KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT ECONOMIE EN BEDRIJFSWETENSCHAPPEN

Pattern-based Coordination in Process-based Service Compositions

Proefschrift voorgedragen tot het behalen van de graad van Doctor in de Toegepaste Economische Wetenschappen

door

Geert MONSIEUR

Nummer 351 2010

Committee

president	Prof. dr. Marleen Willekens	K.U.Leuven
supervisor	Prof. dr. Monique Snoeck	K.U.Leuven
supervisor	Prof. dr. Wilfried Lemahieu	K.U.Leuven
co-supervisor	Prof. dr. Guido Dedene	K.U.Leuven
	Prof. dr. Willem-Jan Van den Heuvel	Tilburg University
	Prof. dr. Uwe Zdun	University of Vienna

Dankwoord

Feeling gratitude and not expressing it is like wrapping a present and not giving it.

– **William Arthur Ward** (1921-1994), American writer

Eerst en vooral wil ik mijn dank betuigen aan mijn promotoren, prof. dr. Monique Snoeck en prof. dr. Wilfried Lemahieu, en mijn co-promotor, prof. dr. Guido Dedene.

Monique, bedankt om mijn schrijfsels steeds te voorzien van uitgebreide en constructieve commentaar. Ook al kreeg u het als programma-directeur en later vice-decaan onderwijs steeds drukker, toch maakte u tijd vrij om mij te begeleiden. In het bijzonder speelde u voor mij een grote rol door me te helpen zoeken naar de rode draad in mijn doctoraat en onderzoek. Indien ik op zekere dag de draad eerder als *blauw* aanzag, gaf u niet op en kon u mij weer motiveren om verder te gaan door samen een *groene* draad te zoeken.

Wilfried, ook u las mijn artikels vaak tot in de kleinste details na, waarvoor mijn oprechte dank. Doctoreren vergeleek u soms met een zoektocht naar de kerk in een onbekend dorp wanneer je alleen de kerktoren ziet. Vandaag heb ik de kerk gevonden en dit is mede dankzij uw bijdrage. Bedankt voor de vele onderzoeksideeën die u deelde met mij, de grote interesse in mijn werk, en de bereidheid om in mij en mijn onderzoek te investeren.

Guido, van u heb ik niet alleen geleerd dat studie en onderzoek gepaard moet gaan met een Belgisch bier (e.g. Witte van Hoegaarden en pindanootjes), maar tijdens de doctorale seminaries wist u me ook steeds te verbazen met nieuwe inzichten en suggesties voor verder onderzoek. Verder wil ik u graag bedanken om mijn werk steeds nog beter te willen maken.

Naast mijn promotoren en co-promotor kon ik ook rekenen op het deskundig advies van twee buitenlandse commissieleden, prof. dr. Willem-Jan Van den Heuvel (Tilburg University) en prof. dr. Uwe Zdun (University of Vienna).

Willem-Jan, tot ongeveer een half jaar geleden kende ik u voornamelijk als een auteur van de vele artikels die ik de voorbije jaren las. Ik ben dan ook vereerd dat u deel uit wilde maken van mijn jury. Verder zou ik u oprecht willen danken voor de tijd die u vrijmaakte om mijn werk nauwgezet te lezen. Uit de talrijke tips die u mij gaf om dit doctoraat een niveautje hoger te tillen, heb ik enorm veel geleerd. Het deed me dan ook veel plezier om te horen dat u een duidelijke evolutie zag in de korte tijd tussen ons gesprek in Tilburg en mijn presentatie in Leuven.

Uwe, already before you were an ’official’ member of the jury, you took some time to read my papers. I’m really grateful to you for your valuable comments. Each time you managed to come up with lots of *suggestions for improvement*. Furthermore, I would like to thank you for convincing me to implement large parts of my research. Although this required me to spend almost day and night in front of my computer in the past few months, it really helped me to make the story of this PhD complete. Finally, I also thank you for making EuroPLoP conferences unforgettable in two ways. Firstly, having you as workshop leader was the best way to learn everything about pattern writing. Secondly, I always enjoyed beating you when we played the world’s famous soccer game *Germany against the rest of the world*.

Naast mijn doctoraatscommissie hebben ook vele anderen een belangrijke rol gespeeld bij de totstandbrenging van dit proefschrift.

Eigenlijk begon alles bij de start van mijn academische carrière. Na de zomervakantie van 2001 begon mijn universitaire loopbaan in Leuven. Ik koos voor de opleiding tot kandidaat en licentiaat in de informatica. Waarom zou ik iets anders kiezen, want computers waren toch leuk? Bovendien zou je relatief eenvoudig werk vinden, want bedrijven zagen informatici graag toekomen. Mijn omgeving keek vol spanning toe hoe ik het ervan af ging brengen. Na vier jaar kon ik (samen met de mensen rond mij) tevreden terugkijken. Om die reden volgen verschillende woordjes van dank. Dank in de eerste plaats aan mama en papa die me de kansen gaven om een univeritair diploma te behalen en steeds achter mij stonden. Ook mijn broer en zus, Dirk en Nadine, speelden voor mij een belangrijke rol. Dirk was ongetwijfeld de beste medestudent en kotgenoot die je je kan voorstellen, terwijl Nadine zorgde voor de *betere* maaltijden of me door de moeilijke oefeningen van lineaire algebra sleurde in ’eerste kan’. Ook als doctoraatsstudent kon ik de voorbije jaren rekenen op de steun en liefde van mijn ouders, broer, zus en hun partners. Bedankt daarvoor! Mama & papa, Dirk & Liesbeth, Nadine & Wim, ik vind het fijn dat jullie vandaag meevieren. Ook mijn inmiddels overleden grootouders (Oma & ’Peter’, Moeke & Bompa) hebben mijn ’carriere’ steeds met veel fierheid gevolgd. Moeke, je keek heel hard uit naar de publieke verdediging van mijn doctoraat. Jammer genoeg moest je ons onverwacht op 16 november 2010 verlaten. Ik wil je in ieder geval bedanken om steeds achter mij te staan en ik zal nooit je blik vergeten toen je ongeveer twee maanden geleden vol trots door de voorlopige versie van mijn doctoraat bladerde. Het ga je goed!

Tijdens de jaren als KUL-student was het uiteraard niet altijd alleen maar studeren en examens wat de klok sloeg. Ik bedank dan ook graag Bram, Kris, Peter, Piet(er), Sarah, Sven, Tim, Toon (de beste postermaker van de wereld) en Toon (de netwerkspecialist) voor de vele leuke feestjes in de Winabar of een of andere fuifzaal in het Leuvense.

In 2003 werd Marijke een deel van mijn leven. Sindsdien werd ik ook aangemoedigd door haar ouders, Anne-Liese & Henri. Bij deze wil ik hen ook bedanken voor de verschillende 'overlevingspakketten' die Marijke en ik steeds krijgen tijdens zware en drukke (examen)periodes. Ook gaat mijn dank uit naar Marijke's zus Anne-Katrien omdat ze de voorbije jaren steeds klaarstond voor ons. Marijke's broer Johan en diens vriendin Leen ben ik dankbaar voor hun gastvrijheid en het uitwisselen van technologie-weetjes.

Zomer 2005, tijd om een job als informaticus te zoeken, maar plots was er de mogelijkheid om te doctoreren. Doctoreren was grotendeels nog onbekend voor mij, maar al snel was ik overtuigd van deze unieke kans. Bovendien was het een aangenaam gevoel om te zien dat verschillende vrienden en studiegenoten beslisten om aan het werk te gaan als onderzoeker. Zou ik net als hen aan het werk gaan bij het departement computerwetenschappen? Tijdens de lessen van prof. dr. Guido Dedene was ik immers verliefd geworden op *beleids*informatica. Het toeval wil dat er ook een mogelijkheid was om bij de onderzoeksgroep beleidsinformatica aan een doctoraat te werken. Ik besloot ervoor te gaan en solliciteerde met veel motivatie en interesse bij Monique. Zoals hierboven (en op pagina iii) vermeld, is zij een promotor van mijn doctoraat. Ik wil haar dan ook bedanken dat ze in mijn onderzoekscapaciteiten geloofde en me in oktober 2005 tewerkstelde aan de faculteit economie en bedrijfswetenschappen.

Het leven als doctoraatstudent was soms hard labeur, maar ook heel variërend en boeiend. Mijn eerste 'opdracht' bestond uit het maken van een ER-diagram op een reuzeposter als cadeau voor de emeritaatsviering van prof. dr. Jacques Vandenbulcke. Samen met nieuwkomer Birger leerde ik op die manier de vele collega's van toen op een aangename manier kennen. Sommigen van hen zijn ondertussen elders aan het werk gegaan, maar ik dank Johan, Stijn, Joke, Birger en Bjorn alvast voor de fijne tijd. De *ancien* van toen, Frank, wil ik graag bedanken omdat hij me enkele uren voor de emeritaatsviering erop attent maakte dat ik best een *deftige* broek en *sjiek* hemd zou dragen. Frank, later heb je ook op wetenschappelijk vlak een waardevolle bijdrage geleverd aan dit doctoraat, een Rijselse merci! Ex-collega Lotte wil ik graag bedanken om mij te introduceren in het wereldje van patronen en de bijhorende PLoP-conferenties. Manu, ik wil jou bedanken omdat je jouw ervaringen als doctoraatsstudent deelde met mij, vaak spontaan interesse toonde in mijn voortgang en ik mocht samenwerken met je. Prof. dr. ir. David, jou wil ik bedanken om de after-work feestjes steeds te voorzien van een

plezante start. Raf, ook jij mag niet ontbreken in dit dankwoord. Ik vond het steeds prettig om met jou te babbelen over de nieuwste technologie of de nieuwste hype in de SOA-wereld. Verder wil ik ook mijn ex-bureaugenootjes Caroline en Inge bedanken voor de gezellige momenten op kantoor.

Een onderzoeksgroep leeft niet zonder professoren, daarom ook mijn dankbetuiging aan Jan Vanthienen (die de voorbije jaren vaak voor afleiding zorgde met behulp van recepties, etentjes en fijne kerstuitstapjes), Bart Baesens (voor zijn waardevol onderzoeksadvies en wereldwijde restauranttips) en Ferdi Put (voor de leuke, soms technische, gesprekken).

Ook de huidige collega-doctoraatstudenten verdienen uiteraard een vermelding in dit dankwoord: Helen (for convincing me to visit Paris one day), Thomas (om het Limburgs gevoel te versterken), Tom (om mijn ecologische voetafdruk te compenseren door minder af te drukken), Isel (for the temporary increase in the percentage of female researchers), Filip (om avondjes uit met de collega's onvergetelijk te maken), Willem (om data provider te spelen tijdens mijn voorlopige verdediging), Karel (om LIRIS sportief te houden), Wouter (voor de babbels en pintjes op vrijdagavond), Philippe (om me te doen dagdromen over Canada), Jochen (om iets te kunnen bijleren over process mining), Jonas (voor zijn gezelschap tijdens de avondshifts op de faculteit), Flavius (for learning me what 'quality of service' means) en Piet(er) (voor de LaTeX-hulp en inspirerende discussies). Bij doctoreren komt ook heel wat administratie kijken. Gelukkig werd ik de voorbije jaren verlost van deze taken dankzij Elke Tweepenninckx, Nicole Meesters, Annie Vercruysse en Isabelle Theys.

Doctoreren is niet altijd even makkelijk en gaat soms gepaard met een portie stress, maar gelukkig zorgden veel vrienden op tijd en stond voor ontspanning. Met veel plezier denk ik terug aan de zomer van 2007, toen Sven en ik beiden in Salt Lake City (Utah, USA) een presentatie mochten geven op een internationale conferentie. We besloten deze 'zakenreis' te combineren met een grootse rondreis door West-Amerika. Anne-Katrien, Bert, Marijke, Peter, Pieter en Sven, bedankt voor deze prachtige ervaring! Ook het jaarlijks eetfestijn in de Ardennen doet me lachen, net als onze trips naar de groene Alpen in Zwitserland en de witte in Frankrijk.

Het mag duidelijk zijn dat de vrienden rond me een belangrijke rol spelen in mijn leven en tijdens de voorbereiding van dit doctoraat. Om die reden een welgemeende 'dank u' voor Anne-Katrien (die op belangrijke momenten steeds voor mij duimt, zelfs met haar grote tenen), Sofie (voor het amusant gezelschap tijdens reisjes naar Dublin en Sardinië), Sven & Femke (om aan te tonen dat hardwerken perfect gecombineerd kan worden met uitgaan), Annelies & Bart (voor de verrukkelijke etentjes en gezellige pokeravonden), Piet & Eveline (voor de heerlijke Kwak-momenten en de spelletjestoevoer), Kris & Isabel (voor de traditionele verjaardagsfeestjes en Zwitserse uitstapjes), Sarah & Tim (voor de fijne momenten op café en restaurant), Veronique & Karel (voor de spannende wielermomenten en om voor te doen dat partners samen kunnen doctoreren), Bert (om steeds bereid te zijn om te helpen met eender wat), Peter (voor de Prolog-ondersteuning en het gezelschap op café), Geert (voor de boeiende en gevarieerde gesprekken), Bram (voor leuke feestjes en uitstapjes zoals een bezoek aan het autosalon) en alle vrienden van Marijke die in de loop der jaren ook mijn vrienden werden. Ook dank ik de Tilburgse collega's van Marijke die de voorbije maanden meeleefden en aan ons dachten. De vrienden uit Maasmechelen kunnen eveneens niet onvermeld blijven. Barbara & Bas, Kathleen & Philippe, Ilse & Paul, Jochen, Yves & Ans, Veerle, Sarah & Stefan, ook al zie ik jullie niet vaak, ik vind het altijd fijn om jullie terug te zien en bij te praten.

Een constante gedurende het hele verhaal is en blijft Marijke: Marijke, jouw bijdrage aan dit doctoraat heeft voor mij een grote waarde. Je was mijn steeds klaarstaande coach, trouwste fan en meest creatieve inspiratiebron voor tal van voorbeelden in mijn onderzoek. De voorbije maanden kon ik letterlijk en figuurlijk dag en nacht op je rekenen en vormde je jezelf om tot mijn keukenprinses. Bedankt lieverd voor al je steun en lieve woordjes!

Ten slotte wil ik iedereen bedanken die op 17 december 2010 mijn publieke verdediging bijwoonde. Santé!

