)) 
06: //if only one currList contains \textit{send} activity 
07: {Calculate MSD for each \textit{send} activity in non-empty list; 
08: Select the \textit{send} activity \textit{a} with the smallest MSD; 
09: } else //if both currLists have \textit{send} activities 
10: {Calculate MRD for each \textit{send} activity in the sendLists; 
11: Select the \textit{send} activity \textit{a} with the smallest MRD; 
12: } 
13: add \textit{a} to pool; 
14: remove \textit{a} from currList1 or currList2; 
15: add \textit{a} to doneList; 
16: add dual(\textit{a}) to adaptActivitySet; 
17: isStuck = FALSE; 
18: }

Figure 2-11. Minimal SP adapter creation algorithm (SPA-MIN)

2.7 Summary

In this chapter, we viewed business processes in terms of standard patterns, and described a pattern compatibility matrix and rules that allow us to simplify the task of checking compatibility between two or more processes because these prerequisite rules can be applied to each pattern separately, thus reducing the search space. We gave a verification algorithm for
applying these rules to check process compatibility. If two processes are compatible, we
determine whether an adapter is required, and if so, a minimal adapter is generated by another
algorithm. Two variants of the adapter creation algorithm (PBA and PBA-MIN) are presented,
both of which apply a simulation-based method. We discuss implementation issues and evaluate
the performance of these adapter creation algorithms in Chapter 8. Next, we look at a different
approach for adapter creation based on pair wise analysis of messages.
Chapter 3

Pair-wise Analysis for SP Adapter Creation

As observed in the motivating example in Figure 1-1, the incompatibilities are mainly due to crossovers in communication flows. When two inter-process communication flows cross each other, an incompatibility can arise. In this chapter, we study the adapter creation for message pair. Then in Chapter 4 and Chapter 5, we discuss how to extend them to the full SP processes in synchronous and asynchronous communication. First, we discuss the differences among synchronous and asynchronous communication models.

3.1 Communication Modes in Process Composition

As discussed in Section 1.2, the adapters for multiple given processes in synchronous and asynchronous communication may be different. In this section, we discuss the classification of communication modes.

In the **synchronous** mode, when a client invokes a service it "blocks" until a reply is received from the service. Thus, if two peer processes are both performing invoke actions on each other, this causes the processes to get stuck since there is no buffer to store a message and a send must be immediately received. This gives rise to a need for an adapter as we saw in the example of Figure 1-1 (b).

In the **asynchronous** mode of invocation, a message received by a server is placed in a queue until explicitly requested by the *receive* activity in the server process. A WSBPEL [OASIS WSBPEL TC 2007] server like Oracle BPEL Process Manager [Oracle 2008] maintains *queues* for the application server to (i) queue the received message; (ii) send the acknowledge message to the sender if necessary; and (iii) deliver the message to the *receive* activity when it is ready. Since
an organization generally operates a number of different business processes at a same time, one can assume multiple FIFO queues, one for each process (see Figure 3-1 (b)), in charge of the message communication with other parties [Engels et al. 2002]. For a specific process, all received messages are saved in a same FIFO queue, and therefore should be retrieved by the process in a same order as they were received. Further, the queues are assumed to have an unbounded capacity. (The synchronous mode is shown in Figure 3-1 (a) for contrast.)

![Figure 3-1. Synchronous and asynchronous modes of communication between processes](image-url)

Based on this clarification of communication models, in next section we analyze how they impact the interaction between message pair.

### 3.2 Pair-wise Interaction

We focus on SP processes because after two processes are matched in terms of corresponding loop and choice structures, only S and P structures remain. In a SP process, the relation between any two activities can be < (before), > (after) or || (parallel). Therefore, given any two messages $m_i$ and $m_j$ in a S, P or SP pattern, there are 10 possible cases for their allocation and communication. The six cases that are compatible without an adapter for both synchronous
and asynchronous communication are shown in Figure 3-2, and the three incompatible ones that need adaptation and a deadlocked one in Figure 3-3. We also distinguish the adaptation needed in synchronous and asynchronous communication. In the simplified notation in Figure 3-2 and Figure 3-3, $S_i$ and $R_i$ denote send and receive activities, respectively, for message $m_i$.

In synchronous communication, the incompatibility in case 7 is due to the two send messages that cross each other in communication flows. Therefore, we call it a Crossing Sequential Sends (or Type A) incompatibility. For Type A incompatibility we need to adapt message $m_1$ from process $P$, and then send it to process $Q$, after the $S_2$-$R_2$ message transfer. The incompatibility in case 8 is due to the two independent send activities that cross each other in communication flows. Hence, we call it Crossing Independent Sends (or Type B) incompatibility. For Type B incompatibility, we can adapt either $m_1$ or $m_2$. Either of them can unlock the interaction.

Figure 3-2. Compatible cases of a message pair interaction for both types of communications
In asynchronous communication, the incompatibility in case 7 occurs because the two send activities are in a different order from that of the two receive activities. Therefore, we call it Misordered Sequential Sends (or Type C). For Type C incompatibility, we need to adapt $m_1$ and $m_2$ from process $P$ and then send it to process $Q$ in a reversed order. The incompatibility in case 9 is due to the two independent send activities that can happen in any order. Hence, we call it
Any ordered Parallel Sends (or Type D) incompatibility. For Type D incompatibility, we can adapt both messages from $P$ in parallel, and send them to $Q$ in a right order.

### 3.3 Detection of Incompatible Cases

In this section, we discuss the rules to detect the incompatible cases described in Figure 3-3. We consider **Inter-process Message Direction (MD)** and **Intra-process Message Order (MO)**. MD shows the communication direction between processes, say, $P$ and $Q$, e.g. "$P \rightarrow Q$; $P \rightarrow Q"$, "$P \rightarrow Q; Q \rightarrow P".".

MO shows the partial order between two send/receive activities related to this message pair in that process. The relationship could be $<$ (before), $>$ (after), or $\parallel$ (parallel). For a message pair $(m_i, m_j)$ in process $P$ or $Q$, $m_i < m_j$ means the activity involving $m_i$ (say, $S_i$ or $R_i$) executes before the activity involving message $m_j$ ($S_j$ or $R_j$). Next, we design a set of rules based on the MD and MO values to detect each incompatible case, as shown in Table 3-1 (for Type A and B in synchronous communication) and Table 3-2 (for Type C and D in asynchronous communication). For each Type A incompatibility, we add an attribute to record which message should be adapted, i.e. $A(m_i)$. For a Type B incompatibility, we add a set of attributes to record both messages any one of which may be adapted as $(B(m_i, m_j))$. In asynchronous communication, if the MD is "$P \rightarrow Q; Q \rightarrow P"$, then there is no incompatibility. If the MD is "$P \rightarrow Q; P \rightarrow Q"$, then it is either a Type C or D incompatibility depending upon the MO information as shown in Table 3-2.
Table 3-1. Rules for detecting Type A and B incompatibilities for synchronous communication (2 messages \(m_i, m_j\); 2 processes P & Q)

<table>
<thead>
<tr>
<th>Message Order</th>
<th>Message Direction</th>
<th>(m_i : P \rightarrow Q)</th>
<th>(m_j : P \rightarrow Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MO_P(m_i, m_j) = MO_Q(m_i, m_j))</td>
<td>OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>(MO_P(m_i, m_j) = &quot;</td>
<td></td>
<td>&quot; or (MO_Q(m_i, m_j) = &quot;</td>
<td></td>
</tr>
<tr>
<td>((MO_P(m_i, m_j) = &quot;&lt;&quot; and (MO_Q(m_i, m_j) = &quot;&gt;&quot;) or (MO_P(m_i, m_j) = &quot;&gt;&quot; and (MO_Q(m_i, m_j) = &quot;&lt;&quot;))</td>
<td>If (S_i &gt; S_j) Then Type A((m_i)) Else Type A((m_j))</td>
<td>Type B(({m_i, m_j}))</td>
<td></td>
</tr>
</tbody>
</table>

Table 3-2. Rules for detecting Type C and D incompatibilities for asynchronous communication (2 messages \(m_i, m_j\); 2 processes P & Q)

<table>
<thead>
<tr>
<th>Message Order</th>
<th>Message Direction</th>
<th>(m_i : P \rightarrow Q)</th>
<th>(m_j : P \rightarrow Q)</th>
<th>(m_j : Q \rightarrow P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((S_i &lt; S_j) \text{ and } (R_i &lt; R_j)) or ((S_i</td>
<td></td>
<td>S_j) \text{ and } (R_i</td>
<td></td>
<td>R_j)) or ((S_i &gt; S_j) \text{ and } (R_i &gt; R_j))</td>
</tr>
<tr>
<td>(R_i | R_j)</td>
<td>OK</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((S_i &lt; S_j) \text{ and } (R_i &gt; R_j)) or ((S_i &gt; S_j) \text{ and } (R_i &lt; R_j))</td>
<td>If (R_i &lt; R_j) Then Type C((m_i, m_j)) Else Type C((m_j, m_i))</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((S_i | S_j) \text{ and } (R_i &lt; R_j) \text{ or } (R_i &gt; R_j))</td>
<td>If (R_i &lt; R_j) Then Type D((m_i, m_j)) Else Type D((m_j, m_i))</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.4 Summary

In this chapter, we proposed a new adapter creation algorithm based on message pair analysis. We decompose the interaction of two processes to the interaction of each message pair. By exhaustively checking all the communication scenarios for one message pair, we identified incompatibility cases for both synchronous and asynchronous communications. Then, we gave the rules to detect the incompatible cases described. Intuitively, we can design an algorithm for applying these findings on each pair of messages in a process composition. Then, for each detected incompatible case, we can generate an adaptation element as described in Figure 3-3.
Last, we can combine these adaptation elements together to generate a complete adapter. However, these rules introduced in Table 3-1 and Table 3-2 may not remain true when the case involves more messages and processes. Therefore, in the next chapter we extend this analysis to the general cases of $M$ processes and $N$ messages ($M, N > 2$).
Chapter 4

General Case of Synchronous Communication

Chapter 3 focuses on the analysis of two messages and two processes. Now, we examine how our approach can be generalized to the general case of \( M \) messages and \( N \) processes \((M, N > 2)\). In this chapter, we first discuss the situations in synchronous communication. Then, in the next chapter we present the details in asynchronous communication.

### 4.1 Incompatible Cases in 3-process Interaction

Consider the communication of \( m_1 \) and \( m_3 \) in Figure 4-1. If we omit message \( m_2 \), it is easy to see that \( m_1 \) and \( m_3 \) are compatible in all four cases shown in this figure by applying the rules in Table 3-1. However, this analysis is not correct if we bring \( m_2 \) back into the picture and assume that \( S_2 \) and \( R_2 \) are executed synchronously. Combining the causality \( R_3 < R_2 \) (i.e. \( R_3 \) before \( R_2 \)) in process \( R \) with the causality \( S_2 < R_1 \) in process \( Q \), leads to the causality \( R_3 < R_1 \) by transitivity. This is shown in Figure 4-2 (a). We call this a Type A* incompatibility because it is similar to a Type A incompatibility. The only difference is that the crossing of sequential sends arises from an inter-process causal dependency across processes. Similar inter-process causal dependency between a pair of messages and their corresponding incompatibilities can be inferred in Figure 4-1 (b), (c) and (d) by identifying the transitivity created by other messages. In Figure 4-1 (b), there is a \( S_3 < R_1 \) causality leading to an incompatibility between \( m_1 \) and \( m_3 \), similar to a Type B incompatibility, which we call Type B*. In this new incompatibility, the crossing of independent sends arises from an inter-process causal dependency across processes. Finally, in Figure 4-1 (c), there is a \( S_1 < S_3 \) causality, and in Figure 4-1 (d), a \( S_1 < S_3 \) causality, both causing Type A* incompatibilities between \( m_1 \) and \( m_3 \). Thus, two activities have an inter-process
causal dependency relationship in synchronous communication if there is a control flow path between them, treating the communication flow from the send activity of a message to its receive activity as a bi-directional link (Figure 4-2 (a)).

Figure 4-1. Incompatible cases involving 3 messages and 3 processes

Figure 4-2. Illustration of inter-process causal dependency in the two types of communication

Hence, in general, it is possible to apply the pair-wise analysis of Table 3-1 to each pair of messages after considering such inter-process causal dependency, and determine the adaptations needed for each pair. Finally the receive and send activities for each message to be adapted are included in a process to realize the final adapter. As an example, in Figure 4-1 (b), we need to consider three pairs of messages, \( \{m_1, m_2\}, \{m_1, m_3\} \) and \( \{m_2, m_3\} \) in turn. In each case there is a Type B* incompatibility. Thus, the necessary adaptations are \( B(m_1, m_2) \), \( B(m_1, m_3) \), and \( B(m_2, m_3) \). By adapting any two of the three messages, say \( m_1 \) and \( m_2 \), we can create a correct adapter.
However, such an adapter is suboptimal. For instance, in this example, an optimal adapter would adapt only one message \((m_1, m_2, \text{ or } m_3)\), not two. This is because in synchronous communication the inter-process causal dependency may change when some intermediate messages are adapted. Therefore, in order to create an optimal adapter, we must consider all message communication dependencies together, instead of considering one message pair at a time. In next section, we propose to use message interaction graphs (MIGs) to solve this problem.

### 4.2 The MIG Approach to Create a Minimal Adapter

In this section, we introduce a new approach based on message interaction graphs to create a minimal adapter in synchronous communication. First, we give the definition of a message interaction graphs and show how to use it for incompatibility detection in general case. Then, we discuss the idea of the Message Interaction Graph (MIG) analysis for adapter creation. Last, we give the adapter creation algorithm based on MIG analysis, with two options in the adaptation element selection procedure, i.e. a greedy heuristic based approach and an optimal approach based on the Integer Programming technique.

#### 4.2.1 Message interaction graph (MIG) and the general case

In this section, we first give a short summary for the incompatible cases for 2 and 3 processes in synchronous communication.

Figure 4-3, abbreviated from Figure 3-3, summarizes the incompatible cases in a 2-process scenario. Case 1 in Figure 4-3 is deadlocked and no adaptation is possible. Intuitively, this is because both processes are waiting to receive a message from the other one. We call it a Type I(0)
incompatibility, i.e. an incompatibility with 0 initial send messages. In case 2, there are one send and one receive message in the initial set of messages of the two processes. This is called Type I(1) incompatibility because there is one send in the initial set of messages. Similarly, in Type I(2) incompatibility (case 3 in Figure 4-3) there are two send messages in the initial set of messages, and in general, in a Type I(j) incompatibility among n processes there are j send and n-j receive messages in the initial set of messages.

<table>
<thead>
<tr>
<th>Incompatibility</th>
<th>Type I(0) - Deadlocked</th>
<th>Type I(1)</th>
<th>Type I(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2  R1</td>
<td>S1  S2</td>
<td>S1  R2</td>
<td>S1  R2</td>
</tr>
<tr>
<td>(10)</td>
<td>(8)</td>
<td>(9)</td>
<td></td>
</tr>
<tr>
<td>Adaptation</td>
<td>Deadlock cannot be</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Element</td>
<td>resolved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Required</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial set</td>
<td>(S2, R1)</td>
<td>(S1, R2)</td>
<td>(S1, S2)</td>
</tr>
<tr>
<td>Number of</td>
<td>0 sends</td>
<td>1 send</td>
<td>2 sends</td>
</tr>
</tbody>
</table>

Figure 4-3. Incompatible cases involving 2 messages and 2 processes in synchronous communication

The examples in Figure 4-4 involve three messages in a 3-process scenario. In Figure 4-4 (a), each send is waiting for a corresponding receive message before it can proceed itself, thus causing a deadlock. This is a Type I(0) incompatibility because there are 0 send in the initial set of possible messages {R1, R2, R3}. In Figure 4-4 (b), we observe that the communication of message 1 interferes with that of messages 2 and 3. Therefore, process 1 cannot send message 1, because process 3 is waiting for message 3 from process 2, and process 2 is waiting for message 2 from process 1. This is a Type I(1) incompatibility because there is one send in the initial set of messages {S1, R2, R3}. Similarly, Type I(2) and I(3) cases are shown in Figure 4-4 (c) and (d).
respectively. This type of dependency structure among messages is called a Crown in the study of message communication in a distributed system [Charbon-Bost et al. 1996].

![Diagram of Crown dependencies among messages in a distributed system](image)

**Figure 4-4.** Incompatibility involving 3 messages and 3 processes

Notice also, that the basic idea of adapter creation is similar in the 2 and 3 process cases, and can be generalized to n processes. In the n process case for an l(j) incompatibility, we wish to adapt at least one of the j send messages in the initial set of messages to resolve the incompatibility. This adaptation is achieved by creating a new adapter process that receives the message and then sends it at a later time. Thus, for every message to be adapted, a (receive, send) pair of steps is added to the adapter. More specific details on the adaptation will follow shortly.

In order to understand and analyze the incompatibilities discussed above, we propose a new graph named the Message Interaction Graph that captures the dependencies among messages. It is defined first and then examples follow.

**Definition 4-1:** A Message Interaction Graph (MIG) consists of nodes and directed arcs, where:
(1) each node (labeled with a message ID) denotes a unique message in a multiple-process interaction.

(2) each directed arc from node \(i\) to node \(j\) induces a dependency between the message pair \((i,j)\) of one of four types as follows:

- **SS\((i,j)\)**: message \(i\) must be sent before message \(j\) is sent
- **SR\((i,j)\)**: message \(i\) must be sent before message \(j\) is received
- **RS\((i,j)\)**: message \(i\) must be received before message \(j\) is sent
- **RR\((i,j)\)**: message \(i\) must be received before message \(j\) is received.

A MIG graph is constructed by observing the time line of a process (where time runs from top to bottom). The MIG graphs for Figure 4-3 and Figure 4-4 are shown in Figure 4-5. The Type I(0) incompatibly in Figure 4-3 arises from two RS dependencies between messages 1 and 2. The Type I(1) incompatibly in Figure 4-3 results from the flows S1 \(\rightarrow\) S2 in process 1, and R2 \(\rightarrow\) R1 in process 2. Consequently, in its MIG there is an arc from message 1 to message 2 labeled SS (send-send), and an arc from message 2 to message 1 labeled RR (receive-receive), as shown in Figure 4-5 (b). Similarly, Figure 4-5 (e) arises from the SS, RS and RR dependencies in Figure 4-4 (b).

![MIG graphs](image)

Figure 4-5. MIGs for the incompatible cases in Figure 4-3 and Figure 4-4

The algorithm for MIG creation with multiple processes is given next.
Procedure 4-1: (MIG construction)

1) Select a process $p_i$ from the process set $P$.

2) For each message $j$ ($m_j$) sent or received by $p_i$, if there is no node for $m_j$ in MIG, create a node $n_j$ for $m_j$.

3) For each control-flow link $Xa \rightarrow Yb$ in $p_i$, where $X, Y$ are S or R, create an arc from node $n_a$ to node $n_b$ labeled $XY$.

4) Repeat Step 1-3 until no process is left in $P$. ■

In Figure 4-5, we observe that each MIG contains a cycle. Each cycle implies that there is a circular dependency among the messages in it. Hence it is not possible to complete this execution without the use of an adapter. Moreover, in some cases, e. g. Figure 4-5 (a) and (d), incompatibilities cause a deadlock which cannot be resolved by an adapter. We first treat the case of a deadlock.

**Result 4-1:** In a MIG, if all the links in a cycle are of type RS or RR, then the process interaction represented by the MIG is deadlocked, and it cannot be broken with an adapter.

**Proof Sketch:** We can represent this cycle of length $c$ as: $m_1^{R^*} \rightarrow m_2^{R^*} \rightarrow \ldots \rightarrow m_c^{R^*} \rightarrow m_1^{R^*}$, where $R^*$ represents RR or RS. In this cycle, every process is waiting to receive a message from another one. Since there is no process that can send a message in this cycle, there is no way to break it. Hence, this cycle is deadlocked. ■

Next we characterize conditions under which an adapter is needed by Result 4-2.

**Result 4-2:** Multiple interacting processes are compatible without an adapter if and only if their MIG does not contain any cycle.

**Proof Sketch:** (⇒) To prove by contradiction, let us suppose the MIG of multiple processes contains a cycle of length $c$, which is $m_i \rightarrow m_2 \rightarrow \ldots \rightarrow m_c \rightarrow m_i$. The cycle is equivalent to a deadlock in the flow of control among these messages, where message $m_i$ can be completed only if message $m_{j,i}$ is completed, and so on. Thus, message $m_i$ depends upon itself.
Hence, these multiple processes are not compatible. Hence, the first part of the proof follows by contradiction.