– Geert Monsieur
Leuven, december 2010

Inhoudsopgave

Committee **iii**

Dankwoord **v**

Samenvatting **xvii**

1 Introduction **1**

1.1 Research context 2

1.1.1 Services and service-based systems 2

1.1.2 Challenges related to service-based systems 3

1.2 Research goal and questions 6

1.3 Research methodology 7

1.4 Thesis outline 14

2 Related work **15**

2.1 Dependencies in service compositions 15

2.2 Managing sequence dependencies 18

2.2.1 Introduction 18

2.2.2 A meta-model for service composition 19

2.2.3 Orchestration and choreography as composition viewpoints 21

2.2.4 Orchestration and choreography as composition styles 23

2.2.5 Conclusion 25

2.3 Managing data dependencies 26

2.3.1 Running example 26

2.3.2 Existing techniques for dealing with data dependencies 27

2.3.3 Conclusion 29

2.4 Patterns for Service-Oriented Architectures 31
2.4.1 Service interaction patterns 31
2.4.2 Pattern-based architectural framework for Service-Oriented Architectures 32
2.4.3 Flexible coordination of service interaction patterns . . 33
2.5 Related standards 34
2.5.1 Business Process Modeling Notation (BPMN) 34
2.5.2 Web Services Description Language (WSDL) 35
2.5.3 Business Process Execution Language (BPEL) 37

3 Managing sequence dependencies 39
3.1 Introductory example 39
3.2 Pattern language 45
3.2.1 Introduction 45
3.2.2 Pattern overview 48
3.2.3 Controlled Service Provider 49
3.2.4 Independent Controller 53
3.2.5 Controlling Service Provider 55
3.2.6 Coordinator 59
3.2.7 Self-controlled Service Provider 63
3.3 Applying the patterns in practice 66
3.3.1 Design guidelines 66
3.3.2 Coordination in workflow patterns 67
3.4 Evaluation 72
3.4.1 Shepherding and writers' workshop 73
3.4.2 Completeness of the pattern language 74
3.5 Conclusion 76

4 Managing data dependencies 79
4.1 Introductory example 79

4.2 Pattern language 82
4.2.1 Introduction 82
4.2.2 Pattern overview 86
4.2.3 Data flow initiation 87
4.2.4 Direct-Indirect request 91
4.2.5 Direct-Indirect transmission 94
4.3 Applying the patterns to construct coordination scenarios . . . 97
4.3.1 Combining the patterns into coordination scenarios . . 97
4.3.2 Design guidelines for applying the patterns 98
4.3.3 Applying the patterns and guidelines to the hospital example 102
4.4 Demonstrating the practical utility 106
4.4.1 Analytical validation: a shepherding process and a writers' workshop 107
4.4.2 Observational evaluation: real-life insurance case . . . 107
4.4.3 Descriptive evaluation: flexible coordination of service interactions 112
4.5 Completeness confirmation 114
4.5.1 Formalizing a coordination scenario 116
4.5.2 Executing the Prolog program 118
4.6 Conclusion 119

5 Combining the pattern languages 121
5.1 Data dependencies service compositions 121
5.2 Managing data dependencies in service compositions 123
5.3 Combining sequence and data dependency management . . . 124
5.4 Conclusion 128

6 Tool support for pattern-based coordination 131
6.1 From business process modeling to service composition 131
6.1.1 Existing BPMN-to-BPEL transformations 132

6.1.2 Pattern-based service composition and coordination . . 133
6.2 Input for pattern-based service composition and coordination 134
6.2.1 Representing sequence and data dependencies in BPMN 134
6.2.2 An additional data dependencies model 135
6.2.3 Representing a coordination model 136
6.3 Pattern-based service composition and coordination 139
6.3.1 Processing the input models 141
6.3.2 Generating the output files 143
6.4 Demonstration in a concrete example 150
6.4.1 Sequence and data dependencies in the travel agency example 150
6.4.2 Specifying the input models 152
6.4.3 The generated BPEL and WSDL files 156
6.4.4 Testing generated BPEL processes 160
6.5 Conclusion 163

7 Conclusions 165
7.1 Research objectives evaluation 165
7.2 Limitations and issues for future research 169
7.2.1 Sequence versus action dependencies 170
7.2.2 Extending the execution model 172
7.2.3 Time dimensions in data dependencies management . 173
7.2.4 Additional validation 177
7.2.5 Joined forces 177
7.2.6 Composite data requests 178
7.2.7 Patterns for self-adaptive service coordination 178

A Completeness confirmation in Prolog 185

B XML Schemas 189
B.1 XML Schema for a data dependencies model 189

B.2 XML Schema for a coordination model 191

List of Figures **195**

List of Tables **199**

List of Listings **201**

Bibliography **205**

Doctoral dissertations from the Faculty of Business and Economics **217**

Biography **219**

Samenvatting

Om vandaag de dag competitief te blijven zijn bedrijven steeds meer genoodzaakt om zich snel te kunnen aanpassen aan een steeds veranderende omgeving. Zowel organisaties als wetenschappers zijn het er over eens dat *business agility* een kritische succesfactor is om te kunnen antwoorden op trends zoals een stijgende vraag naar meer productvariatie, snellere innovatie en business-on-demand (Poppendieck & Poppendieck, 2006).

Om tegemoet te komen aan deze vraag naar flexibiliteit hebben bedrijven nood aan nieuwe organisatiestructuren. Ze transformeren steeds meer van stabiele en monolitische organisaties naar meer gedistribueerde en service-georiënteerde organisaties (Di Nitto, Ghezzi, Metzger, Papazoglou, & Pohl, 2008). Zulke organisaties concentreren zich steeds meer op hun belangrijkste competenties (Kohlborn, Korthaus, Chan, & Rosemann, 2009; Cherbakov, Galambos, Harishankar, Kalyana, & Rackham, 2005; Hagel III & Singer, 1999). Om flexibel te zijn, ontwikkelen mensen en organisaties bepaalde competenties die hen ondersteunen om typische zakelijke problemen op te lossen. Service-oriëntatie is niets anders dan het bundelen van die competenties in mooi afgebakende en duidelijk beschreven services, vaak gerelateerd aan een bepaalde functionele context (OASIS, 2006a). Een service moet het mogelijk maken om op de juiste momenten gebruik te kunnen maken van deze competenties (Erl, 2005, 2007).

Net zoals organisatiestructuren, worden informatiesystemen ook meer en meer gebouwd door softwareservices te combineren tot flexibele en adaptieve service-gebaseerde systemen. Softwareservices zijn zelfbeschrijvende en platformagnostische software-elementen die toelaten om relatief snel en eenvoudig gedistribueerde applicaties te ondersteunen (Papazoglou, 2003; Papazoglou & Van den Heuvel, 2007b). In het algemeen, voeren services bepaalde functies uit, gaande van eenvoudige taken tot complexe bedrijfsprocessen. Vaak zijn services toegankelijk via Internet (of intranet) en maken ze gebruik van gestandardiseerde talen en vaste protocollen (Papazoglou, 2003; Papazoglou & Van den Heuvel, 2007b).

Service-gebaseerde systemen worden meestal geïmplementeerd met een service-georiënteerde architectuur (SOA). Typerend voor SOA's is de hiërarchische compositie van services. Een servicecompositie kan afzonderlijk aangeboden worden, maar ook deeluitmaken van nieuwe servicecomposities om uiteindelijk een service-gebaseerd systeem te bouwen ter ondersteuning van een bedrijfsproces. De grondgedachte achter een SOA is dat bedrijfsprocessen

ondersteund en uitgevoerd worden door onafhankelijke, combineerbare en configureerbare services.

Jammer genoeg is het niet evident om services te combineren tot volwaardige service-gebaseerde systemen. Momenteel bestaan er geen richtlijnen om hoogniveau bedrijfsprocessen eenvoudig en snel te vertalen naar technische implementaties. Servicecompositie vereist een precieze specificatie van de coördinatie van de te combineren services. Alle interacties met services moeten precies beschreven worden. Dit is meestal een tijdrovende taak en het resultaat is vaak complexe programmacode. Bovendien verloopt het ontwerp van coördinatiescenario's vaak ad hoc. Er bestaan noch specifieke richtlijnen noch tools om coördinatiescenario's te ontwerpen. Op de koop toe moeten ontwikkelaars vaak dezelfde zaken herhaaldelijk implementeren. Samengevat kan men dus stellen dat er nood is aan een systematische manier om coördinatiescenario's op een gepaste wijze te ontwerpen. Met dit doctoraat trachten we hiertoe bij te dragen en stellen we de volgende onderzoeksvragen voorop:

OV1 Bestaat er een systematische manier om coördinatiescenario's samen te stellen, gebruikmakend van fundamentele bouwblokken die gecombineerd kunnen worden tot volledige coördinatiescenario's?

OV2 Bestaat er een manier om ontwikkelaars te begeleiden en ondersteunen tijdens de constructie van coördinatiescenario's met behulp van concrete ontwerprichtlijnen?

Alvorens op zoek te gaan naar antwoorden op deze twee onderzoeksvragen, is het belangrijk om te weten wat coördinatie betekent. In deze thesis volgen we hiervoor de definitie van Malone en Crowston (1994), die coördinatie definiëren als het *beheren van afhankelijkheden*. In het verleden hebben verschillende onderzoekers tal van soorten van afhankelijkheden binnen servicecomposities onderscheiden. De twee meest voorkomende afhankelijkheden zijn volgorde-afhankelijkheden (sequence dependencies) en data-afhankelijkheden (data dependencies). In ons onderzoek richten we ons dan ook voornamelijk tot deze types van afhankelijkheden. Een volgorde-afhankelijkheid tussen twee services betekent dat een service pas uitgevoerd mag worden nadat de andere service is uitgevoerd (bv. een transport-service mag pas uitgevoerd worden nadat een betaalservice succesvol voltooid werd). Een data-afhankelijkheid tussen twee services stelt dat een service data nodig heeft die geleverd kan worden door de andere service (bv. een transport-service kan pas uitgevoerd nadat een klantenservice het afleveradres bezorgd heeft).

De oplossingen gepresenteerd in dit doctoraat worden beschreven in de vorm van zogenaamde *(ontwerp)patronen* (design patterns). Vereenvoudigd voorgesteld kan men een patroon beschouwen als een *oplossing* voor een *terugkerend probleem* in een bepaalde *context*. Typisch is een mogelijke oplossing voor zo'n probleem onderhevig aan zogenaamde *krachten* die aanwezig zijn in de context waarin het probleem zich bevindt. Een oplossing voor het probleem houdt rekening met deze krachten en bevat zonodig een specifieke afweging tussen de verschillende aanwezige krachten. De keuze om de resultaten van ons onderzoek neer te schrijven in de vorm van patronen is niet willekeurig. Niet alleen is coördinatie een terugkerend probleem in servicecompositie, maar een service-georiënteerde architectuur gaat ook gepaard met verschillende krachten (bv. gewenste niveau van losse koppeling tussen services, strenge eisen inzake vertrouwelijkheid van informatie, nood aan flexibiliteit, etc.). Aangezien patronen oplossingen beschrijven door ondermeer een afweging te maken tussen deze krachten, is de patroonvorm uitermate geschikt om te komen tot een antwoord op de tweede onderzoeksvraag. In dit doctoraat worden verschillende patronen voorgesteld, die allemaal gebruikt kunnen worden als fundamentele bouwblokken voor coördinatiescenario's. Samen vormen deze patronen twee zogenaamde *patroontalen*, een voor het beheer van volgorde-afhankelijkheden en een voor het beheer van data-afhankelijkheden. De patronen zijn als het ware het vocabularium, en de regels om de patronen te combineren en te implementeren vormen de grammatica van de (patroon)taal.

In dit doctoraat hebben we zowel voor de patroontaal voor het beheer van volgorde-beperkingen als de patroontaal voor het beheer van data-afhankelijkheden aangetoond dat de beschreven patronen voldoende zijn om alle mogelijke coördinatiescenario's te kunnen samenstellen. Dit heeft als groot voordeel dat ontwikkelaars voortaan coördinatiescenario's kunnen ontwerpen door enkel en alleen een aantal patronen te combineren. Kortom, de patronen vormen bouwblokken voor coördinatiescenario's, waarmee de eerste onderzoeksvraag beantwoord is. Bovendien bieden de patronen ook concrete ontwerprichtlijnen, aangezien ieder patroon heel precies neerschrijft hoe het de verschillende aanwezige krachten afweegt, zodat ook de tweede onderzoeksvraag niet langer onbeantwoord is.

Op basis van de twee patroontalen hebben we vervolgens een tool ontwikkeld die toelaat om automatisch volledige coördinatiescenario's te genereren vertrekkend van bedrijfsprocessen. De invoer voor deze tool bestaat uit drie elementen. Ten eerste is er een bedrijfsproces vereist. Dit bedrijfsproces moet gespecifieerd worden gebruikmakend van de Business Process Modeling Notation (BPMN) (OMG, 2010a) en heeft in de eerste plaats als doel om volgorde-afhankelijkheden te modelleren. Aangezien BPMN slechts in beperkte mate toelaat om data-afhankelijkheden te specifiëren, bestaat het tweede invoerelement voor de tool uit een zogenaamd data-afhankelijkhedenmodel. Dit model is niets anders dan een XML-bestand waarin bijkomende data-afhankelijkheden vastgelegd kunnen worden. Ten slotte moet de gebruiker van de tool een keuze maken welke patronen hij of zij wil gebruiken om de tool een compleet coördinatiescenario te laten genereren. Het resultaat van de generatie is een verzameling van BPEL[1]- en WSDL[2]-bestanden die een volledig en uitvoerbaar coördinatiescenario beschrijven.

Dankzij de ontwikkeling van de twee patroontalen en de bijhorende integratie in een concrete tool, kunnen ontwikkelaars in de toekomst op een systematische en meer eenvoudige manier coördinatiescenario's samenstellen. Het coördinatievraagstuk werd herleid tot de kern van het probleem, en bestaat voortaan uit het kiezen en combineren van patronen. Complexe en herhaaldelijke implementaties van gelijkaardige patronen zijn bovendien dankzij de automatische generatie van coördinatiescenario's niet meer nodig.

[1]Business Process Execution Language (BPEL) (OASIS, 2007)
[2]Web Services Description Language (WSDL) (W3C, 2001, 2007a)

Information technology and business are becoming inextricably interwoven.
I don't think anybody can talk meaningfully about one without the talking about the other.

– **Bill Gates** (1955),
Chairman and founder of Microsoft

1 Introduction

In today's business world the economic success of an enterprise depends on its ability to adapt to the ever-changing business environment. Organizations are confronted with changes such as new introduced laws or changes in customers' attitudes. Therefore, companies and researchers have recognized business agility as a critical success factor for being able to cope with business trends like increasing product and service variability, faster time-to-market, and business-on-demand (Poppendieck & Poppendieck, 2006).

To increase the level of flexibility and agility many organizations are transforming from stable and monolithic enterprises into dynamic and distributed service-oriented enterprises (Di Nitto et al., 2008). Organizations more and more tend to focus on their core competencies (Kohlborn et al., 2009; Cherbakov et al., 2005; Hagel III & Singer, 1999). In order to enhance their flexibility, people and organizations create capabilities to solve or support a solution for the problems they face in the course of their business. Service-orientation is about grouping these capabilities into well-defined and scoped services. A service needs to enable access to one or more capabilities (OASIS, 2006a) that are grouped together because they relate to a functional context established by the service (Erl, 2005, 2007).

For example, Zara, a major fashion retailer, already started restructuring its organization using the service-orientation principles during the 1980s, when they decided to use groups of designers instead of individuals. Since then the company can offer considerably more products than similar companies. It produces about 11,000 distinct items annually compared with 2,000 to 4,000 items for its key competitors[1] (Mcafee, Sjoman, & Dessain, 2004).

[1] http://en.wikipedia.org/wiki/Zara_(clothing)

Moreover, it is claimed that the company can design a new product and have finished goods in its stores in four to five weeks; it can even modify existing items in as little as two weeks. As such, Zara has a competitive advantage because it can more rapidly meet its customers' preferences.

Organizations such as Zara (Mcafee et al., 2004) leverage similar trends and evolutions in the software systems that support the company's business processes. Similar to today's structures of organizations, information systems are more and more built by combining *software services* into flexible, dynamic and adaptive service-based systems. The research described in this thesis is situated in the domain of creating such flexible service-based systems.

In Section 1.1 we further describe the research context of this thesis. In that section we elaborate on software services and service-based systems, and the challenges that come with this innovative way of building information systems. Subsequently, Section 1.2 presents the research goal and questions that are addressed in this thesis. In Section 1.3 the research methodology that was used is presented. This chapter ends with an outline of this dissertation (see Section 1.4).

1.1 Research context

1.1.1 Services and service-based systems

Software services are self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications (Papazoglou, 2003; Papazoglou & Van den Heuvel, 2007b). In general, services perform functions, which can be anything from simple requests to complicated business processes. Services allow organizations to expose their core competencies programmatically over the Internet (or intra-net) using standard languages and protocols, and be implemented via a self-describing interface based on open standards (Papazoglou, 2003; Papazoglou & Van den Heuvel, 2007b).

Service-based systems are mostly implemented using a Service Oriented Architecture (SOA) (Metzger & Pohl, 2009). A central idea in an SOA is the (hierarchical) composition of multiple services. Such service compositions can in turn be offered to service clients, used in further service compositions and eventually be composed to service-based systems to support an organization's business processes. Therefore, service composition can be considered as a way to solve a real-world business problem by combining the functionality provided by several services.

Service composition aims at providing effective and efficient means for creating, running, adapting, and maintaining services that rely on other services in some way (Benatallah, Dijkman, Dumas, & Maamar, 2005). It is the use of service composition that lays the basis for an increased flexibility of an organization's information systems. A new or modified business process is supposed to be easily supported by (re)assembling the appropriate services into a new or adapted service-based system. In the past, information system support for the tasks in a business process was often realized by means of large monolithic applications that include the rules governing the execution of tasks into the programming code. This entails very inflexible systems that require a lot of time to be adapted to the ever changing business. Nowadays, in service-based systems, business processes are supported by highly independent, composable and reconfigurable services. When having services aligned to the business at your disposal, creating service-based systems largely comes down to firstly modeling the business processes that need to be supported. Subsequently, this business process model provides the blueprint for the service composition resulting in the service-based system. This way of creating systems is in contrast to creating programs based on a low-level programming language, which is sometimes referred to as programming in the small (DeRemer & Kron, 1975). Programming in the *large* emphasizes putting together large software components, built by several people over a long period of time, and having local state (e.g services). As such, modeling business processes as a first step towards a service-based system is an example of programming in the large (Emig, Momm, Weisser, & Abeck, 2005; Singh, Chopra, Desai, & Mallya, 2004; Leymann, 2006). In summary, one can say that business domains experts do programming in the large by using business process technology to compose new services out of existing services (Leymann, 2006).

1.1.2 Challenges related to service-based systems

Gap between process design and implementation

Beautiful as the concept of service composition can be, programming in the large comes with both advantages and disadvantages. On the one hand we like this way of creating and changing business applications since designers do not require detailed information technology skills, while on the other hand this causes strong expectations about the service-based implementation of the business processes. In order to maintain a proper alignment between the business and the information systems a precise translation of the high-level business processes to the technical implementation is needed. This is one of the difficulties often discussed in the literature. For example, in a study of Bandara et al. (2007) in which several Business Process Management (BPM) experts were interviewed, major problems that arise in the BPM life cycle are described. In particular they conclude that different vendors specialize in different aspects of the BPM life cycle, and often, due to a lack of standards, activities completed in one phase with one type of tool (e.g. BPMN (OMG, 2010a) processes modeled using Microsoft Visio[2] (Parker, 2010)) do not easily translate to the next steps of the life cycle, which may require the use of another type of tool (e.g. defining BPEL (OASIS, 2007) processes using Eclipse BPEL Designer[3] (Juric, 2006)). This creates a gap between the business process design and implementation. As a consequence the connection between the abstract level and the implementation is easily lost.

Complexity of message-based implementations

Another problem that is often described in the literature is the complexity of service-based business process implementations. Large scale service-based systems rely on complex message exchanges between individual software services. As the complexity and the number of interactions increase, there is a need to explicitly control and implement the service interactions (Henkel, Zdravkovic, & Johannesson, 2004; Monsieur, Snoeck, & Lemahieu, 2007; Monsieur, 2008; Snoeck, Lemahieu, Goethals, Dedene, & Vandenbulcke, 2004). The development of composite (Web) services still largely requires time-consuming hand coding, which entails a considerable amount of low-level programming.

Design of coordination logic and tool support

A service-based business process implementation requires the design of coordination logic. The service composition's coordination logic describes all service interactions needed for the service composition. Although a typical business process definition specifies the sequencing of activities, extra coordination logic is required to manage access to information and the progress of process execution.

Service composition and coordination are often performed in an ad-hoc fashion. Papazoglou and Van den Heuvel (2003) describe this situation as *"not desirable as it does not encourage 'clean' design and architecture, reuse*

[2]http://office.microsoft.com/en-us/visio
[3]http://www.eclipse.org/bpel

of existing building blocks, and, consistency of multi-programmer supported application construction projects”.

Therefore several researchers describe the need for development tools incorporating high-level abstractions for facilitating, or even automating, the (coordination) tasks associated with service composition. They state that these tools should provide the infrastructure for enabling the design and execution of composite services (Benatallah, Sheng, & Dumas, 2003; Benatallah et al., 2005). Additionally, there is a rising demand for business-driven automated composition (Papazoglou, Traverso, Dustdar, & Leymann, 2007; Dustdar & Schreiner, 2005), which implies an (semi-)automated design of coordination logic.

Tool support for constructing coordination logic is limited. For example, although Microsoft Visio (Juric, 2006) is the most frequently used tool for modeling BPMN processes (Recker, 2008), it is not trivial to design coordination logic for a BPMN process modeled in Visio (Juric, 2006). For example, even Microsoft’s Biztalk Orchestration Designer[4] does not support importing BPMN processes modeled in Visio. It is up to the developer to manually interpret the BPMN model and design appropriate coordination logic.

Major BPM players such IBM, Oracle, etc. try to close the gap between process design and implementation by offering a better interoperability between the process modeling and process implementation tools. Typically, these players offer a BPMN modeling tool (e.g. IBM WebSphere Business Modeler (Iyengar, Jessani, & Chilanti, 2007)) in which business analysts need to model business processes. Subsequently, the BPMN model can be imported in the implementation tool (e.g. IBM WebSphere Integration Developer (Iyengar et al., 2007)). Typically, such an implementation tool generates, based on the BPMN model, a BPEL (OASIS, 2007) ’skeleton’ in which developers can add service interactions and so define the coordination logic. However, this methodology decreases the flexibility, which was supposed to be one of the promising advantages of a service-based system. This is so because, if the business process changes, developers need to re-import the BPMN process model and redefine all service interactions and reconstruct the coordination logic. In their article discussing the adoption of BPM and SOA in a large Danish bank, Brahe (2007) state that developers too often need to repeatedly implement the same implementation patterns. Such a manual synchronization of changes between design and implementation is not an efficient development practice. Furthermore, the chance of wrongly designing the coordination logic grows, resulting into a less effective development practice as well.

[4]http://www.microsoft.com/biztalk

Summary of the key research challenges

Based on the previous discussions we list the most important challenges that come with the development of service-based systems:

- **Precise translation of the high-level business processes to technical implementations** without losing the connection between the abstract design level and the implementation level.
- **Managing the complexity of message-based implementations** and reducing the amount of time-consuming hand coding and low-level programming.
- **Avoiding an ad-hoc design** of coordination logic by using a systematic approach for constructing coordination scenarios that is based on best practices.
- **Using tools to support the design of coordination logic**, so that the construction of coordination scenarios is significantly simplified and semi-automated.
- **Increasing the reuse of existing building blocks** as much as possible so that a better consistency of multi-programmer implementations can be achieved and developers less often need to repeatedly implement the same implementation patterns.

As we will describe in the next section this thesis focuses on the design of coordination logic. Ultimately, the solutions presented need to address the challenges listed above as much as possible.

1.2 Research goal and questions

In this thesis, we address the lack of **rigorous and systematic guidelines on how to design appropriate coordination logic**. Such guidelines can significantly contribute to an efficient and effective design of coordination logic and service compositions, because it can support developers to more rapidly design coordination logic that is optimized to the business requirements. Ultimately, this contributes to flexible and dynamic service-based systems, because coordination logic can be designed in a more efficient and semi-automatic way and more easily adapted to changes in a business process. If we define a *coordination scenario* as a specific set of service interactions constituting the coordination logic in a service composition, we can formulate two research questions that we try to answer in this thesis:

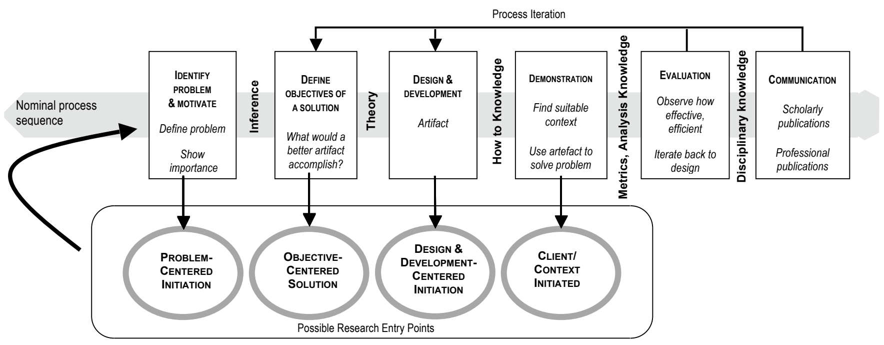

Process Iteration
Nominal process sequence
Identify problem & motivate
Define problem
Show importance
Inference
Define objectives of a solution
What would a better artifact accomplish?
Theory
Design & development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artefact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design & Development-Centered Initiation
Client/Context Initiated
Possible Research Entry Points

Figure 1.1: DSRM Process Model (Peffers et al., 2007)

RQ1 Can we come up with a *systematic way* of composing coordination scenarios, starting from some fundamental building blocks that can be combined to construct all possible scenarios?

RQ2 Can we provide service composers with a set of *design guidelines* for constructing a coordination scenario that complies to some predefined design criteria into account?

1.3 Research methodology

The research in this thesis can be classified as design-science research (Hevner, March, Park, & Ram, 2004). This type of research is fundamentally a problem-solving paradigm, which addresses research through the creation and justification of artifacts. Such artifacts are not exempt from natural laws or behavioral theories. On the contrary, their creation relies on existing theories that are applied, tested, modified, and extended through the experience, creativity, intuition, and problem solving capabilities of the researcher.

In order to answer the research questions described in section 1.2 we followed the Design Science Research Methodology (DSRM)(Peffers, Tuunanen, Rothenberger, & Chatterjee, 2007), which provides a process model consisting of six steps: problem identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation, and communication (see Figure 1.1).

1. **Problem identification and motivation:** In this step the specific research problem needs to be defined and the value of a solution should

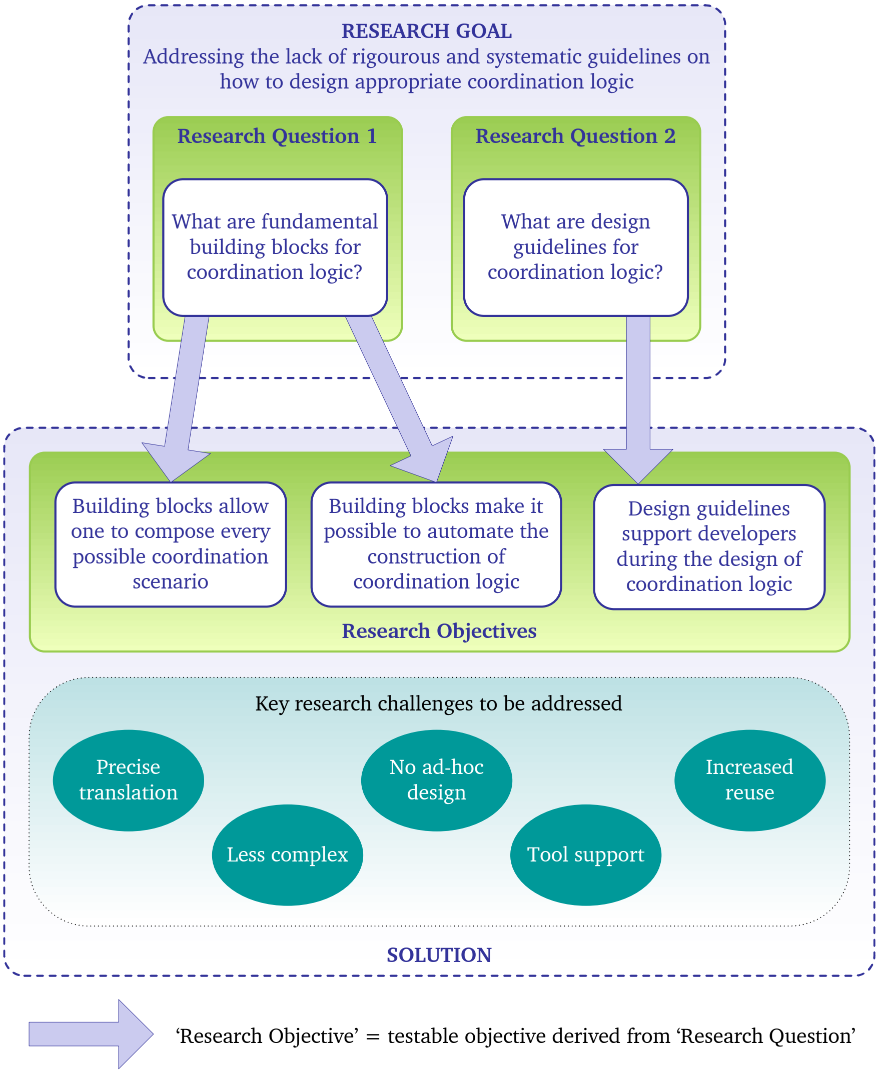

RESEARCH GOAL
Addressing the lack of rigourous and systematic guidelines on how to design appropriate coordination logic
Research Question 1
What are fundamental building blocks for coordination logic?
Research Question 2
What are design guidelines for coordination logic?
Building blocks allow one to compose every possible coordination scenario
Building blocks make it possible to automate the construction of coordination logic
Design guidelines support developers during the design of coordination logic
Research Objectives
Key research challenges to be addressed
Precise translation
No ad-hoc design
Increased reuse
Less complex
Tool support
SOLUTION
'Research Objective' = testable objective derived from 'Research Question'

Figure 1.2: The relationship between our research goal, research questions and research objectives

be justified. Furthermore, Peffers et al. (2007) advise to atomize the problem conceptually, because the problem definition will be used to develop an artifact that can effectively provide a solution. Resources required for this step include knowledge of the state of the problem and the importance of its solution.

Sections 1.1 and 1.2 already provide a clear description of the research context and problem. Since the main topic in our research is about coordination, it is necessary to have a good understanding of the concept of coordination. Coordination is not only in computer science an important research topic. It is also studied in disciplines such as organization theory, operations research, economics, linguistics, and psychology. Malone and Crowston (1994) studied the similarities and connections between the different flavors of coordination and created a more generic coordination theory. They define coordination as *managing dependencies* between activities. This definition is based on the intuitive idea that there is nothing to coordinate without any interdependence. In their work they state that coordination theories should try to characterize different kinds of dependencies and identify the coordination processes that can be used to manage them. Therefore, in Chapter 2, we will discuss several types of dependencies that can exist when composing services to service-based systems. As we will explain in that chapter, our research is targeted at two types of dependencies, namely sequence dependencies and data dependencies. Therefore, we atomize our research problem based on these two types of dependencies. This means we will first try to answer the research questions for both types of dependencies separately (see Chapters 3 and 4), before providing a complete solution to our research problem. Furthermore, in both the chapter on sequence dependencies and the chapter on data dependencies we first introduce a refined research context specific for the type of dependencies we are dealing with. This refined research context defines a terminology used throughout the presented solution. Additionally, it describes how we further decompose the specific problem (Vaishnavi & Kuechler, 2007), which lays the basis for our systematic way of constructing coordination scenarios and the fundamental building blocks for doing so.

Chapter 2 also highlights existing solutions in our problem domain and discusses why these solutions do not suffice.

2. **Definition of the objectives for a solution:** This step is about inferring the objectives of a solution from the problem definition and knowledge of what is possible and feasible.

   The general objective is that our solution provides rigorous and systematic guidelines on how to design appropriate coordination logic. More specifically, we derived *three research objectives* from the two research questions described in Section 1.2. These three research objectives are formulated in such a way that it can be tested whether the objective is achieved or not. In short, the three testable research objectives describe what we expect from a solution:

- By combining the building blocks in several ways, it should be possible to construct every potential coordination scenario. This objective is derived from the first research question (see Section 1.2).
- The solution should describe a set of fundamental building blocks for coordination logic. Based on these building blocks it should be possible to semi-automate the construction of a coordination scenario by letting developers pick specific building blocks for dependencies management and automatically generate a complete coordination from a business process specification. This objective is derived from the first research question (see Section 1.2).
- Our solution should guide developers to choose and combine the building blocks in a way that an optimized coordination scenario can be constructed. This objective is derived from the second research question (see Section 1.2).

In Figure 1.2 the relationship between our research goal, research questions and research objectives is shown.

3. **Design and development:** This step is the core of design science and is about the creation of the artifact. Conceptually, a design research artifact can be any designed object in which a research contribution is embedded in the design.

   In this thesis the artifact created is a set of patterns. In its simplest form a pattern can be defined as a *solution* to a *problem* that arises within a specific *context* (Buschmann, Henney, & Schmidt, 2007). Patterns originated as an architectural concept by Christopher Alexander. In his book *The Timeless Way of Building* (Alexander, 1979) he describes a pattern as follows:

   > *Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.*
   >
   > *As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.*

As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.

The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing.

The design of coordination logic is a recurring problem in service composition, which explains our choice for patterns. Furthermore, as we will describe in Chapters 3 and 4 an SOA creates a context in which several forces are present. As we will show, these forces can be used as evaluation criteria for constructing an optimized coordination scenario. Each pattern balances these forces differently, which brings us a bit closer to an answer to the second research question (see Section 1.2).