(←) If the MIG of multiple processes does not contain any cycle, then all paths in the MIG are of the form \( m_1 \rightarrow m_2 \rightarrow \ldots \rightarrow m_i \rightarrow m_c \) where no node is repeated. We can also rewrite such a path, by replacing each message \( m_i \) by its send and receive parts. Thus, we get \( S_1 \rightarrow R_1 \rightarrow S_2 \rightarrow R_2 \rightarrow \ldots \rightarrow R_i \rightarrow S_i \rightarrow \ldots \rightarrow R_c \rightarrow S_c \). Since each message is received immediately after it is sent, there is no need for an adapter. Thus, we argue that Result 4-2 is valid.

Based on this result, we know that to detect incompatible cases in interacting processes, we just need to detect cycles in the MIG of these processes. Next, we discuss how to resolve these incompatibilities based on the detected cycles in MIG.

Note that though a MIG looks similar to a directed acyclic graph (DAG), they are different constructs. In a MIG, each arc (edge) contains control flow information such as SS, SR, etc. In contrast, the edges in a DAG contain no such information, and just show the dependency relationship between two nodes. Moreover, a DAG does not allow cycles, but MIG does. In fact, our algorithm utilizes the cycle information in MIGs to identify the adaptable messages. In summary, MIG can be regarded as a special extension of DAG, which just provides a basic graph notation. However, they have different representations. Therefore, we use the new term MIG to differentiate it from DAG and to avoid unnecessary confusion.

4.2.2 Using MIG for adapter creation

In previous section, we showed that adaptation elements can resolve the incompatibilities that arise from cycles in the MIG, except for the cases where a cycle consists only of RS or RR arcs. So the next question we pose is, how can we identify which message(s) to adapt based on the MIG? From the observations in Figure 4-3 and Figure 4-4, we know that only one of the
initial send activities needs to be adapted to break a cycle containing the SS, SR, RR, RS arcs. Here we formalize this idea and give an algorithm for creating an adapter. To identify the message for these starting send activities in MIG, we give the following result.

Result 4-3: In a MIG cycle, a message $i$ can be adapted to break the cycle if its outgoing arc is labeled S* (SR or SS), and its incoming arc is labeled *R (SR or RR).

Proof Sketch: The outgoing SS or SR arc for message $i$ means that there is a control-flow link from $S_i$ to $S_j$ or $R_j$ within one process, say $p$. So there must be some (send or receive) activity following $S_i$ in Proc $p$. The incoming RR or SR arc for message $i$ means that there is a control-flow link from $S_k$ or $R_k$ to $R_i$ in another process, say $q$. Therefore, $S_i$ is a starting or initial activity of $p$ in the cycle. Thus, let us assume without loss of generality that the initial cycle was: $m_1 \rightarrow m_2 \rightarrow \ldots \rightarrow m_c \rightarrow m_1$. Then after adapting message $m_1$ (say, it has an S* out arc and *R in-arc), the new graph will have the following links $m_1 \rightarrow m_2 \rightarrow \ldots \rightarrow m_c \rightarrow m_{adapt}$, and $m_1 \rightarrow m_{adapt}$ (see Figure 4-5 (b) for an example). Hence the cycle is broken by adapting $m_1$. ■

From Result 4-1 and Result 4-2, we know that to check whether multiple processes are compatible in an interaction, we transform them into a MIG and check for cycles in it. If there is no cycle in the MIG, then these processes are compatible without an adapter. If at least one cycle exists in which no send is adaptable (Type I(0) incompatibility), these interacting processes are deadlocked. If all cycles in the MIG contain at least one adaptable send (from Result 4-3), then we can always create an adapter to make these processes compatible in interaction. Based on this idea, a naive adapter creation algorithm CreateSPAdapterNaive is given in Figure 4-6. In this algorithm, we arbitrarily identify one adaptable send for each cycle in the MIG graph, and add it to the adaptation element set AES. Then each element in this set is added in parallel to the adapter process.
CreateSPAdapter_Naive(Proc[] ProcList) //MIG-Naïve algorithm

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>identify all cycles in the MIG graph;</td>
</tr>
<tr>
<td>02</td>
<td>if (a cycle c* has all RS or RR arcs)</td>
</tr>
<tr>
<td>03</td>
<td>{ Print(&quot;deadlock cycle c*”); exit }</td>
</tr>
<tr>
<td>04</td>
<td>AES = {}; // AES = adaptation element set</td>
</tr>
<tr>
<td>05</td>
<td>foreach cycle c</td>
</tr>
<tr>
<td>06</td>
<td>{ e = select an &quot;adaptable send message&quot; in c; // see Result 5.3</td>
</tr>
<tr>
<td>07</td>
<td>AES = AES + e;</td>
</tr>
<tr>
<td>08</td>
<td>}</td>
</tr>
<tr>
<td>09</td>
<td>Adapter = {};</td>
</tr>
<tr>
<td>10</td>
<td>foreach (e in AES)</td>
</tr>
<tr>
<td>11</td>
<td>{</td>
</tr>
<tr>
<td>12</td>
<td>Adapter = And(Adapter, Seq(receive(e), Send(e)));</td>
</tr>
<tr>
<td>13</td>
<td>delete e from AES; update AES;</td>
</tr>
<tr>
<td>14</td>
<td>}</td>
</tr>
<tr>
<td>15</td>
<td>return Adapter;</td>
</tr>
</tbody>
</table>

Figure 4-6. A naïve algorithm for creating an adapter based on a MIG (MIG-Naïve)

In the next section we give a detailed example and focus on how to create an optimal adapter by selecting the minimal set of adaptation elements in a smarter way.

4.2.3 Optimal adapter creation

In this section, we apply the results in Section 4.2.1 and 4.2.2 to design an adapter creation algorithm that breaks the cycles in a MIG more selectively. This algorithm consists of three steps. It first identifies all directed cycles in the MIG. Then, it selects a minimal set of adaptable elements from each cycle in two ways: using either a greedy algorithm or the Integer Programming technique. Last, it integrates the selected adaptation elements to a complete adapter process. We illustrate this algorithm using the example in Figure 4-7.

4.2.3.1 Detection of incompatible cases

As discussed above, incompatible cases can be resolved by breaking directed cycles in a MIG. We found the Johnson’s algorithm [Johnson 1975], an efficient algorithm for enumerating
all circuits in a directed graph, in the literature. After finding each cycle using this algorithm, we identify the adaptable sends based on the rules in Result 4-3.

Figure 4-7. An illustrative example of three interacting processes

Johnson’s algorithm searches for circuits in an appropriate search space which is a superset containing all circuits. It uses backtracking to generate dipaths of the graphs, and then identifies if it is a dicircuit. This algorithm has an upper bound of $O((n + e)c)$ on time, for $n$ vertices, $e$ arcs and $c$ elementary circuits in the graph. It is the asymptotically fastest algorithm so far. The details of the Johnson's algorithm are introduced in Appendix A. Next we consider an example.

**Example 4-1.** Consider the example in Figure 4-7. We generate the MIG as shown in Figure 4-8 (a). Using the Johnson’s algorithm, we get 6 elementary cycles (shown in Figure 4-8 (b)). We can adapt:

- message 1 or 9 in cycle 1,
- message 3 or 9 in cycle 2,
- message 4, 5 or 8 in cycle 3,
• message 3 or 4 in cycle 4,
• message 3, 4, 5, 8 or 9 in cycle 5,
• message 3, 4, 5 or 9 in cycle 6.

Figure 4-8. MIG and detected cycles for the example in Figure 4-7

4.2.3.2 Selection of adaptation elements

In the example of Figure 4-7, after identifying all the incompatible cases, we can generate the adapter elements for each of them, and then combine the elements. For the Type I(1) incompatibilities, there is only one option for an adaptable message, so it must be adapted. As for the Type I(j) (j > 1) incompatibilities, there is a choice as per Result 4-3. We use a greedy algorithm to decide the minimum set that can cover all the remaining Type I(j) (j > 1) incompatibilities. The greedy approach always adapts the highest frequency message in descending order until all such incompatibilities are resolved. The algorithm is listed in Figure 4-9, and then explained for the example of Figure 4-7.
SelectAdaptationElement_Greedy(Proc P,Proc Q) // MIG-Greedy algorithm

01: let M={m1, m2,..., mn}, set of all messages;
02: put mi in each Type I(1) incompatibility into AS1; // Type I(1) set
03: put {mi, mj,...} in each Type I(j>1) incompatibility into AS2;
04: let adaptation element set AES = {};
05: foreach (mi in AS1)
06: { if (mi in AES)
07: { delete(mi,AS1)}
08: else {add(mi,AES); delete(mi,AS1);}
09: }
10: foreach {mi, mj,...} in AS2
11: { if (any of {mi, mj,...} in AES)
12: {delete({mi, mj,...},AS2);}
13: }
14: while (AS2 /≠{} ∅)
15: foreach (mi in M) {calculate frequency(mi) in AS2;}
16: mk = argmax,mi∈M(frequency(mi));
17: add(mk, AES);
18: delete any {mi, mj,...} with i=k or j=k;
19: return AES;

Figure 4-9. Greedy algorithm to select adaptation elements (MIG-Greedy)

Example 4-2: Let us continue with the example in Figure 4-7. There is no I(1) case here. Therefore, AS1={}. For all 6 cycles, we get AS2={ {m1, m9}, {m3, m9}, {m4, m5, m8}, {m3, m4, m5, m8}, {m3, m4, m9}, {m4, m5, m8, m9} }. In the 1st round, m3, m4, and m9 all have a frequency of 4. Here we just choose the one with the smallest ID, i.e. m3 and put it into AES. So we have AES={m3}. Accordingly, we delete all the tuples in AS2, which contain m3. Now, AS2={ {m4, m9}, {m4, m5, m9} }. Next, all five messages in AS2 only appear once each. So we choose m4 with the smallest ID. Then, AES={m4, m9} and AS2={ {m4, m5, m9} }. In the last round, we add m4 in AES to resolve the I(3) case left in AS2. Therefore, the final adaptation element set is AES={m4, m5, m9}.

The final adapter process is:

And(Seq(R(m1),S(m1))), Seq(R(m3), S(m3)), Seq(R(m4),S(m4))). ■

We combine the adaptation elements in parallel because this leads to the most flexible and general adapter. However, a “slimmer” adapter that sequences and rearranges these elements
may also be created based on the order in which the adapters are invoked. Techniques for creating a slimmer adapter are discussed in Appendix B.

4.2.3.3 An IP formulation for adaptation element selection

As noted above, while creating an adapter for Type I(j) (j > 1) incompatibilities, there is a choice of which message to adapt. This introduces non-determinism into adapter creation. In previous section, we illustrated a greedy method based on message frequencies. However, there is no guarantee this selection is optimal in terms of total number of messages. In this section, we introduce an Integer Programming (IP) formulation to select a true minimal set of adaptation elements.

The IP formulation in general consists of an objective function to be minimized, or maximized, subject to a set of constraints. The solution gives the optimal values of the variables in the objective function. Our IP formulation consists of an objective function to be minimized and a set of constraints for the three SP processes of Figure 4-7. The constraints encode the adaptation requirements to break all cycles. For each message, an IP variable is created. A solution to the IP formulation assigns a 0-1 integer value to each message \( m_i \) in the message set \( M = \{m_1, m_2, \ldots\} \). The messages assigned a 1 value need adaptation, and are present in the adapter, and the others are not. In the IP formulation below, we define \( m_i=1 \) or 0, to indicate whether the adapter contains message \( i \) or not.

**Definition 4-2.** For multiple processes, the basic IP formulation is:

\[
MIN(\sum_{i \in M} m_i)
\]

s.t.

IP0: For each \( m_i \) in \( M \), \( m_i = 0 \) or \( 1 \).
IP1: For each \( m_i \) in I(1), \( m_i = 1 \).

IP2: For each \( \{m_i, m_j\} \) in I(2), \( m_i + m_j \geq 1 \).

\[ \cdots \]

IPN: For each \( \{m_i, m_{i+1}, \ldots, m_{i+n-1}\} \) in I(n), \( m_i + m_{i+1} + \ldots + m_{i+n-1} \geq 1 \).

IP0 defines each \( m_i \) is binary. The constraint IP1 requires that every Type I(1)
incompatibility discovered in the MIG analysis should be adapted, while constraints IP2-IPN
require that for every Type I(j) (j > 1) compatibility one of the \( N \) messages in the pair should be
adapted. Thus, this IP formulation decides the minimum set of messages which should be
adapted.

The IP formulation for the example in Figure 4-7 is listed in Figure 4-10. It has 6
constraints, one for each cycle. In general, with more activities in each process there will be
additional constraints and a larger formulation. The MIG-IP algorithm identifies a minimal set of
adaptation elements, which includes messages 4 and 9, one fewer message compared with the
adaptation element set generated in the MIG-Greedy algorithm above. It shows that the MIG-IP
algorithm can generate a better adapter in terms of adapter size, compared with the MIG-Greedy
algorithm.

01: MIN: \( m_0 + m_1 + m_2 + m_3 + m_4 + m_5 + m_6 + m_7 + m_8 + m_9 \)

02: s.t.

03: \( m_1 + m_9 \geq 1 \);

04: \( m_3 + m_9 \geq 1 \);

05: \( m_4 + m_5 + m_8 \geq 1 \);

06: \( m_3 + m_4 \geq 1 \);

07: \( m_3 + m_4 + m_5 + m_8 + m_9 \geq 1 \);

08: \( m_3 + m_4 + m_5 + m_9 \geq 1 \);

Figure 4-10. An IP formulation for the example of Figure 4-7
4.2.3.4 IP model extensions

The IP formulation is flexible and lends itself to easy extensions. In this section, we discuss a few key types of constraints which can be added to extend our IP model.

1) Communication Cost: Different messages are of different size. Hence, the adaptation of different message sets may have different communication costs. The objective function can be modified to reflect the weighted sum of message sizes as follows. Thus, by optimizing for weighted message sizes, smaller messages will be more likely to be adapted.

\[
\text{MIN} \left( \sum_{i \in M} a_i m_i \right), \text{where } a_i \text{ is the weight of message } m_i.
\]

Example 4-3. Consider the example in Figure 4-7. Suppose \( m_1 \), \( m_3 \), and \( m_4 \) are simple information such as delivery address, \( m_5 \) and \( m_8 \) are business documents such as contract, and \( m_9 \) is a database file such as a customer list. Let size\( (m_1) = 201\text{KB}, \) size\( (m_3) = 503\text{KB}, \) size\( (m_4) = 127\text{KB}, \) size\( (m_5) = 2.1\text{MB}, \) size\( (m_8) = 4.3\text{MB} \) and size\( (m_9) = 124\text{MB} \). So the objective function of the adaptation message selection problem is updated to:

\[
\text{MIN} \left( 0.201 * m_1 + 0.503 * m_3 + 0.127 * m_4 + 2.1 * m_5 + 4.3 * m_8 + 124 * m_9 \right)
\]

Based on this new objective function, the MIG-IP algorithm generates a new result, which includes message \( m_1, m_3, \) and \( m_4 \). The total size of this message set is 831KB, which is much less than 124.127MB of the set \( \{m_4, m_9\} \). Please note that in this objective function, we ignore other messages \( m_6, m_2, m_7, \) and \( m_9 \), because none of them is an adaptable message in any detected cycle (as shown in Example 4-1). Therefore, their size information does not affect the selection of the minimum adaptation set.
2) **Security control:** Sometimes a user may desire that a message should not be adapter for security reasons, bearing in mind that an adapter process may be hosted by a third party. Thus, if a message \( i \) is not to be adapted, then the following additional constraint can simply be added, and a new solution is found.

\[ \text{IPnew1: } m_i = 0. \]

**Example 4-4.** Again, consider the example in Figure 4-7. Suppose \( m_9 \) (the customer list) includes the social security number of a customer. According to the contract with its customers, the company running Process 3 cannot release this sensitive information to any 3rd party that is not directly involved in this business transaction. Suppose the adaptation service is hosted in an e-market place, which does not have the full obligation of this business transaction. So, we need to add a new constraint \( m_9 = 0 \) to the IP model. Consequently, the MIG-IP algorithm generates a new result, which includes message \( m_1, m_3, \) and \( m_4 \).

3) **Privacy Preservation:** Sometimes, a set of messages may collectively contain enough personal information, such as gender, age, and zip code, etc. to identify an individual [Kumar et al. 2009]. Thus, we would not want to release the whole set of messages to a third-party adaptation service. Instead a limit can be imposed for the number of messages (\( n_{\text{max}} \)) that can be adapted from a given set of messages as follows:

\[ \text{IPnew2: } m_1 + m_2 + \ldots + m_n \leq n_{\text{max}}. \]

**Example 4-5.** Again, consider the example in Figure 4-7. Suppose \( m_1 \) is a data list, which includes the gender and age information of a number of customers. And, \( m_4 \) is another data list, which includes the postal code and region information of the same set of customers. As we know, the information in these two data lists forms a *quasi-identifier* [Kumar, et al. 2009], which can be used to fully or partially determine the
identity of those customers. Suppose the company running process 1 has the privacy agreement with its customers, which forbids it disclosing the customer identification to any untrusted party, e.g. the e-market place hosting the adaptation service. Therefore, we need to add an additional constraint $m_i + m_t \leq 1$ to the IP model, which prevents these two messages from being adapted together. So the set \( \{m_i, m_j, m_t\} \) is not longer an option. Consequently, the MIG-IP algorithm generates the result \( \{m_t, m_o\} \) as the optimal solution.

This partial list of examples shows how the IP model can be easily extended in a variety of ways to accommodate additional constraints in different scenarios. However, when additional constraints are added to the IP model, sometimes it may not be possible to get a solution satisfying all of them, in which case they may have to be modified. In Chapter 8, we design two experiments to evaluate the impact of those additional constraints on the adapter creation.

### 4.3 Summary

In this chapter, we investigated how the pair-wise approach can be generalized to general cases in synchronous communication. After introducing the impact of inter-process causal dependency on incompatibility detection, we proposed a new adapter creation algorithm based on message interaction graphs (MIGs). A message interaction graph captures dependencies among messages exchanged by processes. By analyzing this graph, we are able to show results for whether the processes are compatible (executable without an adapter), adaptable (executable with an adapter), or deadlocked for the general case of $N$ processes and $M$ messages ($M, N > 2$). Further, if they are adaptable we show how to create an optimal adapter of minimal size. The minimum set of adaptation elements required is selected by using an Integer Programming formulation and also compared with a greedy algorithm. Also, the IP model is flexible to include
other additional constraints for adapter creation. We discuss implementation issues and evaluate
the performance of these new algorithms in Chapter 8. Next, we examine how the pair-wise
approach can be generalized to asynchronous communication.
Chapter 5

Adapter Creation for Asynchronous Communication

For asynchronous communication, the extension from the 2-message, 2-process scenario is straightforward. Again consider the various cases of Figure 4-1. In case (a) and (b), \( R_1 \) and \( R_3 \) are in two different processes and hence they do not interfere with each other. Thus \( m_1 \) and \( m_3 \) are compatible. In case (c) and (d), \( m_1 \) and \( m_3 \) are not compatible because \( R_3 \) and \( R_1 \) are in a sequence in process \( R \), while \( S_1 \) and \( S_3 \) are in different processes. Thus, \( S_1 \) and \( S_3 \) are effectively sent in parallel, and \( R_3 \) and \( R_1 \) may be received in any order. If \( S_1 \) executes first, then an incompatibility is created. This is similar to a Type D incompatibility in Table 3-2, and we call it Type D*. In Figure 4-1 (b) there is no incompatibility. Thus, the general rule is:

Async Rule: if a pair of messages is received in a sequence, then they must have been sent in the same sequence as they are received to ensure that no adaptation is needed.

However, this analysis is not always optimal as indicated by introducing \( m_2 \) in Figure 4-2 (b). Combining the causality \( S_1 < S_2 \) (i.e. \( S_1 \) before \( S_2 \)) in process \( P \), the causality \( R_2 < S_3 \) in process \( Q \), and the implicit causality \( S_2 < R_2 \) in asynchronous communication leads to the causality \( S_1 < S_3 \) by transitivity. Therefore, this case is actually compatible! Similar inter-process causal dependency relationships in asynchronous communication between a pair of messages can be inferred by following the control-flow between them treating the communication flow as a one-way link from the send activity of a message to its receive. Compared with the synchronous case, here the communication link is only one-way (Si → Ri). For instance, in Figure 4-1 (c), there is a \( S_1 < S_3 \) causality leading to an incompatibility between \( m_1 \) and \( m_3 \) similar to a Type C incompatibility, which we call Type C*. This would be a Type D* incompatibility if we did not consider the inter-process causal dependency. Thus, the inter-process causal dependency...
relationship analysis in asynchronous communication leads to a more accurate determination of incompatibilities for multiple messages and processes, which generates a smaller adapter.

Besides the impact of the inter-process causal dependency relationship, the adaptations in different message pairs may affect each other. Consider an illustrative example in Figure 5-1 (a), in which two processes P and Q exchange three messages $m_1$, $m_2$ and $m_3$. Applying the rules of Table 3-2 (See page 36), we get $C(m_2, m_1)$ and $C(m_3, m_1)$. After merging the adaptation elements for both incompatibilities, we get Adapter 1 shown in Figure 5-1 (b). However, Adapter 2 in Figure 5-1 (c) is also correct, and smaller than Adapter 1. Figure 5-1 (d) shows the message communication enabled by Adapter 2. It receives $m_1$ from P. Then, P passes $m_2$ to Q directly, and sends $m_3$ to Adapter 2. Last, adapter 2 sends $m_3$ and $m_1$ to Q in the right sequence. Now $m_2$ and $m_3$ are in a same sequence in P and Q. In fact, when we use the adapter to reorder $m_1$ after $m_3$, it also guarantees that $m_1$ is also after $m_2$. Therefore, there is no need to adapt $m_2$ anymore. This transitive relationship is not captured by the previous pair-wise analysis.

Figure 5-1. An illustrative example for optimal adapter creation in async

To solve these challenges, we introduce an integer programming formulation to model the general asynchronous communication cases and show how it can produce the optimal
solution. The IP formulation is shown in Figure 5-2. It uses 0-1 integer variables $SS_{i,j}, SR_{i,j}, RR_{i,j}$ and $RS_{i,j}$ to denote a direct or transitive relationships between all task pairs $(i,j)$ ($i, j \in \text{message id}$) within the composing processes. In addition, 0-1 variable $XSS_{i,j}$ is introduced to denote the need for an adapter between message pair $(i,j)$ if its value is 1 in the solution. Similarly, 0-1 variable $YSS_{i,j}$ denotes the need to reverse the sequence relationship between a message pair $(i,j)$ if it is 1. Finally, 0-1 variable $TSS_{i,j}$ becomes 1 if there is a redundant adapter between message pair $(i,j)$.

<table>
<thead>
<tr>
<th>IP approach for optimal adaptation in async (Async-IP algorithm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP formulation:</td>
</tr>
<tr>
<td>$MIN \left( \sum_{i,j,k \text{ feasible}} (XSS_{i,j} + YSS_{i,j} + TSS_{i,j} + 2 \times SS_{i,j}) \right)$</td>
</tr>
<tr>
<td>s.t.</td>
</tr>
<tr>
<td>IP0: $SS_{i,j} = 1 \land SR_{i,k} = 1 \rightarrow SR_{i,k} = 1$ $\forall i, j, k$ // intra-process</td>
</tr>
<tr>
<td>IP1: $SS_{i,j} = 1 \land SS_{i,k} = 1 \rightarrow SS_{i,k} = 1$ $\forall i, j, k$ // transitivity</td>
</tr>
<tr>
<td>IP2: $SR_{i,j} = 1 \land SS_{i,k} = 1 \rightarrow SS_{i,k} = 1$ $\forall i, j, k$</td>
</tr>
<tr>
<td>IP3: $SR_{i,j} = 1 \land RR_{i,k} = 1 \rightarrow SR_{i,k} = 1$ $\forall i, j, k$</td>
</tr>
<tr>
<td>IP4: $RR_{i,j} = 1 \land SS_{i,k} = 1 \rightarrow RS_{i,k} = 1$ $\forall i, j, k$</td>
</tr>
<tr>
<td>IP5: $RS_{i,j} = 1 \land SR_{i,k} = 1 \rightarrow RR_{i,k} = 1$ $\forall i, j, k$</td>
</tr>
<tr>
<td>IP6: $RR_{i,j} = 1 \land RR_{i,k} = 1 \rightarrow RR_{i,k} = 1$ $\forall i, j, k$</td>
</tr>
<tr>
<td>IP7: $RR_{i,j} = 1 \land SS_{i,k} = 1 \rightarrow RS_{i,k} = 1$ $\forall i, j, k$</td>
</tr>
<tr>
<td>IP8: $RS_{i,j} = 1 \rightarrow SS_{i,j} = 1$ $\forall i, j$ // inter-process constraint</td>
</tr>
<tr>
<td>IP9: $RR_{i,j} = 1 \rightarrow SS_{i,j} + XSS_{i,j} = 1$ $\forall i, j$ // Async constraint 1</td>
</tr>
<tr>
<td>IP10: $RR_{i,j} = 1 \rightarrow SS_{i,j} + YSS_{i,j} = 1$ $\forall i, j$ // Async constraint 2</td>
</tr>
<tr>
<td>IP11: $RR_{i,k} = 1 \land SS_{i,k} = 0 \land SS_{i,j} + XSS_{i,j} = 1 \land SS_{i,k} + XSS_{i,j} = 1$ $\forall i, j, k$ // Async adaptation transitivity</td>
</tr>
</tbody>
</table>

Figure 5-2. IP formulation for optimal adaptation message selection in async (Async-IP)

The objective function minimizes the sum of all $XSS_{i,j}, YSS_{i,j}$ and $TSS_{i,j}$ variables to create a minimal adapter. It also includes the sum of all $SS_{i,j}$ variables weighted by 2 to prevent $SS_{i,j}$ values from falsely assuming a 1 value, i.e. just in order to minimize the objective function. IP0-IP7 are transitivity constraints to derive transitive associations between activities based on the
direct links among them. IP8 is an inter-process constraint to derive an inter-process SS link between two activities. IP9 captures the async rule described above by requiring that if there is a receive-receive sequence relationship between message pair \((i, j)\), then there must also be a send-send sequence relationship between \((i, j)\). If \(SS_{i,j}\) is not 1, then \(XSS_{i,j}\) is forced to be 1 to satisfy this constraint.

Similarly, IP10 requires that if there is a receive-receive sequence relationship between message pair \((i, j)\), and also a send-send sequence relationship between message pair \((j, i)\), then the latter must be reversed by forcing \(YSS_{i,j}\) to 1. Finally, IP11 identifies transitive relationship where if an adapter is required to adapt message pairs \((i, j)\) and \((j, k)\), then there is no need to adapt the message pair \((i,k)\) because it is redundant. In this case variable \(TSS_{i,k}\) is forced to 1 to neutralize the effect of \(XSS_{i,k}\) = 1. Thus if both \(XSS_{i,k}\) and \(TSS_{i,k}\) are 1 for a \((i, k)\) task pair it means that an adapter is not required for this pair of tasks. The constraints above enforce the rules for the creation of optimal adapter in asynchronous communication. In addition, we need to add the process definitions, i.e. the direct control links in the given processes, into our formulation as constraints. Finally, we specify that all variables \(SS_{i,j}, SR_{i,j}, RR_{i,j}\) and \(RS_{i,j}\) are binary.

As it shows, this IP formulation not only maintains all the control flow dependencies (both intra-process and inter-process), but also enforces all the asynchronous rules (both direct and transitive). Therefore, the adapter generated from this algorithm is guaranteed to be correct (no control flow or asynchronous constraint is broken) and optimal (no more adaptation can be saved).

**Result 5-1:** The Async-IP algorithm generates a correct and optimal adapter for a process composition in asynchronous communication.
**Proof Sketch:** To prove this result, we need to argue that: 1) the generated adapter satisfies the Async rule; and 2) the generated adapter does not have any redundancy arising from transitivity.

First, IP0-7 enforce all the intra-process transitivity and IP8 includes all the inter-process transitivity. IP9 enforces that whenever there is a sequential relationship between $R_i$ and $R_j$ ($RR_{i,j} == 1$), there must exist a sequential relationship between $S_i$ and $S_j$ ($SS_{i,j} == 1$). Otherwise, we have to adapt both of them and switch their order ($XSS_{i,j} = 1$). Furthermore, IP10 detects whether a sequential relationship exists between $S_j$ and $S_i$ ($SS_{j,i} == 1$). If so, this is a Type C incompatibility ($YSS_{i,j} = 1$). Otherwise, it is a Type D incompatibility ($YSS_{i,j} = 0$). Therefore, the Async Rule is satisfied in the generated adapter.

Second, IP11 specifies that if there is an adaptation requirement for $S_i$ and $S_k$ ($RR_{i,k} == 1$ and $SS_{i,k} == 0$), we check whether this can be met by other adaptations through a transivity (e.g. there exists $j$, s.t. $SS_{i,j} + XSS_{i,j} == 1$ and $SS_{j,k} + XSS_{j,k} == 1$). This check is exhaustive (for any $i$, $j$, $k$). Therefore, it guarantees that all redundant adaptation is avoided. Besides, we minimize the sum of $XSS_{i,j}$ and $TSS_{i,j}$, which avoids adapting unnecessary messages. Furthermore, we also add $SS_{i,j}$ in the objective function, which ensure that no false control flow information is added to the model.

In summary, the adapter generated from the Async-IP algorithm satisfies the Async Rule, and it adapts a minimal number of messages for a process composition. ■

This formulation was implemented in CPLEX. Here we show an example.

**Example 5-1.** For the case in Figure 5-3 (a), we implemented the IP model in CPLEX. The full CPLEX code is attached in Appendix C. The solution to this formulation found by CPLEX is as follows:

\[
x_{SS_{7,4}} = x_{SS_{8,1}} = x_{SS_{8,3}} = x_{SS_{9,6}} = 1
\]

\[
y_{SS_{6,9}} = 1
\]
$TSS_{0.4} = 1$

This solution means that out of the four adaptation elements required based on the $XSS_{7.4}, XSS_{0.1}, XSS_{8.3}$ and $XSS_{9.6}$ values, $XSS_{0.1}$ is redundant because $TSS_{0.4}$ is 1. Hence, only three adaptation elements are required. Moreover, $YSS_{6.5} = 1$ indicates that the sequence (6,9) should be reversed. The adapter created by combining these adaptation elements is shown in Figure 5-3 (C). In contrast, Figure 5-3 (b) shows a redundant adapter, which is generated based on the rules in Table 3-2.

![Figure 5-3. An illustrative example for the Async-IP approach](image)

In this chapter, we investigate how the pair-wise approach can be generalized to general cases in asynchronous communication. After introducing the impact of inter-process causal dependency on incompatibility detection, we presented another integer programming formulation to model the general cases in asynchronous communication, and show how it can produce an optimal solution. Compared with the general case analysis in synchronous communication, the extension in asynchronous communication is straightforward. The relationship analysis of inter-process causal dependency leads to a more accurate determination of incompatible cases, which minimize the redundancy in message adaptation, and therefore generates a smaller (minimal)
adapter. Please note that the selection of adapted messages in asynchronous communication is
deterministic. In other words, it is not possible to find two different adaptation sets that have the
same number of messages. This is different from that in synchronous communication.

Chapter 2-5 provide a comprehensive solution for the control flow adaptation in
synchronous and asynchronous communication. This approach is not only efficient (reducing the
search space via the patter-based analysis), but also optimal (creating an optimal via the pair-wise
analysis and IP modeling). In the next two chapters, we turn to study message adaptation, and
how to integrate it with control flow adaptation.
Chapter 6
Analysis of Message Mismatches and Design of Message Adaptation Patterns

Due to independent design efforts or non-standardized industry practices, generally the message interfaces between two services are not well matched. Consider a travel booking scenario in which a customer wishes to communicate with an airline company (Figure 6-1). The client wishes to communicate in two ways. She would like to first request the flight information between two cities, and choose one appropriate flight. Then, she reserves tickets for a group of people (e.g. family members, department crew, etc.) using same flight and date information. At the same time, she sends out the payment to the flight service. On the other hand, the interface of the flight booking service requires the arrival and departure cities, as well as the date information for the flight(s) to be provided. And, it accepts flight reservation requests in an iterative way for one passenger at a time.

Figure 6-1. Illustrative example for message mismatch
Obviously, these two parties cannot communicate with each other due to the existing mismatch in messages. Their message mismatches are summarized in Table 6-1 (The mismatch types and adaptation patterns will be discussed shortly). Therefore, message adapters are needed to reconcile this incompatibility. In this section, we first identify the typical mismatches in service message communications. Next, we propose a set of standard patterns for message adaptation. These patterns can be further concatenated, nested, or extended by users.

Table 6-1. Message mismatches in the case of Figure 6-1

<table>
<thead>
<tr>
<th>Customer</th>
<th>Airline company</th>
<th>Mismatch type</th>
<th>Adaptation pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>departCity (atomic)</td>
<td>departCity (atomic)</td>
<td>Data type mismatch</td>
<td>matched</td>
</tr>
<tr>
<td>arriveCity (atomic)</td>
<td>arriveCity (atomic)</td>
<td>Data type mismatch</td>
<td>matched</td>
</tr>
<tr>
<td>date (atomic)</td>
<td>date (atomic)</td>
<td>Missing information</td>
<td>Data generation</td>
</tr>
<tr>
<td>Reserv1 (tuple with list)</td>
<td>Reserv2 (tuple in LOOP)</td>
<td>Data type mismatch</td>
<td>matched</td>
</tr>
<tr>
<td>flight (atomic)</td>
<td>flight (atomic)</td>
<td>Tuple-to-Tuple</td>
<td>none</td>
</tr>
<tr>
<td>date (atomic)</td>
<td>date (atomic)</td>
<td>Data type mismatch</td>
<td>matched</td>
</tr>
<tr>
<td>PassengerList (list)</td>
<td>passenger (atomic)</td>
<td>Data type mismatch</td>
<td>matched</td>
</tr>
<tr>
<td>Payment (tuple)</td>
<td>Payment (tuple)</td>
<td>matched</td>
<td>N/A</td>
</tr>
<tr>
<td>TicketList (list)</td>
<td>TicketList (list)</td>
<td>matched</td>
<td>N/A</td>
</tr>
<tr>
<td>receipt (atomic)</td>
<td>receipt (atomic)</td>
<td>matched</td>
<td>N/A</td>
</tr>
</tbody>
</table>

6.1 Mismatches in Messages

We distinguish six mismatches in two groups: a) Syntactic mismatches, and b) Information mismatches.

The syntactic mismatches may occur between two atomic message types, one complex and several atomic message types, or two complex message types. For instance, some companies may provide date information as a string, but others may format this information as a date data type.
Another example is the flight request messages in Table 6-1. The customer provides `departCity` and `arriveCity` as atomic message, but the airline company wants to receive them in a tuple message `(Request(departCity, arriveCity, date))`.

The information mismatches refer to a higher level of message mismatches. In such cases: 1) the data provided by one party is not required by the other party; 2) the data required by one party is not provided by the other party; or, 3) the data required by one party can be transformed or calculated from the data provided by the other party. For instance, in Table 6-1 the `date` in the `Request` message is not provided by the customer.

### 6.2 Standard Patterns of Message Adaptation

To resolve the types of mismatches defined above, here we propose a set of patterns for message adaptation.

#### 6.2.1 Message data generation

In some mismatch cases, although the required data is not provided, we can still generate default information for it based on specific scenario semantics. One aspect of semantics is based on data types of messages. Some common examples for message semantics are:

- The default date is today.
- The default time is "now".
- The default units system is metric in Europe or US units in America.

In the example of Table 6-1, the `date` in the `Request` message can be generated by default. Another aspect of semantics is from application, which provides the context for this message. For
instance, in an online travel service the default name for a flight booking request is the customer name saved in the account profile of that logged-in user.

6.2.2 Message data casting

In some cases, the data required by one party can be calculated or transformed from the data provided by the other party. For instance, if one party needs the number of participants, while the other party gives the name list of participants, he can use a count function on the name list to get the participant number. We can also use abstraction semantics to perform abstract operations, such as aggregation or disaggregation in a predefined hierarchy. For instance, if there is a time hierarchy of year, month, week and day, sales data could be abstracted up or down this hierarchy, such as year → month, month → day, or week → day.

6.2.3 Message fragmentation and aggregation

Message fragmentation means that a large message is divided into sub-elements in a 1-to-n relationship, while message aggregation means the opposite. Typical scenarios involving message fragmentation are (Figure 6-2):

1) **Tuple-to-Element**: The adapter converts a tuple-loaded message into a sequence of messages each loaded with an element of the tuple. This is based on type deconstruction and message buffering. The adapter will buffer and split the tuple into elements (type deconstruction) and then send each element separately, as shown in Figure 6-2 (a). The reverse pattern **Element-to-Tuple** is shown in Figure 6-2 (b). In general, each adapter has three stages as shown in the examples of Figure 6-2: receive, transform and send.
2) **List-to-Element**: The adapter converts a list-loaded message into a sequence of messages each loaded with an element of the list (Figure 6-2(c)).

Aggregation patterns are similar to these patterns. For instance, in Table 6-1 the transformation between `departCity`, `arriveCity` and `Request` can be fulfilled by an `Element-to-Tuple` aggregation pattern.
6.2.4 Message reconstruction

In some cases, two parties may both have complex data types, such as tuple and list, which share a common set of basic elements, but have different orders or structures. In such a situation, we need message reconstruction patterns as follows.

1) **Tuple-to-Tuple**: A tuple may be rearranged into one or more tuples in various ways. For instance, send elements a, b in different order (in one tuple or separate tuples).

2) **List-to-List(m)**: In this scenario, only m elements from a list or array may be sent in one message at a time due to the size limit on the corresponding list or array of the other party. Hence, in general, several such messages must be sent. This leads to the need for control flow adaptation because now after each such message is sent, the server must be informed whether the transmission of this data has been completed, or if more is to follow. Consequently, the adapter must: 1) reorganize/break the data into smaller pieces, and 2) ensure these pieces are sent according to the flow protocol required by the service process (Figure 6-2 (d)).

These adaptation patterns show that in a complete adaptation solution we need to consider both message interface information and control flow information, because: 1) the control flow structure of the processes may decide what message transformation is needed in adaptation; and 2) the message adapter process should be carefully designed in order to avoid introducing new control flow incompatibility. Therefore, we have to integrate the message and control flow adaptation as a whole, which we discuss in the next chapter.
6.3 Nesting, Concatenating, and Extending patterns

The basic patterns can be further concatenated or nested to deal with more complicated situations. Here is an example.

**Example 6-1.** In the example of Table 6-1, a flight reservation (reserv1) is sent from the customer (party A) as reserv1(flight, date, PassengerList). However, the airline company (party B) expects individual reservation as reserv2[flight, date, passenger], and it allows multiple such transmissions in the same session. For this example, we can nest a Tuple-to-Tuple pattern (for flight and date information) with a List-to-Element pattern (for passenger information), as shown in Figure 6-3. The abbreviated notation for the adapters in the figure shows the name of the adapter pattern, and the action of the transform stage. Thus, the nesting is easily done by connecting two patterns in a sequence.

![Figure 6-3. Tuple-to-Tuple with List-to-Element](image)

Further, users may define new message adaptation patterns, and specify their relations with existing ones for the purpose of nesting and concatenation. Here is an example.
Example 6-2. User may define an M-to-N pattern as shown in Figure 6-4, in which a set of M tuple messages can be transformed to another set of N tuple messages. These two sets of tuple messages contain the same group of elements. But there is no one-to-one or one-to-many mapping between messages. Therefore, we need to design a new adaptation pattern to handle it. This pattern can be easily concatenated or embedded with other patterns in Figure 6-2 to fulfill more advanced adaptation tasks.

![Figure 6-4. M-to-N pattern](image)

6.4 Summary

Current approaches for business process adaptation focus on either the message interface aspect or the control flow interface aspect separately. Our work recognizes that message adaptation may affect control flow adaptation and vice versa in complex ways. Hence an integrated approach is necessary. In this chapter, we identified a set of extendible message adaptation patterns to solve typical message mismatches. At adapter creation time a user can
choose any valid patterns or their combination, and apply them to interact with a variety of services. Another interesting feature of our approach is that the patterns are extendible. Thus, a user may add more message transformation patterns and define their relation with existing ones. In the next chapter, we show how to detect the message mismatches and then generate a message adapter if necessary. Furthermore, we show how the generated adapter can be combined with control flow adaptation by another algorithm to create a complete adapter for multiple processes.
Chapter 7

Algorithms for Integrated Adapter Creation

In this chapter, we first introduce the algorithms to create a message adapter by automatically detecting message transformation requirements, and then discuss how to integrate it with the control-flow adapter.