The patterns presented in this thesis can be classified as *design patterns*[5]. A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly recurring structure of communicating components (e.g. services in a service composition) that solves a general design problem within a particular context (Gamma, Helm, Johnson, & Vlissides, 1995).

The patterns presented in this thesis are the building blocks for constructing coordination logic. This means that the patterns presented do not exist in isolation. The patterns are supposed to complement each other and there should exist a way of combining the patterns in order to obtain a complete coordination scenario. In other words, the patterns together form a pattern language for constructing coordination logic, in which the patterns make the vocabulary of the language, and the rules for their implementation and combination make up its grammar (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996).

Similar to the approach taken for the discovery of the service interaction patterns (Barros, Dumas, & Hofstede, 2005), our patterns have been derived and extrapolated from insights into real-scale B2B transaction processing, BPEL (OASIS, 2007) and WS-CDL (W3C, 2005) examples in academic and industrial literature, generic scenarios identified in indus-

[5]The patterns presented in this thesis are intended to support the *design* of coordination logic. Important run-time issues such as message buffering, task execution scheduling and message reliability do not belong to the research scope of this dissertation.

try standards (e.g. RosettaNet Partner Interface Protocols (RosettaNet, n.d.)), and case studies reported in the literature.

4. **Demonstration:** The demonstration step is about showing the use of the artifact to solve one or more instances of the problem. This could involve its use in experimentation, simulation, case study, proof, or other appropriate activity.

   In this thesis we demonstrate the artifacts created in several ways:

   - Based on the patterns for managing sequence and data dependencies in service compositions, we have developed a tool for generating coordination logic from business process specifications (see Chapter 6).
   - We have applied the patterns for data dependencies management in both a fictive and real-life business case (see Subsections 4.3.3 and 4.4.2 in Chapter 4).
   - Workflow patterns are often used for the evaluation of business process modeling languages, because these patterns are considered as fundamental process constructs (Van der Aalst, Ter Hofstede, Kiepuszewski, & Barros, 2003). For a similar reason, we demonstrate the patterns for sequence dependencies management by showing that our patterns can be used to coordinate the sequence of activities specified in a workflow pattern. In particular, we have applied our patterns to the basic control flow patterns (see Chapter 3).

5. **Evaluation:** In the evaluation step researchers need to observe and measure how well the artifact supports a solution to the problem. This activity involves comparing the objectives of a solution to actual observed results from use of the artifact in the demonstration.

   First of all, we have implemented a tool that demonstrates that it is possible to semi-automate the construction of a coordination scenario (in the form of BPEL (OASIS, 2007) processes) by letting developers pick specific patterns for sequence and data dependencies management and automatically generate a complete coordination from a business process specification (in the form of a BPMN process (OMG, 2010a)). This implementation (presented in Chapter 6) also shows the practical utility of our pattern languages, which is an important evaluation aspect in design science (Hevner et al., 2004).

   In both the chapter on sequence dependencies (see Chapter 3) and the chapter on data dependencies (see Chapter 4) evaluation sections are included. In these sections it is shown that every possible coordination scenario can be constructed using the patterns presented (see Section 3.4.2 in Chapter 3 and Section 4.5 in Chapter 4). Furthermore, the chapter on data dependencies contains a real-life case study with a Belgian bank and insurance company, in which the patterns for managing data dependencies are applied (see Subsection 4.4.2 in Chapter 4).

6. **Communication:** In this step the researcher needs to communicate the research problem and its importance, the constructed artifact, its utility and novelty, the rigor of its design, and its effectiveness to other researchers and relevant audiences such as practicing professionals, when appropriate.

   In general, we have communicated our research in several ways. We have published several articles for presentation at an international conference. Preliminary versions of the pattern languages presented in this thesis were published in the following conference articles:

   Monsieur, G., Snoeck, M. & Lemahieu, W. (2009). A pattern language for service input data provisioning. In *Proceedings of the 16th Conference on Pattern Languages of Programs (PLoP 2009)*. New York, NY, USA: ACM.

   Monsieur, G., Snoeck, M. & Lemahieu, W. (2010). Managing sequence dependencies in service compositions. In *Proceedings of the 15th European Conference on Pattern Languages of Programs (EuroPLoP 2010)*.

   Additionally, we submitted a large part of Chapter 4 for publication in an international journal:

   Monsieur, G., Snoeck, M. & Lemahieu, W. (2010). Managing data dependencies in service compositions. Submitted for publication in *IEEE Transactions on Software Engineering*.

   Finally, the following publications were written during the preparation of this thesis:

   Monsieur, G., Snoeck, M. & Lemahieu, W. (2007). Coordinated Web Services Orchestration. In *Proceedings of IEEE 2007 International Conference on Web Services (ICWS 2007)* (pp. 775–783). Washington, DC, USA: IEEE Computer Society.

   Monsieur, G., De Rore, L., Snoeck, M. & (2008). Handling transactional business services. In *Proceedings of the 15th Conference on Pattern Languages of Programs (PLoP 2008)*. New York, NY, USA: ACM.

Monsieur, G. (2008). Gestructureerd bedrijfsprocessen implementeren (structured business process implementation). In *IT Professional*, 3(43), 26–27.

As explained before, the core part of our research, or the created artifacts, were written as patterns and pattern languages. This immensely facilitates the communication of our research. Although we have successfully applied the patterns to generate coordination logic from business process specifications, people are the prime audience for patterns. Patterns form a specialized but common vocabulary that software architects and developers can use to discuss particular problems that arise in their projects, resulting into a better joint understanding of specific problems and solutions to these problems (Buschmann et al., 2007). Furthermore, previous research has shown that patterns improve the repeatability, usability and reuse of design practices (Ng, Cheung, Chan, & Yu, 2006; Prechelt, Unger, Tichy, Brössler, & Votta, 2001).

1.4 Thesis outline

Chapter 2 presents an overview of research that is related to this dissertation. Based on the literature, it presents several types of dependencies that can exist when composing services to service-based systems. Subsequently, the chapter continues with a detailed discussion on existing approaches for managing these dependencies, followed by a description of existing patterns and pattern languages for building SOAs. The chapter concludes with a description of a few standards that are frequently used in the rest of this thesis. **Chapters 3 and 4** present the pattern languages for managing sequence and data dependencies, respectively. Subsequently, in **Chapter 5** it is described how these two pattern languages can be combined. **Chapter 6** provides an in-depth discussion on how these pattern languages are applied to build a tool for generating coordination logic from business process specifications. In the last chapter of this dissertation, **Chapter 7**, we draw several conclusions, point out limitations and present several challenges for further research.

If I have seen further it is by standing on the shoulders of giants

– **Isaac Newton** (1643–1727),
English physicist,
in a letter to Robert Hooke
in February 1676

2 Related work

The main topic of this thesis is coordination. As explained in our research methodology (see Section 1.3), we follow the definition by Malone and Crowston (1994), who define coordination as *managing dependencies* between activities. Therefore, this chapter starts with an overview of several types of dependencies that can exist when composing services to service-based systems (see Section 2.1).

In Section 2.2 we discuss several studies on managing sequence dependencies, while in Section 2.3 we present an analysis of current approaches for managing data dependencies.

As mentioned in our research methodology (see Section 1.3) our solution to the research problem described in Section 1.2 consists of a set patterns. Therefore, in Section 2.4 of this chapter we elaborate on several important patterns for Service-Oriented Architectures and how these patterns can potentially contribute to a solution that manages sequence and data dependencies in service compositions.

This chapter ends with a discussion on several related standards that will be frequently referenced and used in the rest of this dissertation (see Section 2.5).

2.1 Dependencies in service compositions

Multiple researchers have studied dependencies in the context of service composition (Janssen & Feenstra, 2008; Bhiri, Perrin, & Godart, 2005, 2006;

Yang, Papazoglou, & Van den Heuvel, 2002; Papazoglou, Delis, Bouguettaya, & Haghjoo, 1997).

Bhiri et al. (2005, 2006) have used dependencies to specify how (component) services are coupled and how the behavior of some given services influences the behavior of some others. In their view on inter-service dependencies they assume that a service's behavior can be modeled using a finite state machine. In particular, they assume that a service has a minimal set of states (*initial, active, aborted, canceled, failed* and *completed*) and transitions (*activate, abort, cancel, fail* and *complete*). When a service is instantiated (e.g. the service received a request for executing a business task), the state of the instance is *initial*. Then this instance can be either *aborted* or *activated*. Once it is *active*, the instance can normally continue its execution or it can be *canceled* during its execution. In the first case, it can achieve its objective and successfully *complete* or it can *fail*. Additionally, the service's transactional properties can be extended by adding a *compensated* state and *compensate* transition, which can be fired when the service is in the *completed* state. Based on this behavior model Bhiri et al. (2005, 2006) consider two classes of dependencies: activation dependencies and transactional dependencies. The activation dependencies class contains two types of dependencies:

Activation dependency There is an activation dependency between a service s_1 and a service s_2 if the completion of s_1 can fire the activation of s_2.

Abortion dependency There is an abortion dependency between a service s_1 and a service s_2 if the failure, the cancellation or the abortion of s_1 can fire the abortion of s_2.

In the transactional dependencies class a distinction is made between three types of dependencies:

Compensation dependency There is a compensation dependency from s_1 to s_2 if the failure or the compensation of s_1 can fire the compensation of s_2.

Cancellation dependency There is a cancellation dependency from s_1 to s_2 if the failure of s_1 can fire the cancellation of s_2.

Alternative dependency There is an alternative dependency from s_1 to s_2 if the failure of s_1 can fire the activation of s_2.

Janssen and Feenstra (2008) argue that the analysis of dependencies is necessary to create feasible service compositions and to identify alternative compositions. In their social-technical theory they identify two main classes of dependencies among component services: resource and link dependencies (Janssen & Feenstra, 2008).

Resource dependency There is a resource dependency between two services if both services use the same resource, which sets constraints on the execution order (i.e. a service should be consumed before another service can start).

Link dependency There is a link dependency between two services if one service depends in some way on the output of the other service (e.g. a service cannot continue its execution until another service has provided certain data)

Yang et al. (2002) and Papazoglou et al. (1997) make a distinction between two types of dependencies that can occur among service components:

Sequence dependency There is a sequence dependency (also referred to as a *commit dependency*) between a service s_1 and a service s_2 if the start *or* continuation of the execution of service s_2 depends on the completion of the execution of s_1.

Data dependency There is a data dependency between a service s_1 and a service s_2 if the start *or* the continuation of the execution of service s_2 depends on data that is provided by s_1.

In our research we use the two dependencies as proposed by Yang et al. (2002), because these two types are more general than the dependencies proposed by Bhiri et al. (2005, 2006) and Janssen and Feenstra (2008). The dependencies presented by Bhiri et al. (2005, 2006) are mostly sequence dependencies, while Janssen and Feenstra (2008) listed both sequence and data dependencies. A transactional dependency such as proposed by Bhiri et al. (2005, 2006) can be considered as a special form of sequence dependency. In general, a sequence dependency can be described as a situation in which one service needs to do something after something happened with another service. Hence, this definition also matches with transactional dependencies. For example, if one service fails (i.e. something happened), another service needs to execute a certain business task (i.e. do something).

Based on the distinction between sequence and data dependencies, we can identify two main coordination challenges that one faces when composing service-based systems. The first challenge is the management of sequence dependencies. This challenge consists of consuming all component services in the right order (i.e. according to sequence dependencies as possibly specified in a business process). For example, an order fulfillment process could have a sequence dependency between a payment service and a shipping service, because this business process specifies that the order can only be shipped after the payment is arranged. The second challenge is the management of data dependencies, which is about providing a service with all the data it needs. Consider for instance a data dependency between a shipping service and a customer relationship service, because the shipping service can only ship an order if it received the customer's shipping address, which can be provided by the customer relationship service. It should be mentioned that in this dissertation it is always assumed that a data dependency describes a situation in which one service needs data from second service, without the need for blocking the data in the first service. Obviously, each complete coordination scenario (for an order fulfillment process) needs to take into account both sequence and data dependencies.

In their Extended SOA Papazoglou (2005) and Papazoglou and Van den Heuvel (2007b) describe the *coordination* function that a composite service needs to perform as follows: *"controlling the execution of component services, and manage data flow among them and to the output of the component service (e.g. by specifying workflow processes and using a workflow engine for run-time control of service execution)"*. Controlling the execution of services and managing the data flow, exactly is what managing sequence and data dependencies is about.

Alonso, Casati, Kuno, and Machiraju (2004) consider six different dimensions of a service composition model. Among these are an *orchestration model*, which Alonso et al. (2004) define as a model that specifies the order in which services are to be invoked, and a *data and data access model*, which they define as a model that describes how data is specified and how it is exchanged between components. Hence, in an orchestration model it is specified how sequence dependencies are managed, while a data and data access model describes how data dependencies are dealt with.

2.2 Managing sequence dependencies

2.2.1 Introduction

Many developers use the terms *orchestration* and *choreography* to describe the business interaction protocols that control the execution of services (Papazoglou et al., 2007). Since managing sequence dependencies is about controlling the execution of services and designing the appropriate service interactions for accomplishing this, orchestration and choreography must be relevant terms when designing coordination logic for managing sequence dependencies.

However, orchestration and choreography are not always defined in the same way. Furthermore, some researchers introduce centralized or decentralized variants such as *centralized orchestration*, *decentralized orchestration*, *centralized choreography* and *decentralized choreography*. In addition, it is not always clear how these terms are related to terms such as *centralized coordination*, *decentralized coordination* and *peer-to-peer coordination*. In the same way, it is not always easy to understand the relations to concepts such as a *coordinator*, an *orchestrator* or a *choreographer*. To make things even more complex, there exist several synonyms or overlapping concepts such as *behavioral interface*, an *abstract process* in BPEL (OASIS, 2007) and an *executable process* in BPEL (OASIS, 2007).

In order to understand terms like orchestration and choreography (and related concepts) it is beneficial to have a set of basic concepts that are related to service composition in general, regardless of whether an orchestration- or a choreography-oriented description is used. In Subsection 2.2.2 we discuss such basic concepts. Subsequently, we use these concepts to discuss two groups of orchestration and choreography definitions that can be found in the literature (see Subsections 2.2.3 and 2.2.4).

2.2.2 A meta-model for service composition

As mentioned in the introduction of this section (see Subsection 2.2.1), in our discussion on orchestration and choreography (see Subsections 2.2.3 and 2.2.4) we distinguish between several orchestration and choreography definitions by using a set of basic concepts that are related to service composition in general. These concepts are explained in our meta-model for service composition design (see Figure 2.1). The purpose of this meta-model is only to illustrate common concepts in service composition design and their relationships using the UML class diagram notation (OMG, 2010b). It is not a core research contribution in this dissertation, but it helps to better understand and analyze current approaches for managing sequence dependencies. If one wants to use the meta-model in a broader context, more details and specifications would be required. For example, for several concepts (e.g. composition) it would be better to make a distinction between an abstract type and a concrete type.