7.1 Definitions of Data Types and Messages

In this work, we consider three data types as follows.

- A simple type $S$ is a primitive data type such as integer, string, datetime, etc. For a detailed list of primitive data types used in XML, please refer to [W3C XML Schema WG 2004].

- A tuple type $T=T_{\text{name}}[I_1, I_2, \ldots, I_n]$, where $I_1, I_2, \ldots, I_n$: S, T, or L, and $n$ is a positive integer. A tuple type has a name $T_{\text{name}}$, and it consists of a finite set of items $I_1, I_2, \ldots, I_n$. Each item $I_i$ has a unique name, and its type could be S, T, or L. There is no order among items. Therefore, $T_1[A,B,C]$ is same as $T_1[C,A,B]$.

- A list type $L=L_{\text{name}}<I, n>$. where $I$: S, T, or L, and $n$ is a positive integer. A list type has a name $L_{\text{name}}$, and it consists of one item $I$ with a maximum occurrence limit $n$. The type of $I$ could be S, T or L.

A message $M$ could be of type S, T or L. Based on its type, a message is called a simple message (S type), tuple message (T type), or list message (L type). Tuple and list Message are also called composite message. An item in a message is called message component if the type of this item is either T or L. Otherwise, it is called message element. Therefore, a message element is an atomic item, while a message component is a composite item. Hence, a simple message is a
message element itself. And, a composite message is a message component itself. To differentiate
them in notation, generally we capitalize the first letter of composite items' name. In contrast, the
first letter of message elements' name is in lower case.

In order to simplify the notations in this paper, we omit the data type name and simple
type name when we present a message. For instance, the complete definition of a tuple type is,

PassengerType[firstName:string, lastName:string].

If a message is defined as

Passenger : PassengerType,

then its simplified notation is

Passenger[firstName, lastName].

**Example 7-1.** In Figure 7-1, we give a simple illustrative example. The messages in this
example are as follows,

Query[departCity, arriveCity, Passenger[firstName,lastName]]

Booking[airline, flight#, date]

Request[departCity, arriveCity, airline]

Reservation[flight#, date, Passenger[firstName, lastName]]

As shown in Figure 7-2, each message can be represented by an unordered tree, e.g.
Query, Booking, Request, and Reservation. One message tree may contain a number of internal
nodes (message component) and leave nodes (message element). In this example, Passenger is a
message component, and departCity and lastName are message elements. ■
Based on the definition of types and messages, we give the matching definitions for messages as follows.

**Definition 7-1.** Two message elements are *matched* if and only if they have a same name.

Two matched message elements may have different primitive data types, e.g. string vs. dateTime. The transformation between basic data types can be easily done by XPATH functions. Those simple transformations are not the focus of this work. Therefore, for simplicity, we assume that matched message elements have same basic data type.

**Definition 7-2.** Two message components are *matched*, if and only if,

1. They have a same name,

2. If they are of Tuple types, each of their items is matched.
3. If they are of List types, their items are matched, and their max capacities are same.

In a process composition involving message transformation, generally all output message elements and all input message elements are well matched. In other words, any information provided by a process can be consumed by another process, or perished in the adapter. On the other hand, any information required by a process can be provided by another process in a direct (no transformation required) or indirect (transformation required) way. Or, this information can be generated by default in the adapter based on the application scenario. Otherwise, we need manual intervention to add additional data sources, which is out of scope of this paper. Therefore, in this work we assume that each message element in a process can be: 1) matched to one and only one message element in another process; 2) perished in the adapter if it is an output; or 3) generated by default in the adapter if it is an input.

### 7.2 Compatibility Verification

The compatibility verification for the process composition involving message transformation is similar to the approach used in the process composition involving only control-flow mismatches (Section 2.4). In that approach, we transform interacting processes into a directed graph (communication graph), and then check if there is any cycle in it. The absence of cycle means that the process composition is deadlock-free. Here, we follow the same idea. The only difference is how to add communication links. In Section 2.4, we add links from each send activities to its corresponding receive activity. These two activities work on the same message, and they are dual of each other. However, in the integrated adaptation cases such as Example 7-1 in Figure 7-1, we do not have such exact message matching. Instead, the communication dependency exists from a send activity to a receive activity if at least one message element is matched between the send's output and the receive's input.
**Example 7-2.** In Figure 7-3, we show another example, and its communication graph. In this example, Send(T1[A,B]) overlaps with Receive(T3[A,D]) at A, and overlaps with Receive(T4[B,C]) at B. Therefore, we add links from Send(T1[A,B]) to Receive(T3[A,D]) and Receive(T4[B,C]). For the same reason, add links from Send(T2[C,D]) to Receive(T3[A,D]) and Receive(T4[B,C]). The complete communication graph is shown in Figure 7-3. Since this graph is absent from cycle, this example is a deadlock-free case, i.e. we can always create an adapter to make the process composition compatible.

**Example 7-3.** Figure 7-4 shows another example. In this example, we detect one cycle as shown in Figure 7-4 (c). Therefore, this is a deadlocked case, for which we are unable to make it compatible even with the support of adapter.

![Figure 7-3. Illustrative example (Example 7-2)](image-url)
Matching of Message Elements

If a process composition is verified to be deadlock-free we can always create an integrated adapter to make the composition compatible. To create this adapter, the first step is to match the message elements and components between processes. In this section, we introduce the algorithm to match message elements and components. We use Example 7-2 in Figure 7-3 to illustrate each step. The matching information obtained from this algorithm will help us to identify the required message transformation between activities, which we explain in the next section.

Based on the definitions in Section 7.1, we design the matching algorithm as follows,

**Algorithm 7-1.** Message matching algorithm (Msg-Matching)

1) For each process \( p_i \), list all its output message in an array \( OM \) (output messages), and input messages in an array \( IM \) (input messages), by process name. For Example 7-2 in Figure 7-3, the OM of Proc 1 and the IM of Proc 2 are listed in Table 7-1 and Table 7-2. Moreover, the OM of Proc 2 and the IM of Proc 1 are listed in Table 7-3 and Table 7-4.
Table 7-1. Output message for Proc 1

<table>
<thead>
<tr>
<th>Message</th>
<th>Process</th>
<th>Message Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Proc1</td>
<td>T1[A,B]</td>
</tr>
<tr>
<td>Y</td>
<td>Proc1</td>
<td>Y</td>
</tr>
<tr>
<td>T2</td>
<td>Proc1</td>
<td>T2[C,D]</td>
</tr>
</tbody>
</table>

Table 7-2. Input messages for Proc 2

<table>
<thead>
<tr>
<th>Message</th>
<th>Process</th>
<th>Message Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T3</td>
<td>Proc2</td>
<td>T3[A,D]</td>
</tr>
<tr>
<td>Y</td>
<td>Proc2</td>
<td>Y</td>
</tr>
<tr>
<td>T4</td>
<td>Proc2</td>
<td>T4[B,C]</td>
</tr>
</tbody>
</table>

Table 7-3. Output messages for Proc 2

<table>
<thead>
<tr>
<th>Message</th>
<th>Process</th>
<th>Message Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Proc2</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 7-4. Input messages for Proc 1

<table>
<thead>
<tr>
<th>Message</th>
<th>Process</th>
<th>Message Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Proc1</td>
<td>X</td>
</tr>
</tbody>
</table>

2) For each message \( m_j \) in \( OM \) or \( IM \), list all its message elements with XPATH information in an array \( OME \) (output message element) or \( IME \) (input message element). Table 7-5, 7-6, 7-7, and 7-8 show the OMEs and IMEs for Example 7-2.

Table 7-5. Output message elements for Proc1

<table>
<thead>
<tr>
<th>Message Element</th>
<th>Message</th>
<th>Process</th>
<th>XPATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T1</td>
<td>Proc1</td>
<td>T1/A</td>
</tr>
<tr>
<td>B</td>
<td>T1</td>
<td>Proc1</td>
<td>T1/B</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Proc1</td>
<td>Y</td>
</tr>
<tr>
<td>C</td>
<td>T2</td>
<td>Proc1</td>
<td>T2/C</td>
</tr>
<tr>
<td>D</td>
<td>T2</td>
<td>Proc1</td>
<td>T2/D</td>
</tr>
</tbody>
</table>
Table 7-6. Input message elements for Proc2

<table>
<thead>
<tr>
<th>Message Element</th>
<th>Message</th>
<th>Process</th>
<th>XPATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T3</td>
<td>Proc2</td>
<td>T3/A</td>
</tr>
<tr>
<td>D</td>
<td>T3</td>
<td>Proc2</td>
<td>T3/D</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Proc2</td>
<td>Y</td>
</tr>
<tr>
<td>B</td>
<td>T4</td>
<td>Proc2</td>
<td>T4/B</td>
</tr>
<tr>
<td>C</td>
<td>T4</td>
<td>Proc2</td>
<td>T4/C</td>
</tr>
</tbody>
</table>

Table 7-7. Output message elements for Proc2

<table>
<thead>
<tr>
<th>Message Element</th>
<th>Message</th>
<th>Process</th>
<th>XPATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>Proc2</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 7-8. Input message elements for Proc1

<table>
<thead>
<tr>
<th>Message Element</th>
<th>Message</th>
<th>Process</th>
<th>XPATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>Proc1</td>
<td>X</td>
</tr>
</tbody>
</table>

3) Match message elements in OME and IME using Definition 7-1. The matching table for the example is in Table 7-9.

Table 7-9. Matching table for Example 7-2

<table>
<thead>
<tr>
<th>Message Element</th>
<th>Output/Source</th>
<th>Input/Target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Process</td>
<td>Message</td>
</tr>
<tr>
<td>A</td>
<td>Proc1</td>
<td>T1</td>
</tr>
<tr>
<td>B</td>
<td>Proc1</td>
<td>T1</td>
</tr>
<tr>
<td>Y</td>
<td>Proc1</td>
<td>Y</td>
</tr>
<tr>
<td>C</td>
<td>Proc1</td>
<td>T2</td>
</tr>
<tr>
<td>D</td>
<td>Proc1</td>
<td>T2</td>
</tr>
<tr>
<td>X</td>
<td>Proc2</td>
<td>X</td>
</tr>
</tbody>
</table>

4) Identify matched message components based on Definition 7-2. If two message components are matched, we do transformation for these matched message
components directly instead of their sub message elements. This saves the copying efforts in the assignment activity. Given two matched message components, we update the matching table as follows: 1) adding the matched message components; and 2) deleting the message elements included in the matched message components. This matching procedure is iterated until no more message components can be matched.

5) Update the matching table by deleting matched messages. If two matched message elements or components are messages themselves, we do not need to adapt this message. The related two activities can directly communicate with each other to deliver this message. Therefore, we need to delete them from the matching table. In Example 7-2, we have two matched messages, X and Y. They are highlighted in red in Table 7-9. We delete them, and update the matching table as shown in Table 7-10.

<table>
<thead>
<tr>
<th>Message Element</th>
<th>Output/Source</th>
<th>Input/Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Proc1</td>
<td>T1</td>
</tr>
<tr>
<td>B</td>
<td>Proc1</td>
<td>T1</td>
</tr>
<tr>
<td>C</td>
<td>Proc1</td>
<td>T2</td>
</tr>
<tr>
<td>D</td>
<td>Proc1</td>
<td>T2</td>
</tr>
</tbody>
</table>

7.4 Algorithm to Create Message Adapter for Tuple Messages

In this section, we discuss how to create a message adapter for tuple messages in a process composition. The handling of list messages will be discussed in next section.
Based on the matching table, the algorithm for message adapter creation is designed as follows,

**Algorithm 7-2.** Tuple message adapter creation algorithm (Msg-Tuple)

1) Create receive activity $R_i$ for each message $m_i$ appearing in the output column of the matching table.

2) Create send activity $S_j$ for each message $m_j$ appearing in the input column of the matching table.

3) After each receive activity $R_i$, create an Assignment activity $C_i$ to include all the copying for sub message components $m_{c_i}$ or elements $m_{e_i}$.

4) Create a control flow link from each Assignment activity $C_i$ to each of those send activities $S_j$ for which there exists a row in the matching table between the message components or elements of $m_i$ and $m_j$.

5) Create a control flow link between two activities $a$ and $b$, if there exists a control flow dependency between dual(a) and dual(b) in their process.

6) Delete redundant control flow links.

7) If there are isolated pieces, put them into a parallel structure. ■

In this algorithm, Step 5 and Step 6 rely on the process normalization and sliming procedures. For the details of the process normalization procedure, please refer to Section 2.4.1. For the process slimming procedure, we discuss the details in Appendix B.

Figure 7-5 shows the message adapter for Example 7-2. The Assignment activities are highlighted with a blue background.
7.5 Create Message Adapter for List Messages

Compared with tuple message, the adaptation and transformation of a list message generally perform a suitable 1-to-n or n-to-1 mapping, and require a loop structure in the message adapter. This makes it challenging to create a message adapter and integrate with other adaptation elements. The situation is even worse when a list message is embedded with a tuple message as shown in Figure 6-3. In order to handle this difficulty, we label the group of n activities as a block (see Figure 7-6(a)) and store the block in the messageAdapters array (see Figure 7-6(c)). The block activity can now be treated as an atomic activity in the adapter creation algorithm. The send activity in the src unit and the block activity in the tgt unit form a send-block pair, which means that they are full duals of each other with identical arguments. In this way, all message adapter information between two processes is saved in the messageAdapters array, and can be retrieved by invoking messageAdapters[src,tgt].
Based on the idea of block activity, the message adapter creation algorithm works as follows:

**Algorithm 7-3.** Message adapter creation algorithm (Msg-Adapter-Creation)

1) Identify the innermost list message in processes, and check if it needs message adaptation. If yes, go to Step 2. Otherwise, repeat this step.

2) If the item of this list message is of a tuple type, use the tuple message adapter creation algorithm (Section 7.4) to generate message adapter for matched activities with only tuple and simple message, and create an message adapter for them.

3) Replace these loop structures in the adapter and the process with two new Block activities, and store their BlockIDs and the adapter in a BlockAdapters array. This form a send-block or block-receive pair, in which they are full duals of each other. They are treated as atomic activities like send and receive in the following steps.

4) Repeat steps 1-3 until no more list message remain.
5) Use the Msg-Tuple algorithm (Section 7.4) to generate the complete message adapter for the remaining processes (with only tuple and simple messages).

### 7.6 Integration of Message and Control-flow Adaptations

After the message adapter is created, the checking of the control flow incompatibility is same as before. Here we give an example to illustrate it.

**Example 7-4.** Consider the case in Figure 7-3 (a). We add its message adapter (Figure 7-5) between the two processes to form a new composition scenario, as shown in Figure 7-7. Then, we combine two processes and the message adapter to build the communication graph (Figure 7-8), for which we can generate a MIG (Figure 7-9(a)) to detect the control flow incompatibility. Please note that the *assignment* activities in the message adapter process are omitted, because they are regarded as the internal/private activities of the message adapter processes. We use the MIG algorithm to identify one cycle (Figure 7-9(b)) and its corresponding adaptation element (Figure 7-9(c)). Then, we combine them with the message adapter to create an integrated adapter process (Figure 7-10). We can either simply put all adapters in parallel (Figure 7-10(a)), or further slim it by introducing the control flow links inherited from existing processes (Figure 7-10(b)).
Figure 7-7. New composition scenario in Example 7-4

Figure 7-8. Communication graph in Example 7-4
Figure 7-9. MIG analysis in Example 7-4

Figure 7-10. Integrate adapters in Example 7-4
7.7 Summary

In this chapter, we discussed how to generate a message adapter, and then integrate it with the control flow adapter to provide a complete adaptation solution. Our approach is based on building and using message adaptation patterns introduced in Chapter 6. We first studied how to extend the compatibility verification algorithm (Section 2.4.2) to handle message mismatches. Then, based on a message matching algorithm, we introduced an adapter creation algorithm for tuple messages, and another algorithm to generate a message adapter by including the consideration of list messages. Furthermore, we showed how to extend the MIG-based analysis with the generated message adapter, and consequently create a complete adapter for multiple processes.

Now, we have presented the complete solution for business process adaptation. Our approach covers both control flow adaptation and message adaptation in synchronous and asynchronous communication. In the next chapter, we present the design of a prototype system, in which we implemented all these algorithms. Furthermore, we give the results of the validation tests and performance experiments to show the effectiveness and efficiency of our approach.
Chapter 8

Implementation, Validation and Evaluation

In this chapter, we present a Java-based prototype system, in which we implemented the adapter creation algorithms for WS-BPEL processes. To comprehensively evaluate the performance of our algorithms, we designed three different experiments, and reported their results.

8.1 System Architecture

An initial prototype system called BPCT (Business Process Composition Tool) was implemented in Java. Figure 8-1 shows the simplified architecture of the BPCT system. The WS-BPEL Parser receives .bpel files, .wsdl files, and transforms them into an internal process model (AND/OR graph) with the Internal Process Builder. They are stored in a local database, and may be invoked multiple times during the compatibility verification and adapter creation. The Composition Analyzer is in charge of the coordination of the whole analysis. It takes a user's commands and runs the compatibility verification and adapter creation accordingly. The Compatibility Verifier runs the compatibility verification algorithm on the parsed processes (Chapter 2). It invokes the external Process Correctness Checker, i.e. DiagFlow [Eshuis and Kumar 2010] a publicly available Java-based tool for verifying the control flow of process models. If the compatibility of a process composition is verified to be true, the Adapter Creator is called to generate an adapter based on selected communication mode. If the message interfaces are not well matched, we first call the message adapter creation algorithm (Section 7.5) to generate a message adapter. Then, if a user chooses synchronous communication, the MIG-IP
algorithm (Chapter 4) is used for the SP adapter creation. In this algorithm, the Adapter Creator calls the external IP Solver, i.e. Lp_solve [Berkelaar 2010] an open source Mixed Integer Linear Programming (MILP) solver, to solve the IP formulation. If asynchronous communication is chosen, the Adapter Creator runs the modified pair-wise analysis algorithm (Chapter 5). After the control flow adapter is generated, we integrate it with the message adapter to create a complete adapter (Chapter 7). Finally, the Composition Analyzer summarizes diagnosis, and sends results to the user along with adapter information.

Figure 8-1. Architecture of BPCT

8.2 Test Environment

To validate the correctness and evaluate the performance of our approach, we need a large number of testing cases. However, there are limited examples about process cooperation in literature. Therefore, we designed a Test Case Generator (TCG) to randomly create process pairs.
Process size is defined as the number of activities in a process. Given a specific process of size $n$, TCG generates $n$ messages, and for a message $i$ a send activity ($S_i$) and a receive activity ($R_i$) are created. TCG randomly allocate $S_i$ and $R_i$ to two processes (A and B). Therefore, Process A has $n$ activities, as does Process B. For each process, all the activities are added into a pool. TCG randomly selects two activities from the pool and assigns a composition pattern (either an S or a P) to them. Then, these two selected activities are deleted from the pool and the generated composite activity is added into the pool. TCG repeats these steps until there is only one activity left in the pool, which is the root activity for that process. In this way two random processes are constructed.

For the validation tests and the performance experiments, a large number of test cases must be generated. To prevent duplicates, unique cases are recorded in a database, and each newly generated case is checked against this database for uniqueness. Uniqueness is determined by parsing the syntax of processes. For instance, if $P_1=\text{Seq}(S_1,S_2)$ and $P_2=\text{Seq}(S_2,S_1)$, $P_1$ and $P_2$ are different. But, if $P_1=\text{And}(S_1,S_2)$ and $P_2=\text{And}(S_2,S_1)$, $P_1$ and $P_2$ are equivalent. Two cases are equivalent if and only if there is a one-to-one mapping between each of their containing processes. However, we do not consider isomorphic situations. For example, let case1=[$P_1=\text{Seq}(S_1,S_2), P_2=\text{And}(R_1,R_2)$] and case2=[$P_1=\text{Seq}(R_2,R_1), P_2=\text{And}(R_2,R_1)$]. These two cases have the same structure with $m_1$ and $m_2$ switched. We regard them as different cases, since we assume $m_1$ and $m_2$ may have different semantic meaning.

Besides the test cases, we also need a tool to verify if the process composition is correct. Therefore, we designed a Process Execution Simulator (PES) to simulate the activity interactions in a composition scenario. Given a set of processes ($P_1, \ldots, P_m$) including adapters if any, PES puts the initial activity(ies) of each process into an execution pool. Then, PES checks whether any two activities in this pool are duals of each other. If so, these two activities are deleted from the pool and their succeeding activities are added into the execution pool. PES repeats these steps
until there is no pair of activities remaining in the pool can be executed. If the execution pool is empty, the composition succeeds. Otherwise, the composition fails indicating that the two processes cannot be composed.

8.3 Validation Test

The validation test is constructed as follows.

1) Specify a process size and the case number, and invoke TCG to generate test cases
2) For each test case, run the deadlock checking algorithm. The possible results are deadlocked and non-deadlocked.
3) For each test case, run the SP adapter creation algorithm (SPA) and generate an adapter if possible. The possible results are non-compatible, compatible with adapter, and compatible without adapter.
4) For each test case, run CES on the processes in the test case. The possible results are successful without adapter and failed without adapter.
5) For each test case, if an adapter is created from SPA, run CES on the processes in the test case and the generated adapter. The possible results are successful with adapter and failed with adapter.

Result 4-2 is validated if the following conditions are satisfied for each test case:

1. A test case is deadlocked if and only if it is non-compatible and failed without adapter.
2. A test case is non-deadlocked if and only if it is compatible with adapter or compatible without adapter.

The SPA algorithm is validated if the following conditions are satisfied for each test case:

1. If a test case is non-compatible, it is failed without adapter.
2. A test case is *compatible without adapter* if and only if it is *successful without adapter*.

3. A test case is *compatible with adapter* if and only if it is *successful with adapter*.

In Table 8-1, we show the results of 10000 test cases for the processes of size 5, 10, 20 and 50. It shows that:

- The numbers of "Execution Succeeds" cases in Part 1 (w/o adapter) are same as the number of "Empty Adapter" cases in Part 2 (w/adapter);
- The numbers of "Execution Fails" cases in Part 1 (w/o adapter) are same as the number of "Execution Succeed w/ adapter" cases in Part 2 (w/adapter);
- The numbers of "Execution Fails w/ adapter" cases in Part 2 (w/adapter) are all 0.

Therefore, all the test cases passed the validation tests for Result 4-2 and SPA algorithm.

We note that as the number of messages increases, the proportion of deadlocked cases increases significantly, and *the number of cases of successful execution without adapter is nearly 0 when the process size is larger than 10*. This suggests that *in a complex system it is difficult to make the collaboration work without the aid of adapters.*

<table>
<thead>
<tr>
<th># of Message</th>
<th>Number of Case</th>
<th>Deadlock</th>
<th>Part 1 (w/o Adapter)</th>
<th>Part 2 (w/ Adapter)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Execution Succeeds</td>
<td>Execution Fails</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Empty Adapter</td>
<td>Execution Succeed w/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>adapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Execution Fails w/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>adapter</td>
</tr>
<tr>
<td>5</td>
<td>10000</td>
<td>Deadlock</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2092</td>
<td>903</td>
<td>7005</td>
<td>903</td>
</tr>
<tr>
<td></td>
<td>20.9%</td>
<td>9.0%</td>
<td>70.1%</td>
<td>9.0%</td>
</tr>
<tr>
<td></td>
<td>5190</td>
<td>23</td>
<td>4787</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>51.9%</td>
<td>0.2%</td>
<td>47.9%</td>
<td>0.2%</td>
</tr>
<tr>
<td></td>
<td>8492</td>
<td>0</td>
<td>1508</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>84.9%</td>
<td>0%</td>
<td>15.1%</td>
<td>0%</td>
</tr>
<tr>
<td>10</td>
<td>10000</td>
<td>Deadlock</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2092</td>
<td>903</td>
<td>7005</td>
<td>903</td>
</tr>
<tr>
<td></td>
<td>20.9%</td>
<td>9.0%</td>
<td>70.1%</td>
<td>9.0%</td>
</tr>
<tr>
<td></td>
<td>5190</td>
<td>23</td>
<td>4787</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>51.9%</td>
<td>0.2%</td>
<td>47.9%</td>
<td>0.2%</td>
</tr>
<tr>
<td></td>
<td>8492</td>
<td>0</td>
<td>1508</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>84.9%</td>
<td>0%</td>
<td>15.1%</td>
<td>0%</td>
</tr>
<tr>
<td>20</td>
<td>10000</td>
<td>Deadlock</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2092</td>
<td>903</td>
<td>7005</td>
<td>903</td>
</tr>
<tr>
<td></td>
<td>20.9%</td>
<td>9.0%</td>
<td>70.1%</td>
<td>9.0%</td>
</tr>
<tr>
<td></td>
<td>5190</td>
<td>23</td>
<td>4787</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>51.9%</td>
<td>0.2%</td>
<td>47.9%</td>
<td>0.2%</td>
</tr>
<tr>
<td></td>
<td>8492</td>
<td>0</td>
<td>1508</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>84.9%</td>
<td>0%</td>
<td>15.1%</td>
<td>0%</td>
</tr>
<tr>
<td>50</td>
<td>10000</td>
<td>Deadlock</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2092</td>
<td>903</td>
<td>7005</td>
<td>903</td>
</tr>
<tr>
<td></td>
<td>20.9%</td>
<td>9.0%</td>
<td>70.1%</td>
<td>9.0%</td>
</tr>
<tr>
<td></td>
<td>5190</td>
<td>23</td>
<td>4787</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>51.9%</td>
<td>0.2%</td>
<td>47.9%</td>
<td>0.2%</td>
</tr>
<tr>
<td></td>
<td>8492</td>
<td>0</td>
<td>1508</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>84.9%</td>
<td>0%</td>
<td>15.1%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>9953</td>
<td>0</td>
<td>47</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>99.5%</td>
<td>0%</td>
<td>0.5%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>9953</td>
<td>0</td>
<td>47</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>99.5%</td>
<td>0%</td>
<td>0.5%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 8-1. Results for the validation test
8.4 Performance Experiments

To comprehensively evaluate the performance of our algorithms, we designed three different experiments as follows,

1. Comparison of adapter creation algorithms: In this test, the adapters were evaluated primarily based on the business process complexity metrics [Cardoso et al. 2006], such as Number of activities (NOA) and Number of control elements (NOAC), that measure the number of activities and control structures in them. We ran the performance experiment on the MIG-Greedy, MIG-IP, PBA-MIN, and the Service Execution Tree (SET) [Brogi and Popescu 2006] methods, and show the advantages of our algorithm in synchronous communication. Furthermore, we compare the optimal adapter size in synchronous and asynchronous communication, and discuss the causes of their differences.

2. Optimizing the adapter based on message sizes: In this test, we consider messages of different sizes to check how it affects the creation of the optimal adapter. We modify the objective function in the MIG-IP algorithm (for synchronous communication) to minimize the total size of the adapted messages as opposed to minimizing the total number of adapted messages. The MIG-IP algorithm is able to select those messages with smaller size. In the asynchronous communication such optimization based on message size is not possible because the set of messages to be adapted is deterministic. Therefore, in this test we focus on the analysis of the MIG-IP algorithm for synchronous communication.

3. Security and privacy test: In this test, we include security and privacy constraints. As discussed in Section 4.2.3, adding more such constraints reduces adaptation flexibility. We define the adaptability ratio metric as the ratio between the number of
adaptable cases and the total number of cases. For the same reason in the sensitivity
test, we focus on the analysis of the MIG-IP algorithm, and study how the
adaptability rate changes when more constraints are added.

8.4.1 Comparing adapter complexity with process metrics

8.4.1.1 Comparison of adapter creation algorithms in synchronous communication

One way to evaluate the performance of our adapter creation algorithms is to compare the
complexity of the created adapters. Since the adapters themselves are business processes, we can
use the existing evaluation metrics for a business process model in the literature. In this section,
we used four major metrics of business process complexity. Using these metrics, we ran the
performance experiment on the MIG-IP, MIG-Greedy, PBA-MIN, and the Service Execution
Tree (SET) [Brogi and Popescu 2006] methods. The results will be discussed shortly after
introducing the metrics.

Business processes play a key role in the implementation of large-scale information
systems. There is an increasing demand for insight on how to avoid errors, to facilitate
maintenance, and how to improve the process quality. Some evidence has shown that process
complexity is a measure of the likelihood of a business process having an error [Mendling et al.
2006]. As the process becomes more complex, the difficulty of locating and correcting problems
rises dramatically. Therefore, a business process complexity metric reflects whether the business
process has a clear structure, and is easy to analyze, understand or explain, and further improve
through analysis. Some of the metrics originated from the research on software complexity. In
this section, we briefly introduce four major metrics (NOA, MCC, CFC and HPC) for business
process complexity [Cardoso, et al. 2006]. As also in the in software engineering area, there is no
single measure of process complexity, which can serve as a universal predictor for business processes.

**NOA Metric**

The NOAC (Number of activities and control-flow elements in a process) metric gives the sum of the activities and the process control-flow elements in a process. Thus, we count the number of activities, and the control activity pattern pairs, a split-join pair being counted only once since we assume that the processes are well structured, and so are the adapters.

NOAC = Number of activities and control-flow elements in a process

**MCC Metric**

MCC (McCabe’s Cyclomatic Complexity) is an indication of a program module’s control-flow complexity and has been found to be a reliable indicator of complexity in large software projects [Zuse 1991]. MCC is defined as follows:

\[ MCC = e - n + 2, \]

where \( e \) and \( n \) are the number of arcs and nodes in the control flow graph, respectively. The nodes include both the activity and the pattern elements in process model. The arcs represent the control flow between nodes.

**CFC Metric**

Following Cardoso [Cardoso 2005], the CFC (Control-flow Complexity) metric is defined as follows (where \( P \) is a process and \( c \) is a control node in the process):

\[ CFC(P) = \sum_{XOR-splitt} CFC_{XOR-splitt}(c) + \sum_{AND-splitt} CFC_{AND-splitt}(c) + 2 \sum_{LOOP-splitt} c \]

where, \( CFC_{XOR-splitt}(c) = fan-out(c) \), and \( CFC_{AND-splitt}(c) = 1 \).
In this metric, for an XOR (or OR-split) node with fan-out of \( n \), exactly \( 1-of-n \) possible paths is taken in execution. Since this measure of complexity is based on the number of possible paths, every XOR-split with \( n \) outgoing transitions adds \( n \) to the CFC metric of this model. For a PAR (or AND-split) node, with \( n \) outgoing activities, all the outgoing activities must be done. Therefore, every AND-split in a model adds 1 to the CFC metric. The LOOP pattern is treated similarly to an XOR-split since it can be implemented using a 2-way XOR-split and join; therefore every LOOP-split adds 2 to the metric.

**HPC Metric**

Halstead’s metric [Halstead 1987] is a composite measure of psychological complexity. The HPC (Halstead-based Process Complexity) measures the length, volume, and difficulty of process, which are calculated as:

- Process Length: HPC-N = \( n1 \times \log_2(n1) + n2 \times \log_2(n2) \)
- Process Volume: HPC-V = \((N1 + N2) \times \log_2(n1 + n2)\)
- Process Difficulty: HPC-D = \((n1 / 2) \times (N2 / n2)\)

In this work, \( n1 \) is the number of unique activities and pattern elements, i.e. Sequence, Parallel, Choice, Loop, etc. \( n2 \) is the number of unique data items/messages manipulated by the process. Since the adapter has a Receive activity and a Send activity for each message, \( n2 \) is the half of the number of activities. \( N1 \) is the total number of operator occurrences; and \( N2 \) is the total number of operand occurrences.

To comprehensively evaluate the performance of our algorithms, we did the following experiment. For 5, 10, 15 and 20 messages, we used the TCG to create 500 non-deadlocked cases. For each case, we generated the MIG-Greedy and MIG-IP adapters as well as the PBA-MIN and the SET adapter. Then, we calculated the complexity metrics on each of them.

The results in Table 8-2 show that for each parameter setting, the average complexity of the PBA-MIN adapter is only half (or less than half) of the SET adapter complexity, and the
average complexity of the MIG-Greedy adapter is only half (or less than half) of the PBA-MIN adapter complexity. Compared with SET, the complexity of the adapter generated in the MIG algorithms decreases more than 70% in average. It is also evident that MIG-Greedy is just about as good as MIG-IP, with some exceptions. Thus, the greedy heuristic works very well here.

Table 8-2. Results of performance experiments

<table>
<thead>
<tr>
<th># of Messages</th>
<th>Metric</th>
<th>SET</th>
<th>PBA-MIN</th>
<th>MIG-Greedy</th>
<th>MIG-IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>M1:NOA</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>M2:NOAC</td>
<td>22</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>MCC</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>CFC</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>HPC N</td>
<td>14</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>HPC V</td>
<td>43</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>HPC D</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>M1:NOA</td>
<td>20</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>M2:NOAC</td>
<td>43</td>
<td>9</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>MCC</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CFC</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>HPC N</td>
<td>37</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>HPC V</td>
<td>112</td>
<td>16</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>HPC D</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>M1:NOA</td>
<td>20</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>M2:NOAC</td>
<td>43</td>
<td>15</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>MCC</td>
<td>14</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>CFC</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>HPC N</td>
<td>43</td>
<td>8</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>HPC V</td>
<td>124</td>
<td>34</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>HPC D</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>M1:NOA</td>
<td>40</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>M2:NOAC</td>
<td>83</td>
<td>28</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>MCC</td>
<td>25</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>CFC</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>HPC N</td>
<td>84</td>
<td>17</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>HPC V</td>
<td>252</td>
<td>63</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>HPC D</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

8.4.1.2 MIG-IP vs. MIG-Greedy

The optimal selection of adaptation elements is a hitting set problem, which is equivalent to the set cover problem [Cormen et al. 1990], a NP-hard problem. Intuitively, one would expect that an optimal IP solution would outperform the one from the greedy algorithm. Table 8-2 shows
that the greedy algorithm has similar average performance results to the IP algorithm. In order to find heuristics that might give an optimal solution faster we investigated matroid structures to represent our IP formulation [Faigle and Fujishige 2009; Whitney 1935]. Intuitively, if a matroid representation can be found for the IP formulation, then a greedy algorithm such as Kruskal algorithm [Kruskal 1956] always yields an optimal solution [Edmonds 1971]. However, such is not the case with our formulation and, in fact, as shown in Example 4-1 (see Page 56), the greedy algorithm yields a set \( \{m_1, m_3, m_4\} \), while the IP algorithm selects a smaller set \( \{m_4, m_6\} \).

In order to find the real reason of this similarity, we did an elaborative investigation on the test cases. We find that the various test cases can be categorized as follows:

1. **Compatible case**: no adapter is needed. In other words, both the algorithms generate an adapter of zero-size.

2. **Deterministic case with only I(1) compatibilities**: In this case, both the algorithms generate an adapter of the same size.

3. **Deterministic case with I(1) and I(2+) incompatibilities**: Here if the adaptation elements determined by the I(1) can resolve all the I(2+) incompatibilities, then both the algorithms generate an adapter of the same size.

4. **General nondeterministic adaptable case**: Here I(2+) incompatibilities remain after determining the adaptation elements from I(1) incompatibilities. In some of these cases, the greedy algorithm goes through a few iterations, while in others it iterates many times, before finding the complete solution.

The number of iterations in category 4 of the greedy algorithm influences the gap in the solution produced by it and the IP algorithm. The proportion of cases in the first three categories is so large that on average the advantage of the MIG-IP algorithm does not show up. To further verify this argument, we designed another experiment to compare the performance of the two algorithms. In this new experiment, we generated 1000 nondeterministic adaptable cases (all from
category 4). Using the iteration count of the MIG-greedy algorithm as a case difficulty metric, we classified those cases into different difficulty levels, and calculated the proportion of cases in which the IP algorithm generated a smaller adapter than the greedy one. Figure 8-2 shows the results of these experiments for two groups of settings. Figure 8-2 (a) shows results for 30 messages (30m), and varying number of processes, from 2 (p2) to 5 (p5). Figure 8-2 (b) gives the results for 3 processes (p3), but with the number of messages varying from 20 (m20) to 50 (m50). Both the charts show that the advantage of the IP approach increases significantly as the case difficulty rises. For those cases where the greedy approach needs more than 6 iterations to solve, the IP approach almost certainly creates a better adapter. It was also observed that the gap in the sizes of adapters produced by the two algorithms increased with case difficulty.

8.4.1.3 Comparing optimal adapters for synchronous vs. asynchronous communication

Asynchronous communication is based on each process maintaining instance-level message queues to facilitate communication between processes. Intuitively, one may expect that a queuing mechanism in asynchronous communication might improve the interaction between two processes, but it turns out that it makes the coordination more complicated. As Figure 3-3 (page...
34) shows, both synchronous and asynchronous communications lead to two incompatibilities. For each incompatibility, the adapter in the former case only needs to adapt one message, while the one in the latter case has to adapt two messages. Hence, on average, an asynchronous adapter might be twice as large as a synchronous one in terms of the number of adapted messages under certain assumptions. We did a preliminary test of both algorithms to compare the adapter sizes, i.e. the number of adapted messages assuming each message is associated with a unit size 1, in synchronous and asynchronous cases. We use the same sets of test cases in the previous experiment. The results in Table 8-3 show that when the average number of messages per process is more than 5, the average size of asynchronous adapters is much larger than the synchronous ones. Besides, we find two other explanations for this phenomenon:

Table 8-3. Adapter size comparison in synchronous and asynchronous communication

<table>
<thead>
<tr>
<th>Communication mode</th>
<th>Process Size (# of msg)</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Asynchronous</td>
<td></td>
<td>3</td>
<td>9</td>
<td>19</td>
<td>48</td>
</tr>
</tbody>
</table>

1) The adaptation elements in synchronous communication can be reused to resolve different incompatibilities. For instance, incompatibilities $B(m_1,m_2)$ and $B(m_1,m_3)$ share the same adaptation element $Seq(R1,S1)$. Hence, adding this element solves both incompatibilities. However, for asynchronous incompatibilities, each adaptation element is different. Although there may exist a number of identical, overlapping activities among them, the reuse rate is far less than that in synchronous mode.

2) The Type A and B incompatibilities in Figure 3-3 are caused by the sequential relationship between activities, while Type D and D* incompatibilities are caused by a parallel relationship between send activities. Intuitively, when more processes are involved, the parallel
relationship between activities occurs more often. Therefore, Type D* is more frequent than other incompatibilities, which also contributes to the larger size of asynchronous adapters.

8.4.2 Optimizing the adapter for message size

In this test, we assign size to messages in the test cases to check how this affects the choice of the messages to be adapted. Hence, the generated adapter is optimal in terms of total size of adapted messages, instead of the process complexity metrics or total number of adapted messages used in Section 8.4.1.

In this test, we use the same 4 sets of test cases as before. However, now we randomly assign a size from a specific range, e.g. [1, 50] bytes, to each message in a test case. To show how optimizing for message size affects the optimal adapter solution, we run the experiment for two versions of the MIG-IP algorithm, i.e. one in which the objective function minimizes the total size of adapted messages (a.k.a. MIG-IP1) and another where it simply minimizes the total number of adapted messages (a.k.a. MIG-IP2). Then, we compare the total size of adapted messages in these two approaches. To reduce the variance in random size assignment, we generate 100 random size assignments in a same size range for each test case in a specific set (e.g. m10, the set with 10 messages) and average the total sizes of adapted messages in all 100 assignments. Furthermore, we repeat this experiment for 10 different message size ranges (in bytes), i.e. [1, 50], [1, 100], [1, 150], [1, 200], [1, 250], [1, 300], [1, 350], [1, 400], [1, 450], [1, 500], and check how the increase in size range affects the increase in total size of adapted messages.

Figure 8-3 (a) shows the result for the test case set m10. As we expect, the total size of adapted messages in MIG-IP2 increases at a same rate with the upper bound of size range, while the total size of adapted messages in MIG-IP1 increases at a lower rate. Furthermore, the difference between two approaches increases with the upper bound of the size range. Figure 8-3
(b) shows the differences between the two approaches in all 4 sets of test cases varying number of involved messages, i.e. m5, m10, m20 and m50. It indicates that when the number and the size of messages in a test case increase, the advantage of MIP-IP1 over MIG-IP2 becomes more significant.

![Graphs showing the differences between MIP-IP1 and MIG-IP2](image)

Figure 8.3. Optimizing the adapter for message size

8.4.3 Adding security and privacy constraints

In this test, we added security and privacy constraints to our test cases to check how these constraints affect the size of the optimal adapter. As we show in Section 4.2.3, the only difference between a security constraint and a privacy constraint is the number of messages involved. Hence, for simplicity we use the term security constraint to refer to both these two types of constraints.