The model in Figure 2.1 extends the meta-model proposed by Benatallah et al. (2005) by adding business process-related concepts such as business

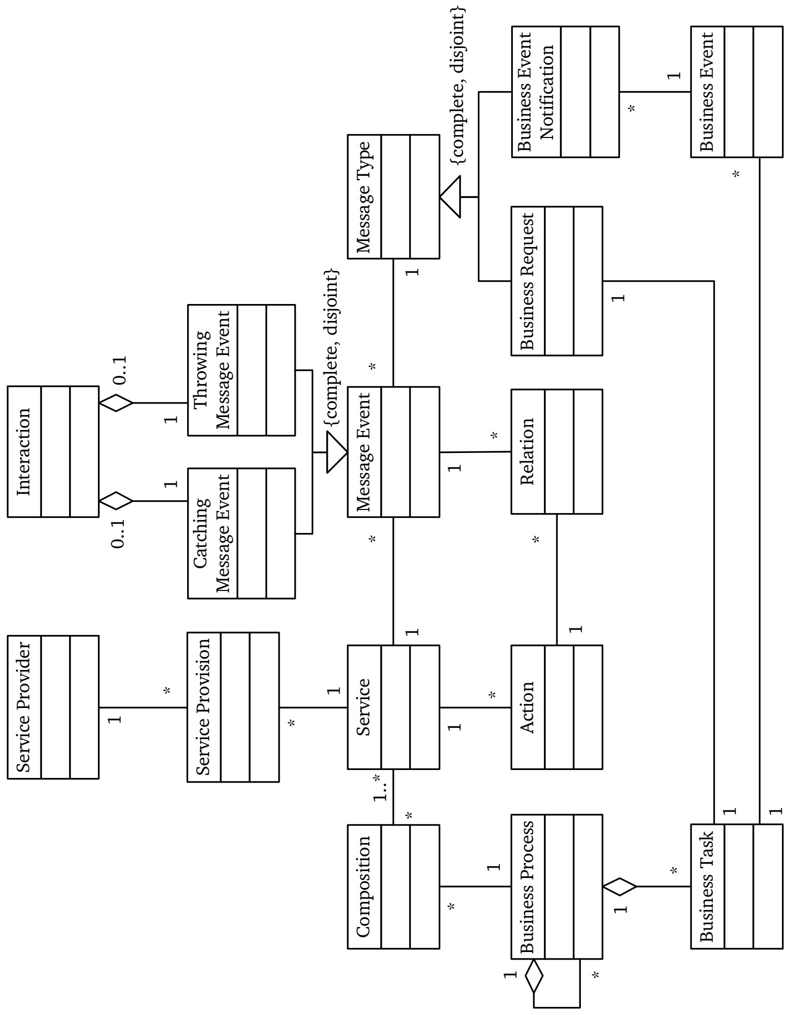


Figure 2.1: Basic concepts in service compositions

process, business task, business request and business event. A service *composition* supporting a *business process* consists of a number of (component) *services* that are provided by *service providers*. The same service can be provided more than once by different service providers (e.g. a flight booking service can be provided by different airlines). A service has several (internal) *actions* and is associated to several *message events* that are part of an *interaction* with other services. The interactions are message-based because this is the basic mechanism for interaction used in the mainstream service description languages (e.g. WSDL (W3C, 2001, 2007a)). Each interaction consists of a *catching message event* (receive) and a *throwing message event* (send). A message event is associated with a *message type*, which is either a *business request* or a *business event*. A *business request* - also referred to as a *service request* - means that the service sends (in case of a throwing message event) or receives (in case of a catching message event) a request for executing a certain business task in the business process. Business requests typically are defined in a *contract* between two parties in which is stated what the requester can expect from the party that receives the request (Weigand & Van den Heuvel, 2002). A *business event* means that the service sends or receives an event notification that describes a business task-related event (e.g. the end of a certain business task) (Hens, Snoeck, Poels, & De Backer, 2009). *Relations* relate actions and interactions to each other via message events. The interactions (including the message events) form the coordination logic in a service composition.

In this meta-model abstraction is made of the sequence constraints between interactions. Furthermore, the meta-model does not take into account the transactional aspect of coordinating business processes and tasks. In Chapter 7 we briefly discuss how interactions for transactional purposes can be integrated in this meta-model (see Subsection 7.2.2).

2.2.3 Orchestration and choreography as composition viewpoints

A first group of researchers consider orchestration and choreography simply as different *viewpoints* in a service composition. The most detailed definitions in this category are proposed by Barros, Dumas, and Oaks (2006). Therefore, we will summarize their definitions below and indicate the relationship with the meta-model presented in Subsection 2.2.2. Subsequently, we highlight similar definitions by other researchers.

Barros et al. (2006) define a choreography (model) as follows:

- *"A description of a collaboration between a collection of services to achieve a common goal.*
- *"It captures the interactions in which the participating services engage to achieve this goal and the dependencies between these interactions, including the causal and/or control-flow dependencies (i.e. that a given interaction must occur before another one, or that an interaction causes another one)"*
- *"The interactions are captured from a global perspective meaning that all participating services are treated equally. In other words, a choreography encompasses all interactions between the participating services that are relevant with respect to the choreography's goal."*
- *"It does not describe any internal action of a participating service that does not directly result in an externally visible effect, such as an internal computation or data transformation."*

Hence, choreography is about the design of all interactions, including the message events, in a service composition. In a service composition component services are collaborating together to achieve a common goal, which is the realization of a new composite service. A choreography does not describe the relationship between these interactions and the internal actions of the component services.

Barros et al. (2006) define an orchestration (model) as *"a description of the communication actions and the internal actions in which a service engages. Internal actions include data transformations and invocations to internal software modules."* Thus, orchestration is about designing all (internal) actions and related message events and interactions in which a *particular* component service is involved.

The above definitions of orchestration and choreography fit well in several business-to-business service composition methodologies, in which both orchestration and choreography (as viewpoints) need to be combined in order to build a service-based application that supports the inter-enterprise business process. The main idea in these methodologies (Papazoglou et al., 2007; Papazoglou & Van den Heuvel, 2007a; Peltz, 2003) is that companies involved in the business process first agree upon on the way they collaborate by designing and specifying a choreography. Subsequently, each participating company can develop an orchestration that fits into the choreography. In practice companies first design a choreography (e.g. in a WS-CDL (W3C, 2005) or BPEL4Chor (Decker, Kopp, Leymann, & Weske, 2009) representation) and then generate a sort of orchestration skeleton for each company (e.g. in an abstract BPEL process (OASIS, 2007)). Barros et al. (2006) refer to this orchestration skeleton as a behavioral interface:

- *"A behavioral interface captures the behavioral aspects of the interactions in which one particular service can engage to achieve a goal."*
- *"It focuses on the perspective of one single party (in a choreography). As a result, a behavioral interface does not capture 'complete interactions' since interactions necessarily involve two parties."*
- *"Basically, it consists of communication actions performed by one participant"*
- *"A behavioral interfaces does not describe internal tasks such as internal data transformations.*

Note that these definitions imply that in each service composition there is only one choreography viewpoint, but several orchestration viewpoints.

2.2.4 Orchestration and choreography as composition styles

A second group of researchers describe orchestration and choreography as composition *styles*, which are two generic ways that illustrate how component services in a service composition interact. This means that orchestration and choreography can be considered as two *templates* for coordination scenarios that manage sequence dependencies.

Several researches define orchestration as a service composition in which a central coordinator - sometimes also referred to as the orchestrator (Busi, Gorrieri, Guidi, Lucchi, & Zavattaro, 2005; Tabatabaei, Kadir, & Ibrahim, 2008; Beek, Bucchiarone, & Gnesi, 2006), the controller (Barker, Weissman, & Hemert, 2009) or the choreographer (Mitra, Kumar, & Basu, 2008) - is responsible for coordinating the business process execution by invoking all component services in the right order (Busi et al., 2005; Barker et al., 2009; Tabatabaei et al., 2008; Malinova & Gocheva-Ilieva, 2008; Beek et al., 2006; Bellini, Prado, & Zaina, 2010). This coordinator can be either one of the component services or another service (e.g. a BPEL engine) (Malinova & Gocheva-Ilieva, 2008). Pedraza and Estublier (2009) describe orchestration as a service composition in which *"a single engine on a single machine forms the heart of the (service-based) system, with all communication going to and coming from that machine"*. Similarly, Pessoa, Silva, Sinderen, Quartel, and Pires (2008) state that all the interactions in an orchestration pass through the coordinator.

In summary, this means that orchestration can be considered as a service composition style in which one service (the coordinator) interacts with all component services. There are no (explicit) interactions between component services; each interaction in the orchestration is between the coordinator and a component service. Typically, only the coordinator has knowledge of the business process and sends business requests to the component services. The component services mostly send business events to the coordinator. In the literature, orchestration defined in this way is also referred to as *centralized coordination* (Pessoa et al., 2008; Benatallah et al., 2003), *centralized orchestration* (Binder, Constantinescu, & Faltings, 2006) or *centralized choreography* (Mitra et al., 2008). This definition can also be explained by means of the orchestra metaphor. As in a real-life orchestra, one service is playing the role of the conductor and coordinates all component services (Lin & Chang, 2005).

In line with orchestration as a composition style, choreography is often described as a service composition in which there is *no* central coordinator. In a choreography the component services are rather collaborating together. The business process is executed and coordinated by several peer-to-peer interactions among the collaborating services (Busi et al., 2005; Barker et al., 2009; Tabatabaei et al., 2008; Malinova & Gocheva-Ilieva, 2008; Beek et al., 2006; Bellini et al., 2010). The component services directly communicate with each other, and not through a central coordinator (Pedraza & Estublier, 2009; Pessoa et al., 2008). According to Barker et al. (2009) and Malinova and Gocheva-Ilieva (2008) all participants in a choreography need to be aware of the business process, operations to execute, messages to exchange, and the timing of message exchanges. In contrast, Pedraza and Estublier (2009) state that each participant is responsible for routing messages to the 'next' (Web) service(s) *without* any global view such as the business process. In summary, choreography can be considered as a service composition style in which the component services interact with each other and there is no central coordinator that exclusively interacts with component services. Typically, several component services have knowledge about (part of) the business process and send business requests to each other; component services mostly also send business events to other component services. In the literature, choreography defined in this way is also referred to as *peer-to-peer coordination* (Pessoa et al., 2008; Benatallah et al., 2003) or *decentralized orchestration* (Binder et al., 2006). This definition can also be explained by means of the dancers metaphor. As a group dancers do in real life, each service knows exactly when to execute and with whom to interact (Lin & Chang, 2005).

2.2.5 Conclusion

As discussed in the introduction of this section, orchestration and choreography are important terms when designing coordination logic that deals with sequence dependencies. Based on our literature overview (see Subsections 2.2.3 and 2.2.4) we can conclude that orchestration and choreography are relevant terms in two ways. On the one hand, orchestration and choreography can be considered as viewpoints in a service composition. In particular, one can say that orchestration and choreography provide local and global views on a coordination scenario. On the other hand, orchestration and choreography can refer to two composition styles. In the context of this thesis, this perspective is probably the most relevant, because in a way these composition styles refer to different kinds of coordination scenarios.

We believe both composition viewpoints and styles are useful concepts when designing service interactions. However, as shown in the previous subsections the terms orchestration and choreography are used for both purposes, which can potentially lead to confusion and inconsistencies. In this thesis, therefore, we consider orchestration and choreography only as viewpoints in a service composition. When referring to the composition styles, we prefer the terms *centralized* and *decentralized coordination*. Terms such as *centralized orchestration* or *decentralized orchestration* are very confusing when orchestration is considered as a viewpoint, because then orchestration is, per definition, always executed by one service. *Centralized choreography* or *decentralized choreography* are perhaps easier to understand, because then the adjectives tell something about the interactions. However, these terms, currently, are not so frequently used in the literature.

Note that it is relatively easy to understand that people confuse composition viewpoints with composition styles. Indeed, from the viewpoint perspective orchestrations are always executed by one service. However, it is important to realize that this does not necessarily imply that the service executing the orchestration have to coordinate the complete business process. Similarly, it is true that in a choreography (as viewpoint) there are several services interacting with each other, but this does not imply a centralized or decentralized coordination. Indeed, also in a centralized coordination one can consider the choreography viewpoint.

As mentioned earlier, in this thesis we are particularly interested in composition styles, because centralized or decentralized coordination describes a sort of coordination style. Nevertheless, orchestration and choreography as composition styles does not explicitly describe the types of messages that are exchanged between services (i.e. business requests or business event notifications). As we have described in Subsection 2.2.4, centralized coordination (or orchestration as composition style) typically implies that one coordinating service has business process knowledge and sends *business requests* to the component services. However, one could also think of a coordination scenario in which a central business process-agnostic service (e.g. an event broker) does not send business requests but only forwards *business event notifications* to interested services. Similarly, a choreography as composition style does not describe the nature of inter-service communication. On the one hand, one can easily imagine scenarios in which component services collaborate together and send business requests to each other. On the other hand, however, services can also collaborate together by only sharing business events. For example, in a group of dancers typically dancers do not tell each other what and when to do something (i.e. they do not send business requests to each other). Dancers rather react on movements etc. from others to know when to do something. In summary, we can conclude that the common distinction between orchestration and choreography does not describe the complete set of possible coordination scenarios. However, currently, there are no concrete guidelines available for building such scenarios and choosing an appropriate solution.

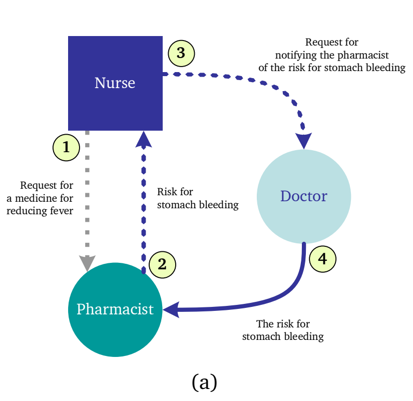
2.3 Managing data dependencies

In this section we will discuss existing solutions for managing data dependencies in service compositions. Before discussing these studies in detail, we first describe a small running example that we use for illustrating the results of these studies.

2.3.1 Running example

We use a simple (fictive) example of a service composition in hospitals as running example. We deliberately chose a non-automated example so as not to clutter the discussion with (low-level) implementation issues, but the conclusions presented in this section are equally applicable to software services.

Consider a nurse who wants to treat a patient's high fever using a febrifuge. Getting the right febrifuge requires consumption of several services. In particular, the nurse should request a febrifuge from the pharmacist, who of course also provides several services to the hospital staff. Hence, the nurse can be considered as a service composer that needs to consume the service of a pharmacist. Both aspirin and paracetamol are fever reducers. However,


Nurse
3
Request for notifying the pharmacist of the risk for stomach bleeding
1
Request for a medicine for reducing fever
Risk for stomach bleeding
Doctor
2
4
Pharmacist
The risk for stomach bleeding
(a)

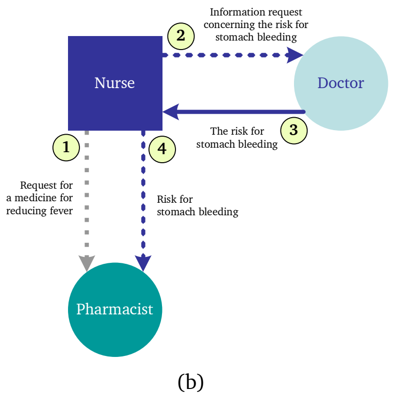

Information request concerning the risk for stomach bleeding
2
Nurse
Doctor
The risk for stomach bleeding
3
1
4
Request for a medicine for reducing fever
Risk for stomach bleeding
Pharmacist
(b)

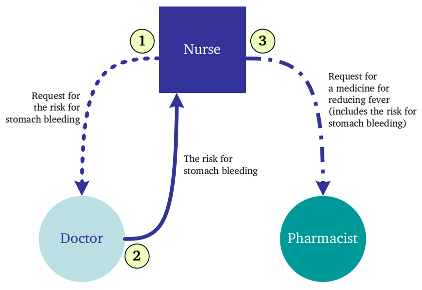
Figure 2.2: Two ways of coordinating the pharmacist and doctor

aspirin has the unpleasant side effect that it can cause stomach bleeding in certain circumstances. Therefore, the pharmacist needs patient-specific information concerning the risk for stomach bleeding, before he or she can deliver an appropriate febrifuge. The risk for stomach bleeding is only known by the patient's doctor. This means that the doctor provides a second service that needs to be consumed in order to support the task of choosing a febrifuge. Hence, this implies that there is a data dependency in this service composition between the pharmacist and the doctor. In this case, service coordination means providing the pharmacist with the data that is held by the doctor.

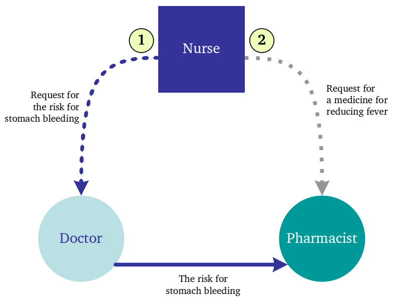
Even though this is a rather small example of data dependencies in a service composition, many coordination scenarios are possible. In figures 2.2(a) and 2.2(b) two ways of managing the data dependency between the pharmacist and the doctor are shown.