A security profile is a set in which each element represents a security constraint. For instance, a security profile \{\{m_1, m_2\}, \{m_3, m_5, m_9\}\} means that there are two constraints, i.e. \(m_1 + m_2 < 2\) and \(m_3 + m_5 + m_9 < 3\) (here we let \(n_{\text{max}} = n\)). Each security profile has two important properties, i.e. the number of constraints and average number of messages. These two properties characterize security profiles into three general groups of flexibility, i.e. low, medium, and high, as shown in Figure 8.4. Intuitively, when a security profile contains more constraints, it places a
higher degree of restriction on the message selection of the adapter creation. However, when a constraint contains more messages, it offers more flexibility for the selection of adaptable message. To simplify the experiment, we let all the constraints of a security profile contain the same number of messages.

In this test, we use the same sets of test cases as before. When flexibility is low (e.g. large number of constraints containing a small number of messages and), it is possible that no adapter can be found. For each set, we check the adaptability ratio, i.e. the percentage of adaptable cases in all 1000 test cases, for different security profiles. Since the restriction level of a security profile changes with both the number of constraints and messages, we generate two groups of security profile settings. In the first group, we fix the number of messages at 3, and vary the number of constraints from 1 to 10. In the second group, we fix the number of constraints at 5, and vary the number of messages in each constraint from 1 to 5. At each setting of security profile, e.g. 4 constraints and 3 messages per constraint, we generate 100 random security profiles. For each profile, we use the MIG-IP algorithm to check whether we can find an adapter for each case of a test set, and then calculate corresponding adaptable ratio. Then, we compute the average of all 100 adaptability ratios as the adaptability ratio for this set of test cases for this security profile setting.
Figure 8-4. Groups of security profile by flexibility

Figure 8-5 shows the results for the two groups of security profile settings. We observe that the adaptability ratio increases near-linearly with the increasing restrictions imposed on the security profiles. Moreover, the ratio is more sensitive to the number of messages per constraint than to the number of constraints.

![Figure 8-5. Adding security and privacy constraints](image)

8.5 Summary

In this chapter, we presented a Java-based prototype system, in which we implemented the compatibility verification and adapter creation algorithms discussed in Chapter 2-7. This
system parses process information defined in the WS-BPEL language, and generates a compatibility diagnosis report with an adapter process if necessary. We also discussed the implementation of the test case generator (TCG) and the process execution simulator (PES), based on which we conducted the validation test and performance experiments. The validation test shows the correctness of the compatibility verification and adapter creation algorithms in synchronous communication. And, the three different experiments evaluate our solution from different aspects, the results of which show that:

1. The MIG algorithms outperform PBA-MIN and other existing solutions in literature in a wide margin. Compared with the SET algorithm, the MIG algorithms decrease the size of generated adapters by more than 70% on average.

2. The adaptation in asynchronous communication is much more complicated than in synchronous communication.

3. The MIG-IP algorithm is agile to handle both message size and security constraints in business process composition.

In the next chapter, we give a comprehensive literature review in related areas.
Chapter 9

Literature Review

In literature, the research of process adaptation is still very limited. In this section, we first present several closely related approaches for control flow adapter creation. Then, we introduce existing work in message adaptation. A related research area to our study is the formal verification of Web service composition, in which there is considerable work with different directions and focuses. To clarify their relation with our work, in section 9.3 we give a brief summary of the formal verification works in the literature of Web services composition, and identify our interests. In section 9.3.3, we introduce three mainstreams on the formal study of Web services composition, which is adapted based on three recent surveys [Seguel et al. 2008; ter Beek et al. 2007; van Breugel and Koshkina 2006] and extended with several recent works. In section 9.3.4, we compare these models and summarize their pros and cons.

9.1 Related Works in Control Flow Adaptation

The research for automatic behavioral adaptation was started by Yellin and Strom in their seminal article [Yellin and Strom 1997]. They introduced formally the notion of adapter as a software entity capable of enabling the interoperation of two components with mismatching behavior. They used finite state machines to specify component interaction, to define a relation of compatibility, and to address the task of semi-automatic adapter construction. However, their approach is limited to finite state automaton, and it does not allow parallelism.
A closely related work to ours is that of Brogi and Popescu [Brogi and Popescu 2006]. There the authors have given an approach to create adapters based on service execution trees (SET). Their adaptation methodology includes the following steps,

1) Service Translation: translate BPEL descriptions into YAWL workflows.
2) Adapter Generation: generate SET, create their duals, and merge into an adapter workflow.
3) Lock Analysis: detect if there is lock in YAWL-based aggregation.
4) Adapter Deployment: convert YAWL to BPEL.

SET is similar to Label Transition System (LST) or Tree of Execution Path. The only difference is that in SET a node may consist of more than one activity, which can be concurrently executed. However, it may lead to problems. For instance, suppose a node \( n \) in a SET \( T_a \) consists of two activities \( a_1 \) and \( a_2 \) in parallel. And, activity \( a_1 \) sends a message to activity \( a_3 \) in another SET \( T_b \). Meanwhile, activity \( a_2 \) receives a message from activity \( a_4 \) in \( T_b \). Let \( a_3 \) and \( a_4 \) are in a sequence pattern in \( T_b \). Such a setting is compatible if we do not merge \( a_1 \) and \( a_2 \) to \( n \) in \( T_a \). But the merging leads to a cycle if we check \( T_a \) with \( T_b \). The adapter generation algorithm in this paper cannot solve this kind of problems.

Moreover, they do not provide a full algorithm for the adapter creation. And, their generated adapter is exhaustive because it buffers every send activity; they do not consider all basic patterns; and, the approach leads to a combinatorial explosion with choice patterns because a service execution tree is generated for every path that can be taken.

Another relevant work is [Seguel et al. 2009]. The authors treat business protocols as synthesis of fundamental patterns, such as SEQ, AND, XOR, and LOOP. They analyze the compatibility of Choice and Loop patterns based on the matching of their branches, which is similar to our previous work [Kumar and Shan 2008]. They also perform a pair-wise analysis, which may not be correct in N-process cases as shown in section 4.1. However, they have a rather
trivial element selection procedure which ignores the interference among different communication pairs, and it may lead to redundancy and thus a non-minimal adapter. Therefore, our approach dominates that of [Seguel, et al. 2009] and hence, we do not include that algorithm in our performance evaluation.

In [Benatallah, et al. 2005; Motahari Nezhad, et al. 2007], Motahari Nezhad et al provide semi-automated support for the identification of protocol-level mismatches. After pairing activities via a probability-based XML schema matching method, their approach generates a tree for all mismatches that result in a deadlock and suggesting some hints for assisting the designer in the manual implementation of the actual adapter. Since it translates all process patterns into a finite state machine, this approach increases the computing complexity of the process composition. While our work focuses on creating an optimal adapter, their paper aims at providing a diagnosis mechanism for users to fix the deadlocks in a semi-automated way. Therefore, these two approaches are complementary to each other.

In literature, there are many approaches focusing on the verification of web services composition. Generally they may be classified as: Logic based, Petri-net based, Pi-calculus based and State machine based. We give a comprehensive summary for those related approaches in Section 9.3.

9.2 Related Works in Message Adaptation

There is limited research involving integration of both control flow and message aspects in business process adaptation. The approach in [Motahari Nezhad, et al. 2007] is quite comprehensive in that it focuses on both message interface mapping and control flow mediation. However, it separates these two aspects of adaptation into two independent steps. Its message
mapping step is limited to a 1-to-1 XML schema matching. Therefore, strictly speaking this approach does not include the necessary functions for message adaptation.

In [Kongdenfha, et al. 2009], the authors propose an aspect-oriented framework to solve the mismatch at runtime. The patterns introduced in this work involve the message adaption such as one-to-many and many-to-one cases, which are similar to some of our basic patterns. Our effort is complementary; however, our patterns are more comprehensive (including the List-to-

List pattern, etc.) and extendible (e.g. by concatenation, nesting, and generating on the fly). Furthermore, the query-advice approach in [Kongdenfha, et al. 2009] separates service adaption into individual pieces, which makes it difficult to verify overall compatibility for two processes. In implementation, it requires an aspect-oriented programming extension to existing BPEL engines, while our approach does not have this constraint.

In [Li et al. 2008], the authors propose a pattern-based approach for the control flow adaption. It includes some simple message adapters for splitting and merging. But they do not address the issues of integration fully.

In [Wang, et al. 2008], the authors introduce the concept of a runtime service adaption machine that reconciles pairs of incompatible services by intercepting, transforming and forwarding messages according to a set of mapping rules. The adaptation machine masks and resolves incompatibilities by enacting the role of a compatible partner for each of the interacting services. The mapping rules have the form of production rules. Each rule describes a transformation between one or multiple source message types and a target message type. The adaptation machine fires these rules in a chained manner to produce messages expected by the interacting services based on previously intercepted messages. Moreover, the mapping rules are only fired if their output is required. By tracking the state of each interacting service (with respect to their behavioral interface) the adaptation machine is also capable of detecting two types of undesirable scenarios:
• **Deadlock**: At least one of the interacting services expects a message but none of the services can produce any further messages.

• **Information loss**: A message sent by a service is not consumed in any form by the intended recipient.

This paper also provides an operational semantics of the adaptation machine, including backward chaining algorithms for computing rule firing sequences and conditions for detecting deadlocks and information loss at runtime. Compared with our work, this approach has four major limitations,

1) It is a runtime, instead of design time, solution. Therefore, it is impossible to determine whether or not a given set of mapping rules is sufficient to fully reconcile differences between a pair of services.

2) The mapping rules in this paper are manually designed. In contrast, the message mismatches are automatically detected in our pattern-based approach.

3) The selection algorithm of mapping rules works in a traverse way. Whenever a message is received, the algorithm needs to check all rules to build the firing chains. It is not efficient when the rule set is large and many processes are running.

4) In their approach, the explicit specification of the adapter process is missing. It makes it difficult to verify the correctness of the process composition at design stage.

Another work related to message adaption is [Zhou et al. 2008; Zhou et al. 2010]. It generates an abstract service protocol from a service protocol (business process) by applying a set of rules. Control structures in a service protocol prescribe sequencing constraints between activities. These sequencing constraints specify expected message exchange orders. Through taking dependencies into consideration, the authors propose a set of rules for relaxing sequencing constraints between activities, and hence, generate an abstract service protocol from a service protocol. When dealing with large case, this transformation has the state explosion problem. Then
the approach generates abstract instance subgraphs from abstract service protocol, which is same as the approaches in [Eshuis and Kumar 2010; Sadiq and Orlowska 2000]. After that, it checks the adaptability between abstract instance subgraphs using Space-based Process Mediator (SPM), in which they use a mediation service to store the exchanged messages, and generate default messages if necessary. A number of mismatched patterns are also discussed. This part is similar to [Wang, et al. 2008]. Based on abstract service protocols, they further construct a matrix, called adaptation matrix, using an adapted depth-first search with backtracking technique. Based on this matrix, they compute an adaptation degree and identify the conditions, in which they check the pair-wise adaptability for all abstract instance subgraphs, and calculate the percentage of adaptable ones as the adaptation degree. This approach uses the checkpoint and backtracking to save the duplication in checking pair-wise adaptability, which is similar to the idea of the pattern-based approach introduced in Chapter 2.

Interface adaptation is closely related to process adaptation. An interesting approach to interface adaptation is given in [Dumas, et al. 2006]. It is based on new operators like flow, gather, scatter, collapse, burst and hide. The notion of service views as means for adapting a web service is discussed in [Fuchs 2004]. Service views are implemented using a scripting language. The semantics based methods such as OWL-S [W3C Member 2004], WSMO [Stollberg 2005] and Meteor-S [Patil et al. 2004] capture the semantics of web services and also use concept ontologies. A formal approach for reasoning about interaction patterns is given in [Busi et al. 2006; Gorrieri et al. 2005]. Our work is complementary to these efforts since both kinds of adaptation are necessary.
9.3 Summary of Formal Modeling Works in Web Services Composition Literature

In literature, there are many approaches focusing on the verification of web services composition. Generally they may be classified as: Logic based, Petri-net based, Pi-calculus based and State machine based. Besides the verification power inherited from used formal methods, those verification approaches differ in the following aspects related to business process composition.

- Formal Method: which formal method underlines the approach?
- Composition Method: how to connect multiple processes using the formal model?
- Communication Model: are synchronous, asynchronous or both communication models supported?
- Correctness Notation: what is the definition for composition compatibility?
- Proof Method: how to prove the correctness of the method?

Table 9-1 summarizes these approaches in terms of their differences in formal method, composition method, communication mode, correctness notation, and proof method. We also list our approach in the end for comparison. As it shows, most approaches take a synchronous semantics. Only in Kazhamiakin’s work [Kazhamiakin and Pistore 2005] heterogeneous communication models are studied in a systematic way. But they have to check one composition against each communication model to detect which communication model is adequate. Such an approach is not efficient. Most works inherit verification algorithms or tools from their adopted formal methods. The correctness of those algorithm or tools has been proved in their respective literature. Some of these approaches lead to very large state spaces and can be computationally expensive. To clarify their relation with our work, in this section we give a comprehensive summary of related verification work in the process composition research.
Table 9-1. Comparison of the verification approaches for process composition

<table>
<thead>
<tr>
<th></th>
<th>Formal Method</th>
<th>Composition Method</th>
<th>Communication Model</th>
<th>Compatibility Notation</th>
<th>Correctness Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salaun [Salaün et al. 2004]</td>
<td>CCS</td>
<td>Parallel operator</td>
<td>sync</td>
<td>Behavior equivalence</td>
<td>Inherited</td>
</tr>
<tr>
<td>Foster [Foster et al. 2004]</td>
<td>FSP</td>
<td>FSP monitor</td>
<td>sync</td>
<td>Safety and Liveness</td>
<td>Inherited</td>
</tr>
<tr>
<td>Fu [Fu et al. 2005]</td>
<td>PROMELA</td>
<td>Conversation</td>
<td>async</td>
<td>LTL properties</td>
<td>Inherited</td>
</tr>
<tr>
<td>Martens [Martens et al. 2006]</td>
<td>Petri Nets</td>
<td>Place merge</td>
<td>async</td>
<td>Weak-soundness</td>
<td>Inherited</td>
</tr>
<tr>
<td>Haddad [Haddad et al. 2004]</td>
<td>LTS</td>
<td>N/A</td>
<td>sync</td>
<td>Termination</td>
<td>N/A</td>
</tr>
<tr>
<td>Pistore [Pistore et al. 2005]</td>
<td>STS</td>
<td>Parallel product</td>
<td>sync</td>
<td>Deadlock-free &amp; EaGLe properties</td>
<td>Inherited</td>
</tr>
<tr>
<td>Kazhamiakin and Pistore 2005</td>
<td>STS</td>
<td>CSTS</td>
<td>sync, async</td>
<td>Termination &amp; LTL properties</td>
<td>Inherited</td>
</tr>
<tr>
<td>Bordeaux [Bordeaux et al. 2005]</td>
<td>LTS</td>
<td>N/A</td>
<td>Sync</td>
<td>Behavior equivalence</td>
<td>N/A</td>
</tr>
<tr>
<td>Shan [This work]</td>
<td>AND/XOR graph</td>
<td>MIG</td>
<td>sync, async</td>
<td>Soundness</td>
<td>Proved</td>
</tr>
</tbody>
</table>

9.3.1 Motivation

Services are typically designed to interact with other services to form larger applications. It raises a number of challenges, however, one of them being the challenge to guarantee the correct interaction of independent, communicating pieces of software. Due to the message-
passing nature of service interaction, many subtle errors might occur when several services are put together (unreceived messages, deadlocks, incompatible behaviors, etc.). These problems are well known and recurrent in distributed applications, but they become even more critical in the world of service-oriented computing that is ruled by the long-term vision of “services used by services”, rather than by humans, and in which interactions should ideally be as transparent and automatic as possible. A major problem of the approaches, viz. the lack of software tools to verify the correctness of service compositions, is at the same time the main advantage of most formal methods. In particular, formal methods and tools can be used to decide whether services:

1) are in some precise sense equivalent and

2) satisfy certain desirable properties.

If one should discover that a service composition does not match an abstract specification of what is desired, or that a main property is violated, this can be of help to correct a design or to diagnose bugs in a service [ter Beek, et al. 2007].

### 9.3.2 A taxonomy of research concepts for services composition verification

A main feature of services is the reuse mechanism to build new applications, which often need to be defined out of finer-grained subtasks that are likely available as services again. Composition rules describe how to compose coherent global services. In particular, they specify the order in which, and the conditions under which, services may be invoked. We distinguish syntactic (XML-based) and semantic (ontology-based) service composition [ter Beek, et al. 2007]. Our study focuses on the syntactic aspect of the service composition.

In the syntactic category, there are two main approaches. The first approach, referred to as service orchestration, combines available services by adding a coordinator (the orchestrator)
that is responsible for invoking and combining the single subactivities. The second approach, referred to as service choreography, instead does not assume a coordinator but rather defines complex tasks via the definition of the conversation that should be undertaken by each participant; the overall process is then achieved as the composition of peer-to-peer interactions among the collaborating services [ter Beek, et al. 2007]. While BPEL and WS-CDL are widely regarded as orchestration languages to enable the coordination and composition of a set of services, the composition of multiple BPEL processes is a choreography problem, which is the focus of our study.

Any service composition approach should aim to support some characteristics to illustrate the success of composition. In [ter Beek, et al. 2007], the authors identify the following measures of interest,

a) Connectivity: Reliable connectivity is needed to reason about service interactions before composition, in order to guarantee the continuity of service delivery after composition. A successful composition should be able to correctly deliver messages, promptly respond to service requests, and deliberately handle exception and errors.

b) Correctness: Service composition may lead to large, complex systems of concurrently executing services. An important aspect of such systems is the correctness of their (temporal) behavior. The behavioral properties that a service should satisfy are usually defined by a specification that precisely documents the desired behavior. Formal methods then provide rigorous mathematical means to guarantee a system's conformance to a specification. Generally, the correctness properties of a communication system are classified into two categories, i.e. safety property and liveness property. Safety properties are assertions that some undesired event never happens in the course of a computation, while liveness properties assert that some event does eventually happen. By verifying
such properties, one obtains measures of correctness of a service (composition). Besides, security and trust are another aspect which a service composition needs to provide.

c) Quality of Services: The quality of service is determined by the following measures, such as accuracy, availability and performance.

In our study we focus on the Correctness characteristic of web services composition, since none of the industrial approaches offer any direct support for the verification of service compositions at design time to evaluate its correctness.

The formal analysis of process composition can be categorized into following types [Benatallah et al. 2006],

- **Compatibility analysis** refers to assessing if two services, characterized by specific protocols, can interoperate, and which conversations are and are not possible. Compatibility analysis is necessary for static and dynamic binding, and it also helps in evolution since it helps verifying that a modified client can still interact as desired with a certain service.

- **Replaceability analysis** is concerned with verifying whether two protocols (e.g., of two service providers) are equivalent, that is, if they can support the same set of conversations. This means that they can interact with the same clients. Replaceability analysis also involves finding the set of conversations that both services can support if they are not equivalent. Replaceability can be further extended to four sub-classes: equivalence, subsumption, replaceability with respect to a client protocol and replaceability with respect to an interaction role. These replaceability classes provide basic building blocks for analyzing the commonalities and differences between service protocols. Readers may refer to [Benatallah, et al. 2006; Ryu et al. 2008] for more details.
• **Consistency analysis** studies whether the different specifications of a service are consistent with each other.

• **Compliance analysis** is to verify if an implementation of a service is conformant to the specifications.

In our study, we focus on the compatibility analysis for two or more processes.

In summary, our study focuses on the syntactic service composition in a choreograph approach, in which each service has a complicated behavior characterized by a process or protocol. In our scenario, there is no explicit specification for public conversation between multiple services. Therefore, we target on compatibility analysis, in which we investigate the correctness of the composition, e.g. free-of-deadlock (safety property) and termination (liveness property). In next section, we survey the literature with these focuses.

### 9.3.3 Formal methods for Web services composition

Recently several formal methods, most of them with a semantics based on transition systems (e.g. automata, Petri nets, process algebras), have been used to guarantee correct service compositions. Below we first present a selected overview of the use of well-known languages and models by the formal methods community. We are interested in particular in those techniques that are applied to the verification of BPEL compositions.

#### 9.3.3.1 Process Algebras

Like Petri nets, process algebras are precise and well-studied formalisms that allow the automatic verification of both functional and non-functional properties. They come with a rich
theory of bisimulation analysis to establish whether two processes have equivalent behaviors. Such analyses are useful to establish whether one service can substitute another service in a service composition or to verify the redundancy of a service [ter Beek, et al. 2007].

In [Salaün, et al. 2004], the authors use Process Algebra as an abstract representation means to describe, compose and verify services. Their approach does not focus on a specific language, but on a whole family of formal description techniques. They argue all the theoretical foundations underlying PA are adequate to describe and reason on services, especially to ensure correct composition. They present a two-way mapping between process-algebraic descriptions and concrete implementation descriptions; especially illustrated with CCS and BPEL4WS, a previous version of WS-BPEL. They use the CWB-NC (The Concurrency Workbench of the New Century) [Stony Brook University 2000] software to perform on-the-fly compatibility checks for choreography.

In [Salaün, et al. 2004], no automatic translation is introduced between BPEL and CCS. A choreography situation is directly modeled in the input language of the CWB-NC, which is an ASCII notation for CCS. First, all processes are put in a closed system. They specify communications in a binary and synchronous communication model. The basic verification tools provided by CWB-NC can be used to find deadlocks, missing synchronizations, and incomplete behaviors. Second, it can certify that the composition of service processes can match the client process, i.e. the system is equivalent with the process obtained by reversing emissions and receptions in the client process. Trace equivalence and observational equivalence are discussed. The properties written in CTL can also be proved using CWB-NC.

In [Magee and Kramer 2006], the authors present a process algebra named FSP (Finite State Process). Each FSP represents a finite Labeled Transition System (LTS). Kramer and Magee also present a tool for FSP named Labeled Transition System Analyzer (LTSA). This tool takes as input an FSP and translates it into a labeled transition system. Subsequently, this labeled
transition is analyzed. LTSA can check for safety and progress properties as well as properties expressed in the logic LTL. In [Foster et al. 2003], an extension of LTSA is developed to verify BPEL processes. This tool is named LTSA-WS. A key component of the tool is the translation of the activities of BPEL into FSPs. They show how LTSA can be exploited to check if a web service composition implemented in BPEL satisfies a web service composition specification captured by Message Sequence Charts (MSCs). Both the BPEL process and the MSC are translated into FSPs.

In [Foster, et al. 2004], the authors use LTSA to check compatibility of web service compositions in BPEL. They model the partner communication using the synchronous message passing model described in [Magee and Kramer 2006]. All processes are composed into a LTS. Partner process activity interactions can be represented in FSP by using the notion of a connector, which is implemented in FSP as a monitor. The task of modeling the invocation process and port is completed by using the re-labeling feature of FSP linking the appropriate activities between process and port. Using the LTSA tool, they compile these connectors as FSP models into a complete LTS, and then do the assessment of correctness.

Three levels of compatibility for component compositions have been reported: 1) Interface Compatibility – focusing on semantics of correlating invocations against receiving and message replies between partner processes. 2) Safety Compatibility – assurance that the composition is deadlock free and is checked against partial correctness of transitions. 3) Liveness Compatibility – assurance against starvation of progress (that the service process eventually terminates) and that messages received are served on a first-come-first-served basis. Safety Compatibility focuses on performing safety analysis of composition models, and determining if there exist any deadlocks or other properties specified by the analyst. Liveness Compatibility focuses on performing progress analysis of composition models, and determining if the
composition exhibits behavior which does not meet requirements of success. These properties can be specified as a special formula described in [Foster, et al. 2004].

9.3.3.2 PROMELA and SPIN

PROMELA (Process or Protocol Meta Language) is a model verification language. The language allows for the dynamic creation of concurrent processes to model, for example, distributed systems. In PROMELA models, communication via message channels can be defined to be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered). PROMELA models can be analyzed with the SPIN model checker [Bell Labs 2009], to verify that the modeled system produces the desired behavior, which is specified in Linear Temporal Logic (LTL). [Wikipedia 2009]

In [Fu et al. 2004a; Fu, et al. 2005], the authors present a framework to verify properties of a web service composition consisting of multiple BPEL processes that communicate asynchronously. Each BPEL process is translated to a guarded automaton. Subsequently, these guarded automata are mapped to PROMELA. The combination of these translations provide a map from (a part of) BPEL to PROMELA. Such a two step approach allows for the support of other languages than BPEL and PROMELA in the future. Each process is equipped with a FIFO queue. The composition of multiple processes is captured by the notion of conversation, which model the interactions among processes in a Web service composition. A configuration is defined as the combination of the local state of each process and the contents of each queue. A complete run is a sequence of configurations with initial and final configurations. A conversation is the send sequence generated by a complete run. The authors do not define the correctness notion for a Web service composition. Instead, they focus on the checking of LTL properties.
The semantics of LTL formulas can be naturally adapted to conversations by defining the set of atomic propositions as the power set of messages. However, verification of LTL properties in conversations of a Web service composition is an undecidable problem, since the asynchronous communication leads to state space explosion. [Fu et al. 2004b]. To tackle this problem, the authors present a technique called **synchronizability analysis**, which identifies Web service composition that generate the same conversation set with synchronous and asynchronous communication semantics. Via this technique, we can verify synchronizable Web service compositions using the synchronous communication semantics and the verification results we obtain will hold for the asynchronous communication semantics. Petri Nets

Petri nets are a place-transition network to model concurrent systems. Their main attraction is the natural way of identifying basic aspects of concurrent systems, both mathematically and conceptually. This has contributed greatly to the development of a rich theory of concurrent systems based on Petri nets. Moreover, their ease of conceptual modeling (largely due to an easy-to-understand graphical notation) has made Petri nets the model of choice in many applications. In fact, Petri nets are very popular in BPM-related fields due to the large variety of process control flows they can capture. In particular, the dead-path-elimination technique that is used in BPEL to bypass activities whose preconditions are not met, can be readily modeled in Petri nets [ter Beek, et al. 2007]. Mapping BPEL process to a Petri net allows the verification techniques and tools developed for Petri nets to be exploited in the context of BPEL processes [van Breugel and Koshkina 2006].

In several related papers [Martens 2005a; Martens 2005b; Martens and Moser 2006; Martens, et al. 2006], the authors discuss the compatibility problem of BPEL processes based on a Petri net model. They omit the details of the mapping between BPEL and Petri nets by adopting an existing Petri net semantics of BPEL from [Hinz et al. 2005]. The composition of BPNs (BPEL-annotated Petri net) is achieved by merging each input place (or output, resp.) of the
provided interface of one BPEL process and corresponding output place (or input, resp.) of the required interface of another BPEL process. The authors introduce several criteria for business processes and their compositions.

A BPEL process is called *usable* (or *controllable*) if there exists an environment (another BPEL process) with which the process can interact such that the process terminates properly. Two BPEL processes are called *compatible* if their composition is usable. A BPEL is said to *simulate* another BPEL process if each environment that makes the latter usable makes the former usable as well. Two BPEL processes are called *equivalent* (or *consistent*) if the one simulates the other and vice versa. The correctness of the composition is based on the deadlock-free property.

If a composed system is close (i.e. the composed BPN model has an empty interface), the notion of soundness or weak-soundness is applicable. Soundness is a well-established quality criterion for workflow nets that basically guarantees deadlock-freeness. Two BPN models are compatible if the composed system is weak-soundness (empty interface) or controllable (non-empty interface).

A. If the composition of two BPEL processes yields a BPN with a non-empty interface, behavioral compatibility can be verified by checking the controllability of the composed BPN. In [Martens 2005a], a decision algorithm for controllability is presented based on the *communication graph* (c-graph) of the BPN model. Once the c-graph is generated, the algorithm cuts off all parts that yield to a deadlock. If the remaining graph is not empty, the given BPN is proven to be controllable.

B. If the composed BPN has a non-empty interface, the task is to verify weak-soundness of a workflow net. The authors propose to use model checking method and LoLA [Schmidt 2000] to verify this property based on *reachability graph*. Since no technical details are given for the reachability graph, it is hard to tell whether it takes a synchronous or asynchronous communication model.
Based on my observation, it is more like an asynchronous model in which one buffer is equipped for each message, because there is a place between a send transition and a receive transition, and the message/token may stay in this place without immediate firing.

9.3.3.3 Automata, LTS, STS, FSM

Automata, Labeled Transition System (LTS), State Transition System (STS) and Finite State Machine (FSM) are well-known models underlying formal system specifications. The intuitive way in which automata can model system behavior has lead to several automata-based specification models, like (variants of) I/O automata, timed automata and team automata. Their formal basis allows automatic tool support and as a result automata-based models are increasingly being used to formally describe, compose, and verify service compositions [ter Beek, et al. 2007].

In [Haddad, et al. 2004], the authors argue that the syntactic features of XLANG, one of BPEL's predecessors, make it a good candidate to be an algebra of timed processes. Then, they model some of the activities of XLANG by means of Labeled Transition System (LTS). The transitions capture the passing of time in a discrete way. Furthermore, Haddad et al. define what is a correct interaction between two LTSs, which includes the following rules, a) if one LTS is able to send a message, the other one must be able to receive this message; b) if one LTS is able to let the time pass, the other one must also be able to let the time pass; c) if one LTS is terminating, the other one must also be able to terminate; d) if one LTS expects the reception of a message, the other one may not able to send this message, but it is able to send this message in a other state.

Based on this relation, an algorithm is developed to produce the LTS of the client behavior if such a behavior exists or detecting the ambiguity of the Web service. Their goal is to produce a client behavior which correctly interacts with the service. This is very similar to the
idea of controllability discussed in [Martens, et al. 2006]. There is no composition operator
defined in this paper. From my point of view, they take a synchronous communication model. In
[Haddad et al. 2006], the authors extend their previous results from discrete time to real time.
Instead of LTS, timed automata are used to model the BPEL4WS activities.

In [Pistore, et al. 2005], the authors present a number of tools for BPEL. The tool
BPEL2STS translates an (abstract) BPEL process to a State Transition System (STS). A number
of these state transition systems, representing BPEL processes, are composed in parallel using a
parallel product for STSs. They assume that interactions among Web services are synchronous.
The resulting parallel composition is also represented by a STS. The tool MBP [Bertoli et al.
2001] takes as input such a parallel composition and a requirement, the latter being formalized in
EaGLc [Lago et al. 2002], which is similar to CTL operators, but their semantics take into
account the notion of preference and the handling of failure when subgoals cannot be achieved.
As output, MBP produces a state transition system such that this system in parallel with the input
system satisfies the input specification, which is called Controller system. The controlled system
is guaranteed to be deadlock-free. The tool STS2BPEL translates a state transition system to a
BPEL process. Combined these tools can synthesize web service compositions expressed in

BPEL.

In [Benatallah, et al. 2006], a service business protocol is modeled as a Finite State
Machine (FSM). They use a tree-based, instead of path-based, semantics of protocols in order to
define a general framework. Based on this protocol semantics, the authors define notions of
execution tree and conversation tree (execution tree without state label) for one single protocol,
and then interaction tree for two interacting protocols. The states in interaction tree are the
product of states in both protocols, which is based synchronous communication semantics.
Correct interactions between two protocols are captured by using the notion of complete
interaction trees. The compatible composition operator takes as input two business protocols and
returns a protocol, called a compatible composition protocol, which describes the set of complete interactions trees between the input protocols.

Then, an algorithm, called C-composition, implements this operator. If the result of a compatible composition of two protocols is empty, this means that no conversation is possible between two services that support these protocols. Otherwise, the result is the identification of possible interactions between these protocols. The correctness properties of the algorithm such as termination, soundness, and completeness are proved. The performance of that algorithm is polynomial. They further claim that A) a protocol P1 is partially compatible with a protocol P2 if there is at least one possible conversation that can take place among two services supporting these protocols, while protocol P1 is fully compatible with P2 if all the executions of P1 can interoperate with P2.

In [Kazhamiakin and Pistore 2005; Kazhamiakin et al. 2006], the authors focus on three communication models of business processes: synchronous, ordered asynchronous, and unordered asynchronous. Given a number of communicating BPEL processes, each process is transformed into a State Transition System (STS) and subsequently these systems are composed in parallel, resulting in yet another State Transition System with Channels (CSTS). A global state of CSTS contains two components: a control state that represents the local states of the participating STSs, and a queue content that defines set of messages stored in the queues in a particular moment of time.

Given the same set of STSs, different configurations may be used to represent their composition. These configurations are parametric with respect to the number of queues, to the distribution of the queue alphabets, and to the queue bounds. Such configurations are denoted as communication models. The CSTS is complete if all the terminating global states have empty queue content. The resulting system can be fed into the NuSMV tool [Cimatti et al. 2002] to check the validity of the system with respect to a given communication model.
Furthermore, NuSMV can also be exploited to verify properties of the system expressed as LTL formulas. Based on the notion of CSTS, a hierarchy of communication models is defined by changing the queue model. The relations, i.e. simulate and bisimilar, among communication models are defined. A communication model is adequate if it expresses all the behaviors of the most general model. To analyze compositions of Web services, they incrementally pass through the models starting from the synchronizable until the lease general adequate model is found for the given composition scenario. The composition is checked for completeness (i.e., the queues are empty in the terminal states of the composition) and bounded growth in queues.

In [Bordeaux, et al. 2005], the authors use a LTS notion, and they adopt a synchronous two-party communication model. They introduce three candidate definitions of compatibility.

Compatibility 1: Two services A and B are compatible if they have opposite behaviors. In other words, A and B share a same process structure, but have opposite actions, i.e. sends are changed to receives and vice-versa.

Compatibility 2: Two web services are compatible if they have no unspecified reception. Compatibility here means that one process will not send requests which the other process cannot satisfy.

Compatibility 3: Two services are compatible if the initial state is not a deadlock, i.e., if there is at least one execution leading to a pair of final states.

All these compatibilities are defined based on a simple behavioral representation - sequence of actions. They argue that these three compatibilities can be applied in different situations. For instance, if we want two services to work together without unspecified receptions and without deadlock, in which case we can use definition 2 with an additional test for deadlock-freeness to guarantee that the two services will communicate successfully. If a stricter matching of the behaviors is desired, definition 1 should be considered, and if what is needed is simply to
see whether an interaction shall be possible, one can use definition 3. Since this paper focuses on the formal definitions of the compatibility notions, supporting tools or techniques are not discussed.

9.3.4 Comparison of verification methods

In this section, we compare the verification approaches introduced in Section 9.3.3 to show their pros and cons. We present this comparison in two parts. In first part, we compare the different aspects of formal methods used in these approaches in term of their verification power in a general sense. In second part, we compare the existing approaches in terms of the verification requirements for service composition. This two-fold comparison shows a comprehensive picture of verification capability for these approaches.

9.3.4.1 Comparison of formal methods

According to [Yang and Ward 2003], a formal method should consist of some essential components: a semantic model, a specification language (notation), a verification system and refinement calculus, development guidelines, and supporting tools. Formal methods can be classified into the following five types: model-based, logic-based, algebraic, process algebra and net-based (graphical) methods. In the literature of Web services composition, the following two types are heavily used.

Process algebra approach: in this approach an explicit representation of concurrent processes is allowed. System behavior is represented by constraints on all allowable observable communications between processes.
Net-based approach: Graphical notations are popular notations for specifying systems as they are easier to comprehend and, hence, more accessible to non-specialists. This approach uses graphical languages with formal semantics, which brings particular advantages in system development and reengineering.

To further illustrate pros and cons of the formal methods introduced in Section 9.3.3, we summarize them in following different aspects.

- Automated tools: This criterion refers to whether the formal method has relevant automated tools to support its development, such as tools for checking syntax, verifying semantics and auto-execution.
- Reliability: This criterion refers to the reliability of the formalism.
- Proof system: This refers to whether there is any proof system and what type of proof system exit.
- Industrial strength: This criterion refers to the potential of the formal method for large-scale industrial application.
- Methods of verification: This criterion refers to the existing methods of verification of the formal method. Normally, there are two different methods of verification: model checking and theorem proving.
- Concurrency: This criterion refers to the explicit representation and reasoning of concurrency.
- Communication: This criterion refers to the explicit representation and reasoning of communication.

This comparison in Table 9-2 is modified and extended from [Yang and Ward 2003].

From Table 9-2, we can see that all these formal methods have good reliability, and most of them have automated tools to support. Net-based approaches rely more on the reachability type of proof system, while process algebra approaches use the bisimulation method. Due to state
space explosion problem, all the formal methods have limited industrial strength. Besides, the asynchronous mode of communication is not widely supported.

Table 9-2. Comparison of the formal methods

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Process algebra approach</th>
<th>Net-based approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CCS</td>
<td>FSP</td>
</tr>
<tr>
<td>Automated tools</td>
<td>None</td>
<td>Some</td>
</tr>
<tr>
<td>Reliability</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Proof system</td>
<td>Bisimulation</td>
<td>Reachability</td>
</tr>
<tr>
<td>Industrial strength</td>
<td>Some</td>
<td>Some</td>
</tr>
<tr>
<td>Methods of verification</td>
<td>Both</td>
<td>Model-checking</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Interleaved</td>
<td>Exist</td>
</tr>
<tr>
<td>Communication</td>
<td>Sync</td>
<td>Sync</td>
</tr>
</tbody>
</table>

9.4 Summary

In this chapter, we first presented related work for control flow adapter creation, and then introduce existing work in message adaptation. Furthermore, we surveyed the verification approaches for process composition in the Web services literature, with a special focus on the verification power of the formal methods. From this summary, we can see that most of these works take a general methodology as follows, a) map BPEL components to the constructs in a formal model, b) map composition correctness notation to some properties that can be captured by the existing tools for that formal model, c) use the existing tools to check these properties automatically. Such a methodology leads to significant limitations, e.g. unable to capture heterogeneous communication models, or facing the state explosion problem. Furthermore, none of these approaches can identify the characteristics of synchronous communication or
asynchronous communication directly based on the formal models. The comparison distinguishes the contributions of our solution. In the next chapter, we present the major contribution of this work, and also discuss the limitations and future work.
Chapter 10 Conclusion and Discussion

Information technology has enabled many organizations to collaborate effectively in a supply chain network. For such collaboration to be successful, it is critical that the organizations be able to connect their business processes seamlessly in an automated manner. In this dissertation, we investigated how heterogeneous Web-services-based business processes of multiple organizations can be integrated through the use of adapters in both synchronous and asynchronous communication modes. We proposed a new framework for business process adaptation, which integrates both message and control flow adaptation. It is based on the recognition that message adaptation can affect control flow and vice versa. Hence, a tight integration is required between these two aspects.

For the control flow adaptation, our algorithms are based on using a pattern compatibility matrix and matching rules to reduce the search space. Furthermore, we proposed new analysis techniques based on pair-wise message analysis to understand dependencies among messages communications. We also developed algorithms for minimal adapter creation using message interaction graphs (for synchronous communication), and message pair analysis with transitivity elimination (for asynchronous communication). Our approach generalizes to $N$ processes with any $M$ messages exchanged across them. The ability of handling both synchronous and asynchronous communication makes our techniques more significant in realistic application scenarios.

Compared with existing solutions in literature, the control flow adaptation approach proposed in this dissertation makes the following significant contributions,

1. The pattern-based approach decomposes the adaptation effort into a number of adapter creation problems in sub pattern levels. It avoids the state explosion problem in analyzing large-scale cases.
2. The MIG-IP and Async-IP algorithms generate optimal adapter in synchronous and asynchronous communication. They minimize the communication effort, and also protect privacy and security of business parties at a maximum level.

3. Our approach can handle general cases involving any number of processes and messages, which are very common in real-world business scenarios.

4. The extended MIG-IP method considers not only the number of adapted message, but also other factors, such as message size, and security and privacy constraints. This makes it possible to design a complete adaptation solution.

For message adaptation, we identified common message mismatch patterns and also showed how message adapters can be created on the fly for matching parts of multiple services where basic adapter patterns do not exist. Last, we showed how message adapters can themselves be combined to create an aggregate adapter for the complete service composition. Compared with existing work for message adapter, our integrated adaptation approach outstands itself in the following aspects,

1. We recognize the interplay between control-flow adaptation and message adaptation, and treat them in an integrated way.

2. Our approach generates an explicit adapter process covering both control flow and message adaptation, which makes it possible to verify correctness and improve performance of a process composition at design stage.

3. Our approach minimizes the adaptation efforts not only in control flow aspect but also in message transformation aspect.