2.3.2 Existing techniques for dealing with data dependencies

Data dependencies are related to the *data flow* concept in service compositions. In general, data flow can be defined as the service interactions that are necessary for sending data from one service that can provide certain data to another service that needs that data (Barker, Weissman, & Van Hemert, 2008b; Yang, 2003; Weber, Schuler, Neukomm, Schuldt, & Schek, 2003; Charfi & Mezini, 2007). A data flow thus specifies how data dependencies are managed. Therefore, we will focus on studies that contain approaches



(a) Central data flow



(b) Decentral data flow

Figure 2.3: Two possible data flows for hospital example

to find and use alternative data flows in a service composition. We illustrate the results of these studies by means of the running example presented in Subsection 2.3.1. Subsequently, we will show that these studies do not cover all aspects that are important for the design of an appropriately coordinated service-based system.

In the descriptions below, we use two coordination scenarios for the hospital example, which are represented in figures 2.3(a) and 2.3(b). Each arrow corresponds to a message sent between two entities. The dashed arrows refer to service invocations, while the solid arrows denote the transfer of data between two entities. The semi-dashed arrow (as used in figure 2.3(a)) is used to indicate that the data is included in the invocation message. While in figure 2.3(a) all data passes via the nurse (central data flow), the data flow in figure 2.3(b) is decentral, since data flows directly from one service to the other. As we will show below, the contributions of many studies can be easily explained by means of this small example consisting of two possible coordination scenarios.

Barker et al. (2008b) and Barker, Weissman, and Van Hemert (2008a) have presented a Web services based architecture that allows centralizing component invocations (centralized control flow) and decentralizing data flows (similar to figure 2.3(b)). This architecture consists of a centralized orchestration engine that issues control flow messages to Web services taking part in service composition. However, enrolled Web services can pass data messages among themselves, as in a peer to peer model. The architecture is mainly based on the idea of so called *proxies*, which are deployed in the vicinity of Web services. These proxies realize the more efficient data flow between component services.

Liu, Law, and Wiederhold (2002a, 2002b) have published a mathematical model that is built to compare the data flow performances. They concluded that decentralized data flow is in general superior in performance (i.e. the service composition in figure 2.3(b) outperforms 2.3(a)). Subsequently, they developed a Flow-based Infrastructure for Composing Autonomous Services (FICAS) (Liu et al., 2002a, 2002b). Autonomous services are built to support the service access protocol, which enforces the explicit separation of data flows from control flows. In FICAS the so called autonomous services are implemented by wrapping each software application or service into an autonomous service with a mediator.

The infrastructure based on so called service invocation triggers, introduced by Binder et al. (2006), is very similar to FICAS. In this infrastructure service invocation triggers also act as proxies for individual service invocations. Triggers collect the required input data before they invoke the service. Moreover, they forward service outputs to exactly those services that need the output. In order to make use of triggers, business processes are decomposed into sequential fragments, and the data dependencies are encoded within the triggers. Once the trigger of the first service in a business process has received all input data, the execution of that service is started and the outputs are forwarded to the triggers of subsequent services. Consequently, the service composition is implemented in a fully decentralized way, the data is transmitted directly from the producer to all consumers.

Balasooriya, Padhye, Prasad, and Navathe (2005) use the same ideas for decentralizing data flows. In particular, they create a proxy wrapper around each Web service. The proxy wrappers embed the coordination logic so that instances of wrapped Web services become stateful self-coordinating web objects. However, the proxy wrappers need to interact with the actual Web service to complete each method invocation.

2.3.3 Conclusion

We can conclude that several approaches exists that cater for alternative data flows. Many studies propose architectural infrastructures for such data flows. These infrastructures often use the same idea: wrapping each component service with additional logic that decides where to send input or output data. Obviously, these infrastructures are valuable and useful when one wants to implement a specific data flow. However, the focus on the problem of designing the data flow itself is rather limited. Furthermore, as we will show below, it remains difficult for a service composer to construct a well coordinated service composition. There are two main reasons why the current approaches are not entirely adequate for this purpose:

1. As most approaches have only a limited focus on designing the data flow, these studies fail to systematically analyze the coordination problem. Most approaches allow finding alternative data flows, but do not provide a systematic way of building different coordination styles nor do they analyze the advantages and disadvantages of alternatives. As a consequence, they fail to exhaustively identify all coordination scenarios. The approaches mostly propose techniques for decentralizing data flows in service compositions. Applied to the hospital example, this would mean that a scenario such as shown in figure 2.3(a) can be transformed into a scenario such as shown in figure 2.3(b). However, one can easily see that there are more possibilities. For example, the scenario represented in figure 2.4 contains a different coordination scenario. In this scenario the pharmacist requests and receives the risk information directly from the doctor, which can be considered as yet another different way of managing the data dependency between the pharmacist and the doctor. Other possible scenarios were shown in the subsection discussing the running example (see figures 2.2(a) and 2.2(b)).

2. The main motivation behind existing approaches are performance issues (i.e. communication overhead, etc.). Only the work by Balasooriya et al. (2005) recognizes that decentral data flow can be required due to security, privacy, or licensing imperatives. However, when evaluating their infrastructure, they only focus on the performance aspect. To the best of our knowledge, no studies about data dependency management take into account other aspects that could influence the choice of a specific data flow such as data confidentiality, loose coupling or robustness to change. This can result in badly or suboptimally coordinated service compositions and service-based systems. For example, the pharmacist and doctor in the decentralized data flow scenario shown in figure 2.2(b) are not optimally coordinated. This is due to the fact that nurses probably should not need to understand which data is required by the pharmacist. Nurses simply want to consume the pharmacist's services, and it is to be avoided that changes in data requirements on behalf of the pharmacist result in changes in how nurses need to work (or consume the pharmacist's services). Hence, the scenarios represented in figures 2.2(a) and 2.4 are probably more appropriate, because in these scenarios the nurse does not have to know which data is needed by the pharmacist. This example illustrates that robustness to change is another useful criterion to be considered next to performance issues.

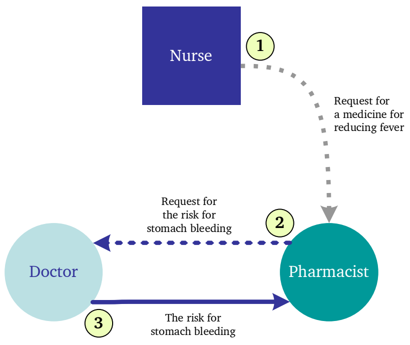


Figure 2.4: An alternative data flow for the scenarios represented in figures 2.3(a) and 2.3(b)

2.4 Patterns for Service-Oriented Architectures

In the two previous sections (Sections 2.2 and 2.3) we have discussed several existing approaches to manage sequence and data dependencies in service compositions. The approach presented in this thesis is described in the form of a set of patterns. Therefore, in this section we discuss several important patterns for Service-Oriented Architectures. In this discussion we focus on how these patterns can potentially contribute to a solution that manages sequence and data dependencies in service compositions.

2.4.1 Service interaction patterns

Barros et al. (2005) have proposed a set of *service interaction patterns*. They identified four groups of patterns. The first group encompasses *single-transmission bilateral* interaction patterns. These correspond to elementary interactions where a party sends (receives) a message, and as a result expects a reply (sends a reply). The second group of patterns consists of *single-transmission multilateral* interactions. In this case, a party may send or receive multiple messages but as part of different interaction threads dedicated to different parties. The third group of patterns is dedicated to *multi-transmission* interactions, where a party sends (receives) more than one message to (from) the same party. The fourth group consists of *routed* interactions, which means that the receiver of the ”response” is not the same as the sender of the request.

Barros et al. (2005) aim to consolidate recurrent interaction scenarios in orchestrations and choreographies, and abstract them in a way that provides reusable knowledge. Furthermore, the service interaction patterns are intended for assessing an orchestration or choreography language for its interaction modeling capabilities. In the past such evaluations were conducted for BPEL (Barros et al., 2005), WS-CDL (Decker, Overdick, & Zaha, 2006), BPMN (Decker & Puhlmann, 2007; Decker & Barros, 2008) and BPEL4Chor (Decker, Kopp, Leymann, & Weske, 2007).

Although the service interaction patterns may be composed through operators expressing flow dependencies (e.g. sequence, choice, etc.) (Barros & Börger, 2005), no guidelines exist on how to combine the patterns to construct coordination logic that manage sequence and data dependencies.

2.4.2 Pattern-based architectural framework for Service-Oriented Architectures

Zdun, Hentrich, and Van der Aalst (2006) have proposed a pattern-based architectural framework for SOAs. In their reference architecture Zdun et al. (2006) adopted software patterns from several sources that were described originally in a number of different domains such as remoting, messaging, resource management, networked and concurrent objects, software architecture, component integration, object-oriented design, e-business, process-driven architectures, business objects, and workflow systems. Their major contribution is the domain-specific combination of these patterns - in the SOA domain.

In the reference architecture by Zdun et al. (2006) the coordination logic in service compositions is referred to as the *process integration logic*. Since both a business process and coordination logic are represented using process flows, Zdun et al. (2006) propose to make a distinction between two general types of process flows: *macroflow* representing the higher-level business process, and microflow addressing the process flow within a macroflow activity. The distinction between micro- and macroflow is a conceptual decision in order to be able to design process steps at the right level of granularity when designing at the long running business process level (macroflow) or the short running, more technical level (microflow) (Hentrich & Zdun, 2006). Typically, a microflow consists of coordinated service interactions. However, no patterns were referenced for constructing such service interactions systematically, nor were patterns referenced for different sorts of microflow (i.e. templates or coordination styles).

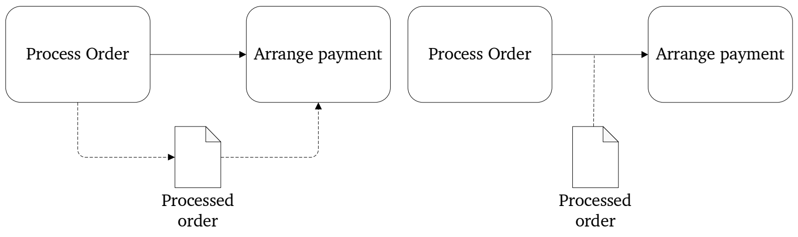
2.4.3 Flexible coordination of service interaction patterns

Zirpins, Lamersdorf, and Baier (2004) propose to make a distinction between the *logical dependencies* that are modeled by the interaction logic and the *operational coordination* that refers to the procedure or method that is utilized to enforce the logical dependencies. In Section 2.1 we identified logical dependencies in the form of sequence and data dependencies. Similarly, the operational coordination matches our vision on coordination, which is about managing the sequence and data dependencies. Zirpins et al. (2004) argued that while workflow processes represent the logical dependencies of interactions (i.e. causal and data relationships of message exchanges) they often simultaneously act as instructions for their coordination on the execution-level by distributed workflow management systems. As such, the coordination procedure emerges only implicitly as a side-effect of dependencies from the interaction logic and not because of application-specific reasons.

However, there are in most cases multiple alternatives for the enforcement of the abstract interaction logic. A reason for this is the multiplicity of possibilities for splitting the dependencies of the interaction logic into different partitions as well as the variety of alternatives for delegating parts of a partition to executive organizations for operational coordination. Therefore Zirpins et al. (2004) suggest that a technical solution for service composition should consist of a combination of design and implementation patterns. A *design pattern* corresponds to the interaction logic that only specifies the generic process characteristics, while an *implementation pattern* refers to the refinement of the interaction logic that is needed for the concrete coordination of services.

Zirpins et al. (2004) state that the criteria for the choice of the most appropriate coordination pattern must be specified by so called coordination policies. A *coordination policy* describes the effect of a coordination variant in terms of specific (non-functional) service properties and thereby controls the choice of alternatives.

In summary, we can conclude that in this thesis we are looking for both implementation patterns and coordination policies (Zirpins et al., 2004; Zirpins & Lamersdorf, 2004). The implementation patterns allow us to systematically construct coordination scenarios, and the coordination policies make it possible to construct the most appropriate coordination scenario in a certain business context.



(a) Two directed data associations (b) One undirected association

Figure 2.5: Two ways of modeling that the output data of a certain task is the input for another task in BPMN (OMG, 2010a)

2.5 Related standards

In this section we describe several standards that are strongly related to service composition and coordination. Moreover, in the rest of this dissertation these standards will be frequently referenced and used.

2.5.1 Business Process Modeling Notation (BPMN)

The Object Management Group (OMG) has developed a standard Business Process Modeling Notation (BPMN) (OMG, 2010a). The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes (e.g. service-based systems), and finally, to the business people who will manage and monitor those processes.

In BPMN one can model both sequence and data dependencies. The latter type of dependencies can be modeled using so called *data objects* that can be associated to BPMN tasks as *data input* or *data output*. Furthermore, it is possible to link an activity's *data output* to another activity's *data input* to indicate that the output data of a certain task is the input for another task (e.g. see Figures 2.5(a) and 2.5(b)). Additionally, it is possible to model that data from outside the process (i.e. data that is *not* the result of a certain task) is the input for a certain task (e.g. see Figure 2.6).

However, currently, BPMN does not provide language constructs for modeling associations between data objects and sequence flow conditions. This implies that it is not possible to model that certain data is required for deciding whether a certain task should be executed or not. In other words, when

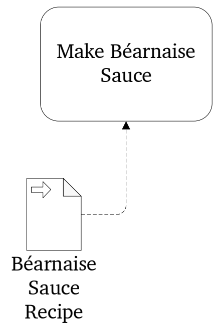


Figure 2.6: Modeling external data as input data in BPMN (OMG, 2010a)

using a data-based gateway to model such a decision, it can not be specified that certain data objects are required in order to decide which task needs to be executed (e.g. see Figure 2.7). When one draws an undirected association between a data-based gateway and a data object (see Figure 2.8), the BPMN specification does not provide a definition describing what this would mean.

The BPMN specification comes together with an XML schema (W3C, 2004), which allows to serialize BPMN models and construct automated model transformations (e.g. generation of coordination scenarios for a specific BPMN business process, see Chapter 6).

2.5.2 Web Services Description Language (WSDL)

The Web Services Description Language (WSDL) provides a model and an XML format for describing Web services (W3C, 2001, 2007a). A WSDL document separates the description of the abstract functionality offered by a Web service from concrete details of a Web service description such as ”how” and ”where” that functionality is offered.

At an abstract level, WSDL describes a Web service in terms of the *messages* it sends and receives. Typically, these messages are defined using XML Schema (W3C, 2004). An *operation* describes an interaction that the Web service supports (e.g. request/response) and specifies which messages are exchanged in that interaction. A *port type* (in WSDL version 1.1 (W3C, 2001)) or an *interface* (in WSDL version 2.0 (W3C, 2007a)) groups operations together without any commitment to transport or wire format.

At a concrete level, a *binding* specifies transport (e.g. HTTP (Fielding et al., 1999)) and wire format (e.g. SOAP (W3C, 2000)) details for one or more

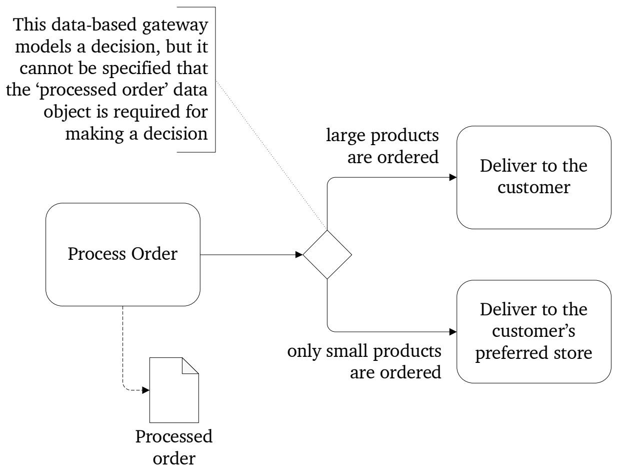

This data-based gateway models a decision, but it cannot be specified that the 'processed order' data object is required for making a decision
large products are ordered
Deliver to the customer
Process Order
Deliver to the customer's preferred store
only small products are ordered
Processed order

Figure 2.7: Modeling that data objects are required to make a decision in BPMN (OMG, 2010a)

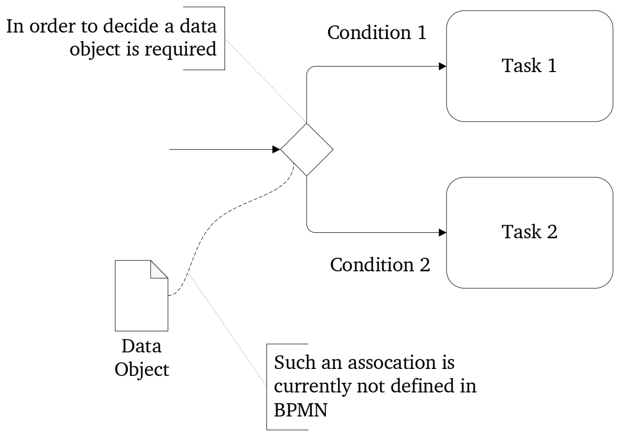

In order to decide a data object is required
Condition 1
Task 1
Task 2
Condition 2
Data Object
Such an assocation is currently not defined in BPMN

Figure 2.8: An undefined way of modeling that data objects are required to make a decision in BPMN (OMG, 2010a)

port types or interfaces. A *port* (in WSDL version 1.1 (W3C, 2001)) or an *endpoint* (in WSDL version 2.0 (W3C, 2007a)) associates a network address with a binding. Finally, a *service* groups together ports or endpoints.