4. The implementation of our solution does not require special extension or component on the existing process engine products. Therefore, it can easily put in practice in current legacy systems.
We implemented all the adapter creation algorithms in a Java-based prototype system, and validated the effectiveness of the compatibility verification and adapter creation algorithms for synchronous communication via a simulation test. Moreover, we designed three experiments to evaluate their performance in multi aspects, and discussed the insights about performance, sensitivity and robustness, and other issues pertaining to the two modes of communication in the context of process composition. The results show that,

1. The PBA-MIN algorithm outperforms the existing SET algorithm in literature in a large margin, and the MIG algorithms further improve the performance in a significant level. Compared with SET, the size of generated adapters by the MIG algorithms decreases more than 70% on average. Therefore, our adapter algorithms are very effective in creating a minimal adapter, and as a result reducing the cost in process adaption.

2. In the complicated scenarios of process-level collaboration, the queuing mechanism in asynchronous communication makes the coordination more complicated, instead of improving the interaction among multiple parties. Therefore, neither communication mode is generally superior to the other. For each collaboration case, we should choose an appropriate one based on its specific characteristics and requirements.

3. The MIG-IP algorithm is able to handle both message size and security constraints in business process composition. It is agile in the selection of adapted messages by avoiding those messages with large size or highly related to security and privacy concerns.

In Table 10-1, we summarize the findings in this dissertation for the adapter creation in synchronous versus asynchronous communication. Considering the interaction of a message pair in two processes, there are two basic types of adaptation for each communication scenario, i.e.
Type A and B for synchronous communication, and Type C and D for asynchronous communication. The adaptation types in more complicated scenarios are combined or extended from those basic types. In the literature, there are many works about compatibility verification in synchronous communication, while those about asynchronous communication are still limited, due to the restricted capability of queue modeling in existing formal models such as Petri nets, pi-calculus, etc. Moreover, for both communication modes there are a limited number of approaches for adapter creation. And, these approaches generate either a full or non-optimal adapter. In this dissertation, we propose the MIG-IP algorithm for the optimal adapter creation in synchronous communication, and the Async-IP algorithm for asynchronous communication.

In both communication modes, we discussed the impact of inter-process causal dependencies. Interestingly, they produce opposite effects. The inter-process relationship in synchronous communication helps us detect more incompatible cases across multiple processes which thus add more messages to a correct adapter. By contrast, the inter-process relationship in asynchronous communication helps us find those compatible cases across multiple processes, which were falsely reported as incompatible. Hence, this effort reduces the redundancy in the generated adapter. Since there are multiple solutions to solve Type B and B* incompatibilities, the optimal adapter creation in synchronous communication is more flexible in the message selection for adaptation; therefore the optimality of the adapter in synchronous communication depends on the number of adapted messages, and also other factors such as message size, security constraints, etc. However, the adapter creation in asynchronous communication does not have this advantage. All messages that violate the async rule must be adapted. Adding the consideration of message size and security constraints does not change the selection of those adapted messages. Finally, both communication modes are supported in existing process engine systems such as IBM WebSphere Integration Developer.
Table 10-1. Comparison of adapter creation in Sync vs. Async communication

<table>
<thead>
<tr>
<th>Feature</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Mode</td>
<td>Basic adaptability types</td>
<td>2 (Type A and B)</td>
</tr>
<tr>
<td></td>
<td>Existing body of work in</td>
<td>2 (Type C and D)</td>
</tr>
<tr>
<td></td>
<td>compatibility verification</td>
<td>large</td>
</tr>
<tr>
<td></td>
<td>Existing work in adapter creation</td>
<td>small</td>
</tr>
<tr>
<td></td>
<td>Impact of inter-process causal</td>
<td>Limited to full and non-</td>
</tr>
<tr>
<td></td>
<td>dependency</td>
<td>optimal adapters</td>
</tr>
<tr>
<td></td>
<td>Proposed algorithm for</td>
<td>limited to full and non-optimal</td>
</tr>
<tr>
<td></td>
<td>optimal adapter creation</td>
<td>adapters</td>
</tr>
<tr>
<td></td>
<td>Factors that affect the optimality</td>
<td>Number and size of adapted</td>
</tr>
<tr>
<td></td>
<td>of the adapter</td>
<td>messages, security and privacy</td>
</tr>
<tr>
<td></td>
<td>Selection flexibility of adapted</td>
<td>constraints, etc.</td>
</tr>
<tr>
<td></td>
<td>messages</td>
<td>Flexible</td>
</tr>
<tr>
<td></td>
<td>Support in existing systems</td>
<td>Yes, e.g. IBM WebSphere</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integration Developer</td>
</tr>
</tbody>
</table>

Nevertheless, several limitations of our work should be noted,

1. We assume that all processes of involved parties are well designed. In other words, the details of activities and control flow structures are specified at design time instead of execution time. However, nowadays business process management practices involve more and more dynamic features, which make the process definition details maybe not available at design stage. Therefore, a static approach for compatibility verification and adapter creation, like ours, is not capable of handling such new situations.

2. In this study we assume that collaborating parties have a same convention on message or message element naming. However, in real world different organizations may have different vocabularies to describe business documents. Therefore, the message or message element matching cannot be achieved via name information.
Instead, we need more advanced techniques, such as semantic ontology, schema matching, etc., to pair input and output data.

3. Since we do not have access to a large number of real business process examples, especially for the collaboration scenario in supply chain network, the performance experiments were conducted on the test cases without business semantics, which were randomly generated by the prototype system. As shown in Table 8-1, a large percentage of the generated cases are deadlocked. This may not reflect the real business situation. If in the future we can find an industry partner who can provide a large library for collaborative business processes, we can use real business cases to further testify the effectiveness of our approach. Furthermore, we can discover heuristics from real cases to guide the generation of more meaningful test cases.

4. In our solution, we assume that there exists a 3rd-party organization that can access the process definitions of all collaborating parties; verify their compatibility and create an adapter if necessary; initialize the deployment of all these processes; and mediate the communication among those parties during run time. Unfortunately, so far there is no standard protocol or SOA framework that can fully support those functions in a mediator such as e-market places. Therefore, we need to investigate which protocol or system is an appropriate alternative to extend by incorporating those advanced functions.

In future work, we plan to extend current approach in the following aspects.

1. We would consider the adapter creation in a mixed communication mode, i.e. some activities are communicated in synchronous mode while others in asynchronous mode. This mixture setting more accurately reflects the business requirements in real world, in which some business documents or activities need to executed in an urgent and secure way, while the uncritical others can be
handled in a more flexible way. The MIG-IP and Async-IP algorithms can be combined and extended to handle this new communication scenario.

2. We would consider data flow information in message adaptation. In this work, we assume that each message communicated is independent with others. Therefore, the delivery of messages can be rearranged in any order in adaptation. However, in real business scenario a message or document sending out may be generated based on another message received earlier from another party. Therefore, this sending message cannot be adapted before the required message is received. This kind of data flow information may add additional constraints on the message selection for adaptation.

3. We would design a straightforward metric to indicate the compatibility ratio of a process composition. This metric can help business managers to evaluate the soundness of business process operation with potential business partners, and therefore provide a decision support tool for the selection of fitting collaborators when planning a new business engagement or forming a strategic alliance. Furthermore, if we can obtain a large dataset of real process collaboration cases, we plan to mine out typical incompatibility patterns for major business operations, e.g. incompatibility patterns in payment processing, etc. The detected patterns form a knowledge base, which can help organization to improve their process design, and consequently avoid the common mismatches in business transactions.

4. We would like to develop and test our prototype system further, and also include ability to compose other control flow patterns into our algorithm for adapter creation. In this work, we only consider the general structures such as sequence, choice, parallel and loop. Advanced patterns such as inclusive-choice and
discriminator patterns are also common in industry practices. Therefore, the extension supporting those patterns will make our tool more useful in real applications.
Appendix A. Johnson's Algorithm [Johnson 1975]

In Johnson's algorithm, the time consumed between the output of two consecutive circuits as well as before the first and after the last circuits never exceeds the size of the graph, \(O(n+e)\). Elementary circuits are constructed from a root vertex \(s\) in the subgraph induced by \(s\) and vertices "larger than \(s\)" in some ordering of the vertices. Thus the output is grouped according to least vertices of the circuits.

To avoid duplicating circuits, a vertex \(v\) is blocked when it is added to some elementary path beginning in \(s\). It stays blocked as long as every path from \(v\) to \(s\) intersects the current elementary path at a vertex other than \(s\). Furthermore, a vertex does not become a root vertex for constructing elementary paths unless it is the least vertex in at least one elementary circuit.

The algorithm accepts a graph \(G\) represented by an adjacency structure \(A_G\) composed of an adjacency list \(A_G(v)\) for each \(v \in V\). The list \(A_G(v)\) contains \(u\) if and only if arc \((v, u)\) \(\in E\). The algorithm assumes that vertices are represented by integers from 1 to \(n\).

The algorithm proceeds by building elementary paths from \(s\). The vertices of the current elementary path are kept on a stack. A vertex is appended to an elementary path by a call to the procedure CIRCUIT and is deleted upon return from this call. When a vertex \(v\) is appended to a path it is blocked by setting \(\text{blocked}(v) = \text{true}\), so that \(v\) cannot be used twice on the same path. Upon return from the call which blocks \(v\), however, \(v\) is not necessarily unblocked. Unblocking is always delayed sufficiently so that any two unblockings of \(v\) are separated by either an output of a new circuit or a return to the main procedure. The complete algorithm is shown in Figure.
Circuit-finding Algorithm (Johnson's Algorithm)

```
begin
integer list array \(A_k(n), B(n);\) logical array blocked \((n);\) integer \(s;\)
logical procedure CIRCUIT (integer value \(v;\));
begi logical \(f;\)
procedure UNBLOCK (integer value \(u;\));
begin
    blocked \((u):= false;\)
    for \(w \in B(u)\) do
        begin
            delete \(w\) from \(B(u);\)
            if blocked\((w)\) then UNBLOCK\((w;\)
        end
end UNBLOCK
\(f:= false;\)
stack \(v;\)
blocked\((v):= true;\)
L1: for \(w \in A_k(v)\) do
    if \(w = s\) then
        begin
            output circuit composed of stack followed by \(s;\)
            \(f:= true;\)
        end
    else if \neg\) blocked\((w)\) then
        if CIRCUIT\((w;\)) then
            \(f:= true;\)
L2: if \(f\) then UNBLOCK\((v;\)
    else for \(w \in A_k(v)\) do
        if \(v \not\in B(w)\) then put \(v\) on \(B(w);\)
unstack \(v;\)
CIRCUIT := \(f;\)
end CIRCUIT;
empty stack;
\(s:= l;\)
while \(s < n\) do
    begin
        \(A_k:=\) adjacency structure of strong component \(K\) with least vertex in subgraph of \(G\) induced by \(\{s, s+ 1,\)
        \(n);\)
        if \(A_k \neq \emptyset\) then
            begin
                \(s:=\) least vertex in \(V_k;\)
                for \(i \in V_k\) do
                    begin
                        blocked\((i):= false;\)
                        \(B(i):= \emptyset;\)
                    end;
            end;
        CIRCUIT := \(s;\)
        \(s:= s + 1;\)
        end
    else \(s:= n;\)
end;
end;
```

Figure A-1. Johnson's Algorithm
Appendix B. Process Slimming Algorithm

In Section 4.2.3, we discussed a simple way to integrate the adaptation elements, which puts all elements consisting of a receive-send pair in a parallel pattern with each other. For instance, the adapter process for the example in Figure B-1 (a) is: Par(Seq(R(m_2), S(m_2)), Seq(R(m_3), S(m_3))) (Figure B-1 (b)).

Figure B-1. Example to illustrate the slimming of adapter process

However, this adapter can be made "slimmer" by adding implicit control flows implied by control flow constraints in the two processes. For instance, in the example in Figure B-1 (a), since S2 < S3 (i.e. a flow occurs from S2 to S3) in process P, their dual activities R2 and R3 in the adapter should satisfy R2 < R3. Therefore we add a control flow link from R2 to R3 in the adapter process to reflect this flow. Moreover, the flow R3 < R2 in process Q implies S3 < S2 in the adapter. After adding those additional control flows, we need to delete redundant links implied by transitivity of flows. For example, if S1 < S2, S2 < S3, and S1 < S3, we delete the link from S1 to S3. These modifications result in the adapter of Figure B-1 (c). This process slimming algorithm is formally defined as follows,
Algorithm B-1. Process slimming algorithm

Step 1. Setup a table to record the relationship between any two activities, and initial all relationships to ?, which stands for a null value. The relationship between two activities $a$ and $b$ could be any of the following five values,

?: null value, which means that there is no relationship between two activities.

 <: immediately before. For instance, $a < b$ stands for that $b$ executes immediately after $a$.

>: immediately after. For instance, $a > b$ stands for that $a$ executes immediately after $b$.

<<: implicit before. For instance, $a << b$ stands for that $b$ executes after $a$, but not in an immediate manner.

>>: implicit after. For instance, $a >> b$ stands for that $a$ executes after $b$, but not in an immediate manner.

Step 2. When add a link like $a \rightarrow b$, check the relationship table for $a$ and $b$,

Case 1: if $a < b$ or $a << b$, do nothing and go to step 4.

Case 2: if $a > b$ or $a >> b$, report error and exit algorithm.

Case 3: if $a ? b$, add $a < b$ in relationship table and go to step 3.

Step 3. Given a newly added relationship $a < b$, update relationship table.

Step 3.1. add $b$ to a list $L$;

Step 3.2. If $L$ is empty, go to step 4. Otherwise take an activity $x$ from list $L$; For each $y$ such as $x < y$ or $x << y$, check the relationship between $a$ and $y$,

Case 1: if $a < y$, update it to $a << y$

Case 2: if $a << y$, do nothing.

Case 3: if $a > y$ or $a >> y$, report error and exit algorithm.

Case 4: if $a ? y$, add a $< y$ in relationship table and add $y$ to $L$.,
Step 4. When add a link like a <- b, do a similar processing as step 2 and 3.

Step 5. For each > or < in relationship table, add <- or -> links to graph.

**Example B-1.** For the example in Figure B-1, the insertions of implicit control flow links (R2 → R3 and S3 → S2) update the relationship table as show in Figure B-2. Figure B-2 (a) shows the initial relationship table. When adding R2 < R3, we also have R2 << S3, since R3 < S3 (Figure B-2 (b)). Similarly, when adding S2 > S3, we also have S2 >> R3 (Figure B-2 (c)). More importantly, we update R2 < S2 to R2 << S2 (Figure B-2 (d)), which deletes a direct control flow link (R2 → S2). Figure B-2 (d) gives the final result Seq(R(m_2), R(m_3), S(m_3), S(m_2)), which is shown in Figure B-1 (c).

![Table and Diagram](image-url)
Appendix C. CPLEX Listing for IP-Async Algorithm Implementation

//**************Define decision variables***************
dvar int SS[msgID][msgID] in 0..1;
dvar int SR[msgID][msgID] in 0..1;
dvar int RS[msgID][msgID] in 0..1;
dvar int RR[msgID][msgID] in 0..1;
dvar int TSS[msgID][msgID] in 0..1;
dvar int YSS[msgID][msgID] in 0..1;

//*****Define objective function function*****
minimize sum((i,j) in msgID) (SS[i][j]) + sum((i,j) in msgID) (RR[i][j]) + sum((i,j) in msgID) (RR[i][j]) + 2*sum((i,j) in msgID) (SS[i][j]);

subject to {

    //**** Intra process Transitivity Constraints***
    forall (i,j,k in msgID)
        SS[i][j] == 1 && SR[i][k] == 1 => SR[i][k] == 1;
    forall (i,j,k in msgID)
        SS[i][j] == 1 && SS[i][k] == 1 => SS[j][k] == 1;
    forall (i,j,k in msgID)
        SR[j][j] == 1 && RS[j][k] == 1 => SS[j][k] == 1;
    forall (i,j,k in msgID)
        SR[j][j] == 1 && SS[j][k] == 1 => SR[j][k] == 1;
    forall (i,j,k in msgID)
        RR[i][j] == 1 && RS[i][k] == 1 => RR[i][k] == 1;
    forall (i,j,k in msgID)
        RR[i][j] == 1 && SR[i][k] == 1 => RR[i][k] == 1;
    forall (i,j,k in msgID)
        RR[i][j] == 1 && SS[i][k] == 1 => RR[i][k] == 1;
    forall (i,j,k in msgID)
        RR[i][j] == 1 && SR[i][k] == 1 => RR[i][k] == 1;

    //*****Inter process constraints ******
    forall (i,j in msgID)
        SS[i][j] == 1 => SS[i][j] == 1;

    //*****Async communication Constraints***
    forall (i,k in msgID)
        (RR[i][k] == 1) => SS[i][k] + XSS[i][k] == 1;
    forall (i,j in msgID)
        RR[i][j] == 1 => SS[i][j] - YSS[i][j] == 0;
    forall (i,k in msgID)
        (j != i && j != k)
            && (SS[j][k] + XSS[j][k] == 1) &
            && (SS[i][k] + XSS[i][k] == 1) => TSS[i][k] == 1;
/**other - default values for RR, RS and SR arrays**
forall (i, j in msgID)
  RR[i][j] != 1 => RR[i][j] == 0 ;
forall (i, j in msgID)
  RS[i][j] != 1 => RS[i][j] == 0 ;
forall (i, j in msgID)
  SR[i][j] != 1 => SR[i][j] == 0 ;

/***** direct links in processes to be composed *****/
SS[6][5] == 1;
SR[5][7] == 1;
RS[7][9] == 1;
SR[9][4] == 1;
SS[9][8] == 1;
SS[0][7] == 1;
SS[7][3] == 1;
SR[3][9] == 1;
RR[9][6] == 1;
RS[9][1] == 1;
RR[6][2] == 1;
SR[1][2] == 1;
RR[0][1] == 1;
RR[8][3] == 1;
RS[3][2] == 1;
SR[2][1] == 1;
SR[4][5] == 1;
RR[5][1] == 1;

}
References

BELL LABS 2009. SPIN.


BERKELAAR, M. 2010. lp_solve.


OASIS UDDI SPECIFICATIONS TC 2004. Universal Description, Discovery and Integration (UDDI) 3.


STOLLBERG, M. 2005. Reasoning Tasks and Mediation on Choreography and Orchestration in WSMO. In 2nd International WSMO Implementation Workshop (WIW '05), Innsbruck, Austria, 656 - 664


W3C MEMBER 2004. OWL-S: Semantic Markup for Web Services, Available at: http://www.w3.org/Submission/OWL-S/.

W3C WEB SERVICES DESCRIPTION WORKING GROUP 2001. Web Services Description Language (WSDL) 1.1, Available at http://www.w3.org/TR/wsd1.


VITA
Zhe Shan

EDUCATION

- Ph.D. in Business Administration & Operations Research
  The Pennsylvania State University (PSU) August, 2011
  Dissertation: Integrated Business Process Adaptation
  Advisor: Dr. Akhil Kumar

- M.Phil. in Computer Science City University of Hong Kong (CityU) June, 2003
  Thesis: Time-compatibility and State-compatibility of a Workflow System
  Advisor: Dr. To-yat Cheung

- B.Sc. in Computer Science Nanjing University, China (NJU) June, 2000
  Thesis: A CSCW System Framework based on CORBA/Java
  Advisor: Prof. Daoxu Chen

RESEARCH AND INDUSTRY EXPERIENCE

- Research Intern IBM TJ Watson Research Center Jun 2010 – Sep 2010
  • Entity-Centric Real-Time Business Performance Monitoring

- Research Intern IBM TJ Watson Research Center May 2008 – Aug 2008
  • Optimal Decision Making for Services Enterprises

- Research Assistant SC&IS, PSU Aug 2006 – Present
  • Integrated Service Process Adaptation
  • Composition and Verification of Web Services
  • Mining Business Process Logs

  • Design and Evaluation of Improvement Method on the Web Information Management and Navigation

- Senior Research Assistant CS, CityU, HK Oct 2003 – Feb 2006
  • Workflow View - A Novel Mechanism for Workflow Management System in E-business Environment

- Visiting Research Associate CDE, IIT, Indi Jul 2004-Aug 2004
  • Process Conformances in Workflow Systems -Taxonomy, Framework, Research Issue and Solution

- Visiting Research Associate SKLSE, WHU, China Feb 2004&Sep 2004
  • Deputy Mechanism for Workflow Views

  • Specification and Verification for Computer-based Software and Workflow Systems

- Research Assistant CS, NJU, China Jul 1998 - Jun 2000
  • Computer-Supported Collaborative Scientific Research System

PROFESSIONAL ACTIVITIES

- Student member
  • Association for Computing Machinery (ACM)
  • IEEE Computer Society
  • Association for Information Systems (AIS)
  • INFORMS Information Systems Society