2.5.3 Business Process Execution Language (BPEL)

The Business Process Execution Language (BPEL) is the most widely used standard for specifying Web service orchestrations (Van der Aalst et al., 2005; Papazoglou, 2007; Nitzsche, Lessen, Karastoyanova, & Leymann, 2007). An *executable* BPEL process defines how multiple Web service interactions are coordinated to achieve a business goal, as well as the state and the logic necessary for this coordination (OASIS, 2007). In terms of our service composition concepts (see Subsection 2.2.2) this means that an executable BPEL process both defines service interactions, internal actions and relations between these interactions and actions. In an *abstract* BPEL process only the service interactions are defined. In other ways, an abstract BPEL process defines a behavioral interface (see Subsection 2.2.3).

Since BPEL is mainly intended for defining service interactions (from the perspective of one Web service), a set of BPEL processes can be used to describe a specific coordination scenario that manages sequence and data dependencies.

The BPEL specification comes together with an XML schema (W3C, 2004), which allows to serialize BPEL models and construct automated model transformations (e.g. generation of BPEL-based coordination scenarios for a specific business process, see Chapter 6).

The Belgian federal government is like an orchestra without conductor

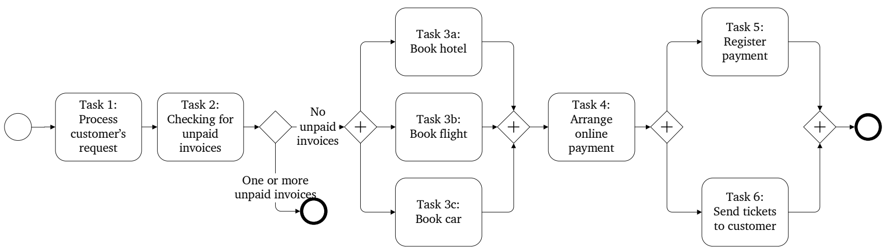
– **Tim Pauwels** (1971),
Belgian political journalist,
September 11th, 2010

3 Managing sequence dependencies

In this chapter we present our pattern language that can support service composers when designing coordination logic that manages sequence dependencies in a service composition. The chapter starts with a concrete example that shows the problem of managing sequence dependencies (see Section 3.1). Subsequently, in Section 3.2 the different patterns are presented. In Section 3.3 we discuss how these patterns can be applied in practice. Then follows a short discussion on the validation of the patterns (see Section 3.4). Finally, the chapter ends with a short conclusion (see Section 3.5).

3.1 Introductory example

In Figure 3.1 a travel agency business process is represented using the Business Process Modeling Notation (BPMN) (OMG, 2010a). The main goals



Task 1: Process customer's request
Task 2: Checking for unpaid invoices
No unpaid invoices
One or more unpaid invoices
Task 3a: Book hotel
Task 3b: Book flight
Task 3c: Book car
Task 4: Arrange online payment
Task 5: Register payment
Task 6: Send tickets to customer

Figure 3.1: A travel agency business process

in the business process are the reservation of a hotel room, a flight and a rental car. As shown in Figure 3.1 the first task (task 1) is about processing a customer's request. It is only after this task that the travel agency knows for which hotel, flight and car it needs to make reservations. In the next task it is checked whether or not the customer has still unpaid invoices (task 2). If this is not the case, the business process continues with the real booking of the desired hotel, flight and car (task 3a, 3b and 3c). These three booking tasks can occur in parallel. Once all three reservations are completed an online payment should be arranged (task 4). Finally, the business process ends by registering the payment (task 5) and sending the tickets to the customer (task 6).

In this example we assume that the business process represented in Figure 3.1 can be supported by seven services: Customer Service (task 1), Finance Service (task 2 and 5), Hotel Booking Service (task 3a), Flight Booking Service (task 3b), Car Rental Booking Service (task 3c), Online Payment Service (task 4) and Mail Service (task 6). Hence, coordination logic for this business process consists of interactions with these seven services. These service interactions should manage the sequence dependencies as specified in the business process (e.g. a request to the Mail Service can only be sent when the Online Payment Service had successfully processed the customer's payment).

In Figures 3.2, 3.3, 3.4 and 3.5 different coordination scenarios for the business process represented in Figure 3.1 are shown using iBPMN (Decker & Barros, 2008; Decker, 2009). iBPMN is an extension of BPMN (OMG, 2010a) aimed at modeling choreographies. While BPMN allows to connect interface behavior models, iBPMN extends BPMN so that choreography designers can fully model interactions between services, including sequences of interactions etc.

In a service-oriented environment, business processes often are implemented by coordinating services centrally using a process engine. This means that there is a central coordinating service that manages all sequence dependencies and (simply) interacts with all services in the order that is specified in the business process. For example, in Figure 3.2 one can see that the independent coordinator firstly interacts with the Customer Service (task 1), before it interacts with the Finance Service (task 2). Both the coordination scenarios in Figure 3.2 and Figure 3.3 consist of one service that centrally coordinates the other services (i.e. centralized coordination as described in Subsection 2.2.5 of Chapter 2). While Figure 3.2 contains a separated independent coordinator, in Figure 3.3 the Customer Service takes the role of central coordinator.

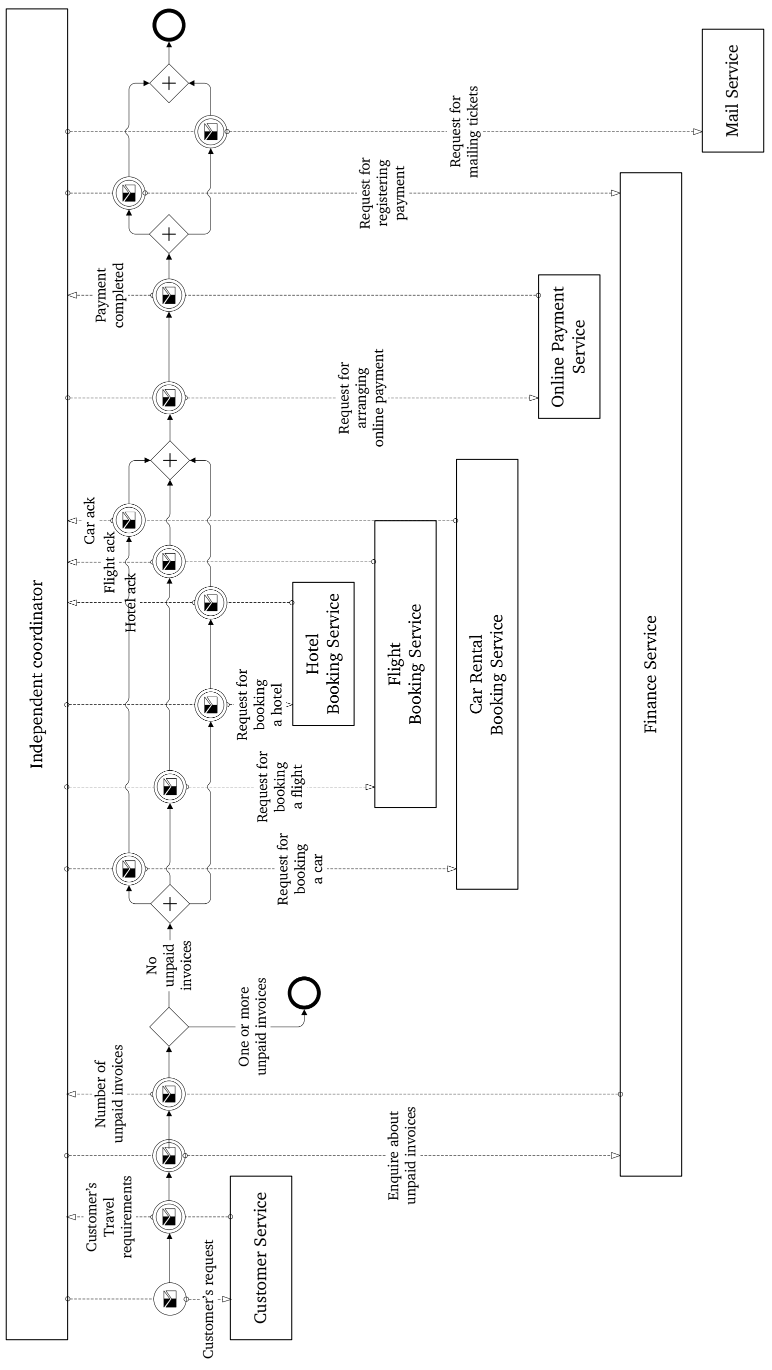

Independent coordinator
Customer Service
Customer's request
Customer's Travel requirements
Enquire about unpaid invoices
Number of unpaid invoices
One or more unpaid invoices
No unpaid invoices
Request for booking a car
Request for booking a flight
Request for booking a hotel
Hotel Booking Service
Flight Booking Service
Car Rental Booking Service
Hotel ack
Flight ack
Car ack
Request for arranging online payment
Online Payment Service
Payment completed
Request for registering payment
Request for mailing tickets
Finance Service
Mail Service

Figure 3.2: Independent coordinator managing all sequence dependencies

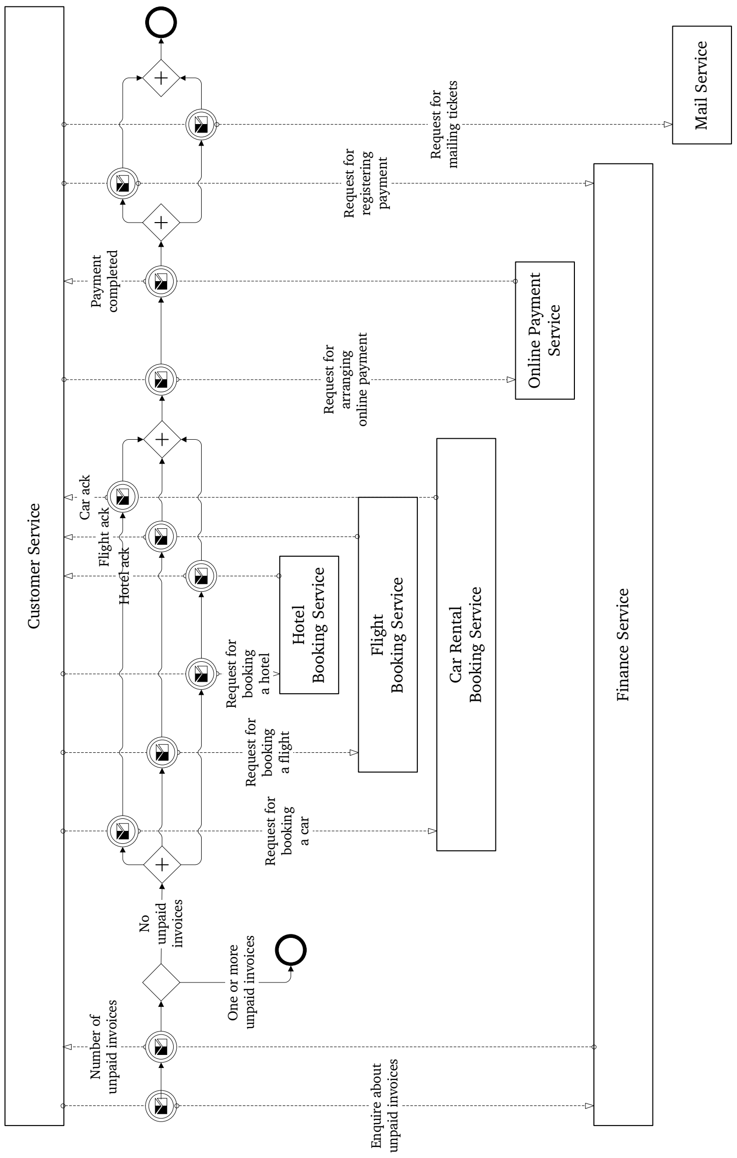

Customer Service
Enquire about unpaid invoices
Number of unpaid invoices
One or more unpaid invoices
No unpaid invoices
Request for booking a car
Request for booking a flight
Request for booking a hotel
Hotel Booking Service
Flight Booking Service
Car Rental Booking Service
Hotel ack
Flight ack
Car ack
Request for arranging online payment
Online Payment Service
Payment completed
Request for registering payment
Request for mailing tickets
Finance Service
Mail Service

Figure 3.3: Customer Service managing all sequence dependencies

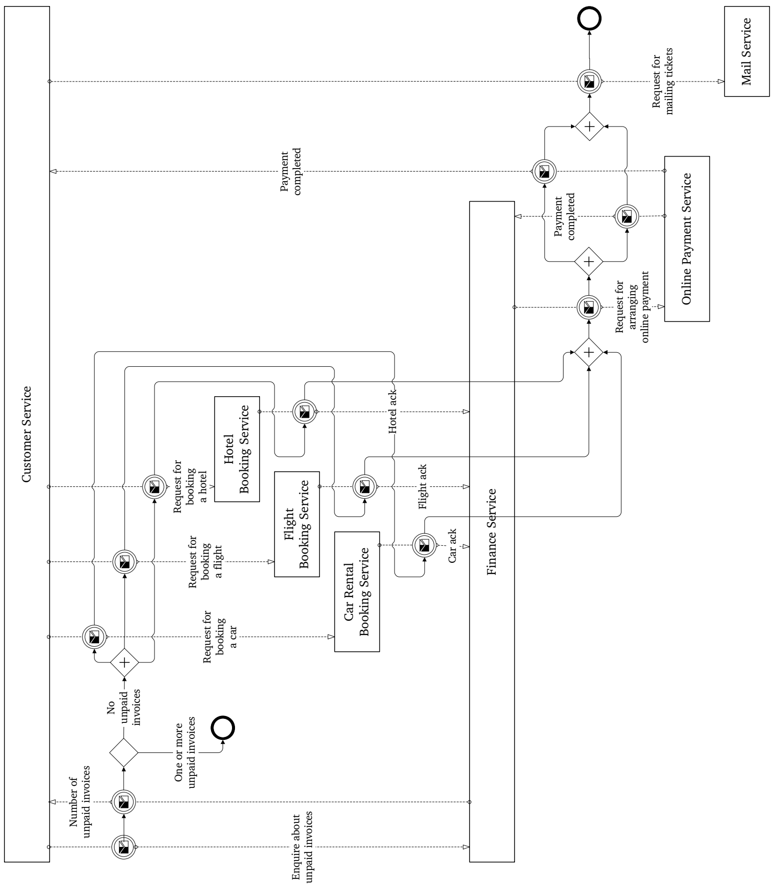

Customer Service
Enquire about unpaid invoices
Number of unpaid invoices
No unpaid invoices
One or more unpaid invoices
Request for booking a car
Request for booking a flight
Request for booking a hotel
Car Rental Booking Service
Flight Booking Service
Hotel Booking Service
Hotel ack
Flight ack
Car ack
Finance Service
Request for arranging online payment
Payment completed
Online Payment Service
Payment completed
Request for mailing tickets
Mail Service

Figure 3.4: Decentralized coordination (1)

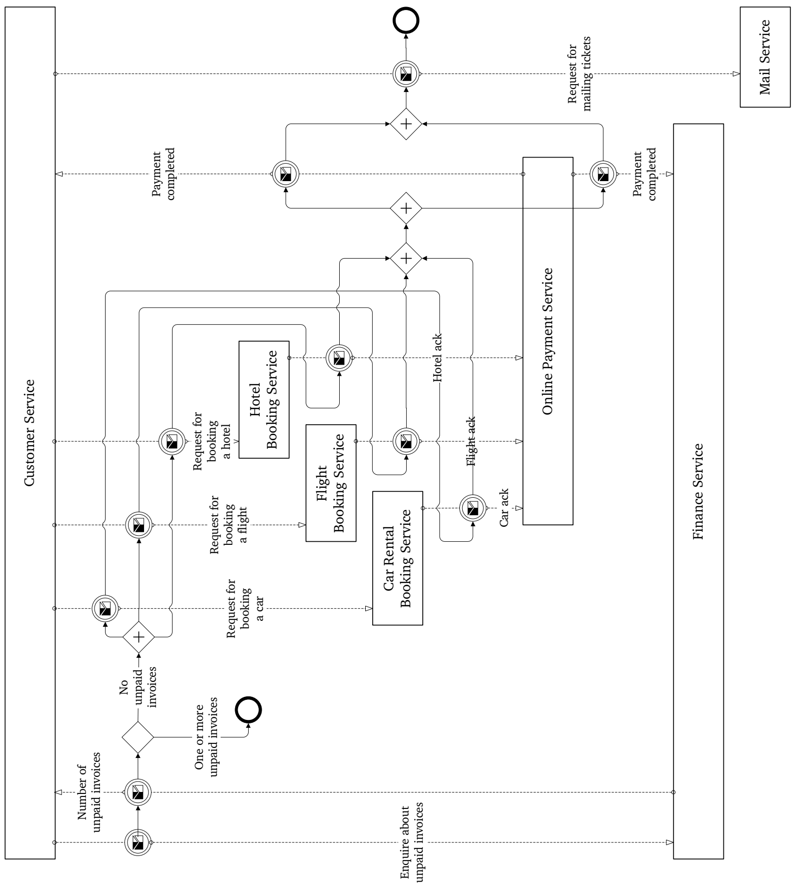

Customer Service
Finance Service
Mail Service
Online Payment Service
Hotel Booking Service
Flight Booking Service
Car Rental Booking Service
Enquire about unpaid invoices
Number of unpaid invoices
No unpaid invoices
One or more unpaid invoices
Request for booking a car
Request for booking a flight
Request for booking a hotel
Hotel ack
Flight ack
Car ack
Payment completed
Payment completed
Request for mailing tickets

Figure 3.5: Decentralized coordination (2)

In contrast, the coordination scenarios represented in Figure 3.4 and Figure 3.5 have no central coordinator. In Subsection 2.2.5 of Chapter 2 we refer to this kind of coordination scenarios as decentralized coordination. While in a centralized coordination scenario all sequence dependencies are managed centrally, in a decentralized coordination scenario the management of sequence dependencies is distributed among all services. For example, in the coordination scenario represented in Figure 3.4 the sequence between task 1 and 2 is managed by the Customer Service, while the Finance Service is responsible for requesting the Online Payment Service.

It is easy to understand that in case of centralized coordination, the example discussed above has eight possible coordination scenarios. This is so because, in theory, every service can take the role of central coordinator. Additionally, a separated service can act as the central coordinator. In case of a decentralized coordination scenario even many more coordination scenarios are possible. Furthermore, one can imagine that the solution space, containing all possible coordination scenarios, enlarges when the business process contains transactions (e.g. a transaction for booking the car, flight and hotel). This raises two main research questions concerning a coordination scenario:

RQ1 Can we come up with a *systematic way* of composing coordination scenarios, starting from some fundamental building blocks that can be combined to construct all possible scenarios that manage sequence dependencies?

RQ2 Can we provide service composers with a set of *design guidelines* for constructing a coordination scenario that complies to some predefined design criteria into account?

An approach covering these research questions allows service composers to construct a coordination scenario that manages all sequence dependencies and is optimized for aspects such as flexibility, loose coupling and performance.

3.2 Pattern language

3.2.1 Introduction

In this section we present five patterns that can be used to construct coordination scenarios that manage sequence dependencies. These patterns all deal with the same problem, in common context, and under influence of the same set of forces. The main difference between the patterns is the solution described in the pattern and the consequences that come with that solution.

Common context

In a service-based business process implementation different services are consumed in a coordinated way. Assuming that business process tasks are directly supported by the services, the business process describes the logic that can be used to find out when and which service should be consumed. In a service-based system this means that for each task in the business process there exists an entity that holds a set of *rules* to determine when it should send a *service request* to a certain *Service Provider*. Typically, these rules, which can be directly derived from the business process specification, specify which *business events* (e.g. the completion of a business process task) need to occur before a service request should be sent[1]. We assume that business events are generated and published by Service Providers and notifications of these events are disseminated by *business event dispatchers*. In practice, such a business event dispatcher can be either a Service Provider itself or part of a PUBLISH-SUBSCRIBE architecture (Avgeriou & Zdun, 2005; Eugster, Felber, Guerraoui, & Kermarrec, 2003).

Common problem

For each task in a business process one needs an answer to the following question:
Which entity sends a service request to the Service Provider supporting the business process task?

Common forces

When trying to find a solution to the problem described above, several forces are present. As we will show, each pattern presented in this chapter balances these forces differently. As such, these forces can be used as evaluation criteria for constructing an optimized coordination scenario.

In this thesis we do not aim to exhaustively identify all relevant forces when trying to construct a coordination scenario that manages sequence

[1]Concepts such as service requests (also referred to as business requests), Service Providers and business events were explained in our service composition meta-model proposed in Subsection 2.2.2 in Chapter 2.

dependencies in a service composition. Rather, we refer to common and frequently used forces in the literature (Erl, 2007; Goethals, 2008; Chang, Mazzoleni, Mihaila, & Cohn, 2008; Haesen, De Rore, Snoeck, Lemahieu, & Poelmans, 2006; Zirpins et al., 2004; Monsieur, De Rore, Snoeck, & Lemahieu, 2008):

- Performance: Intuitively, software engineers tend to create additional logical layers when business (process) logic must be implemented. Typically, such a logical layer contains process or task logic. However, if high levels of performance are required, this approach is less appropriate. In a service-based system additional logical layers increase the inter-service communication, resulting in a less performing system.
- Process flexibility: From time to time business processes change, and so service-based systems need to change. Preferably, business process logic can be easily changed without having to change services dramatically.
- Loose coupling and autonomy: A frequently discussed aspect of service-orientation is loose coupling. This aspect means that it should be relatively easy to replace or change services in a service-based system without having to make changes to other services or the global system. This would allow businesses to rapidly adapt their systems when necessary. In that perspective, some researchers propose to combine a service-oriented architecture with the strengths of an event-driven architecture. They consider services to be more autonomous, and implicitly more loosely coupled, when these are positioned in an SOA as entities that both react to events instead of requests and send out event information to other services instead of requests.
- Complexity: The entity that sends a request to the Service Provider requires business event information (e.g. the completion of another business process task) that is available elsewhere (i.e. business events published by other Service Providers). This potentially makes the global coordination scenario more complex. This is because business event information needs to be transferred between the entity holding the event information and the entity that needs to consume the event information.
- Business process monitoring: A business process forming the basis for a service-based system can be long-running. Therefore, it is potentially required that the business process progress can be easily monitored.
- Access restrictions: In a service-oriented system it can occur that a particular service only accepts requests from a limited set of clients. For

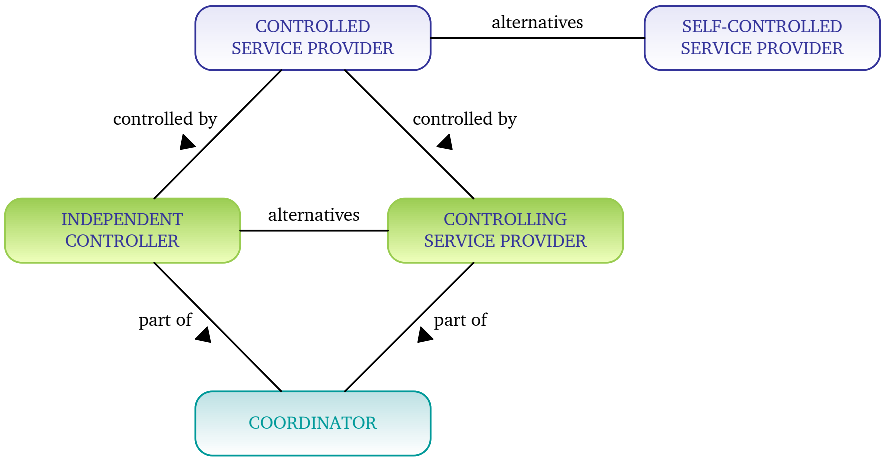


Figure 3.6: An overview of the pattern language

example, in a business-to-business environment between two companies often only a limited set of services in one company are allowed to interact with services in the other company.

3.2.2 Pattern overview

In general, we can distinguish between two possible solutions (see Figure 3.6). Either one applies the CONTROLLED SERVICE[2] pattern or one goes for the SELF-CONTROLLED SERVICE. A CONTROLLED SERVICE can be controlled by either an INDEPENDENT CONTROLLER or a CONTROLLING SERVICE PROVIDER. Additionally, this INDEPENDENT CONTROLLER or CONTROLLING SERVICE PROVIDER can be part of a COORDINATOR.

Each pattern presents a solution by describing which entity sends a request to the Service Provider. We will complement these descriptions with BPMN (OMG, 2010a) models that exactly show the internal actions and interactions that contribute to the management of sequence dependencies.

Additionally, we will summarize the solution using a simplified visual representation that consists of rectangles, circles and arrows (see Figure 3.7). A rectangle C_i denotes the *controller* that both sends a request to Service S_i and holds the set of rules to decide when it should send out this request. Service providers are represented using circles. A rounded rectangle is used

[2]Pattern names are presented in SMALL CAPITALS font

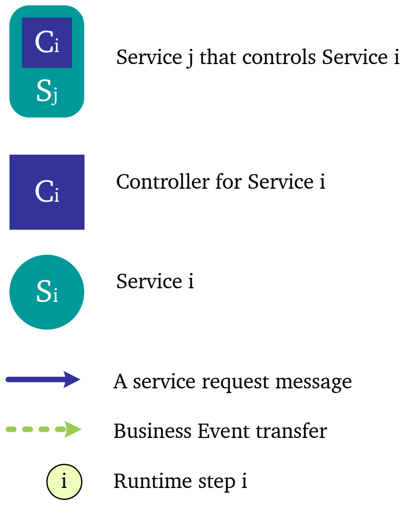


Figure 3.7: A legend for the simplified pattern representation

to represent a Service Provider that also controls the execution of a Service Provider (i.e. a Service Provider that is also a controller). Solid arrows denote requests sent from a controller C_i to a service S_i, while dashed arrows represent the transfer of business event information to a controller C_i.

Finally, we give for each pattern an example of how that pattern is applied in one of the coordination scenarios shown in Figures 3.2, 3.3, 3.4 or 3.5.

3.2.3 Controlled Service Provider

Solution

Use a CONTROLLED SERVICE PROVIDER that receives requests from a so called *controller*. This controller *controls* the execution of the CONTROLLED SERVICE PROVIDER. This means that the controller reacts to business events by sending a request to the CONTROLLED SERVICE PROVIDER so that the business process task supported by the service is executed at the right time, as specified in the business process.

Figure 3.8 shows a BPMN representation of a CONTROLLED SERVICE PROVIDER. A CONTROLLED SERVICE PROVIDER starts the task execution upon request. It receives requests from the controller. The controller receives business event notifications from a business event dispatcher. If all conditions

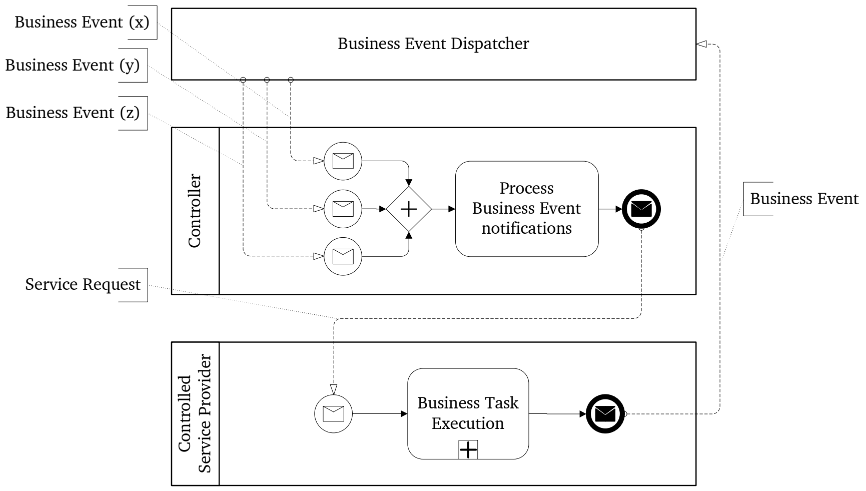


Figure 3.8: A BPMN representation of a CONTROLLED SERVICE PROVIDER

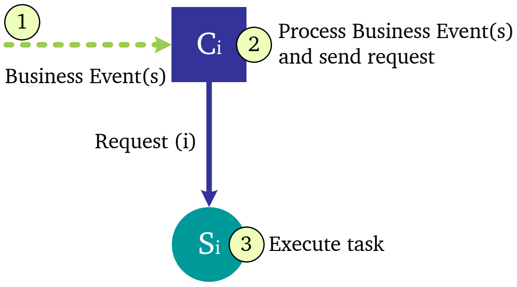


Figure 3.9: A simplified representation of a CONTROLLED SERVICE

as specified in the business process are met (e.g. business events x,y and z have occurred), a request is sent to the CONTROLLED SERVICE.

In Figure 3.9 a simplified representation of a CONTROLLED SERVICE PROVIDER is shown.

Consequences

A CONTROLLED SERVICE does not control its own execution. This means there exists a separate controller that sends service requests to the CONTROLLED SERVICE. Hence, a CONTROLLED SERVICE introduces an additional logical layer (i.e. the controller). Since requests need to be sent to the CONTROLLED SERVICE, this increases the inter-service communication, resulting in a less performing system.

A CONTROLLED SERVICE does not hold business process logic. As a consequence, a change in the business process does not require the modification of the CONTROLLED SERVICE, which increases the process flexibility of the overall system.

It is relatively difficult to replace a CONTROLLED SERVICE, since a CONTROLLED SERVICE implies that requests need to be sent to the CONTROLLED SERVICE. Hence, when the CONTROLLED SERVICE is replaced, the service requester (i.e. the controller) must be adapted so that it can send new requests to the alternative service. Thus a CONTROLLED SERVICE increases the coupling between services. Furthermore, a CONTROLLED SERVICE is less autonomous because it does not react to events, but only accepts requests.

If one wants to check if a certain task execution is already started, an interaction with the CONTROLLED SERVICE PROVIDER is not necessary. One needs to interact with the controller, which is potentially relatively easy. However, in case of multiple CONTROLLED SERVICE PROVIDERS with *different* controllers, multiple interactions with controllers are required in order to retrieve the process progress status, resulting in complex monitoring systems. The COORDINATOR pattern can further facilitate the monitoring (see Subsection 3.2.6).

Example

In the coordination scenarios discussed in the introductory example of this chapter (see Section 3.1), several applications of the CONTROLLED SERVICE PROVIDER pattern can be found. For example, in Figure 3.2 the CONTROLLED Finance Service receives a request 'register payment' from the INDEPENDENT COORDINATOR, which plays the role of controller and service requester. The INDEPENDENT COORDINATOR receives business event information 'payment completed' (see Figures 3.10 and 3.11) and processes this information using its local knowledge of the business process, before it sends out a request for registering the payment to the Finance Service (see Figures 3.10 and 3.11).

Related patterns

Per definition a CONTROLLED SERVICE PROVIDER is controlled by a controller. As mentioned earlier (see Subsection 3.2.2 and Figure 3.6) this controller

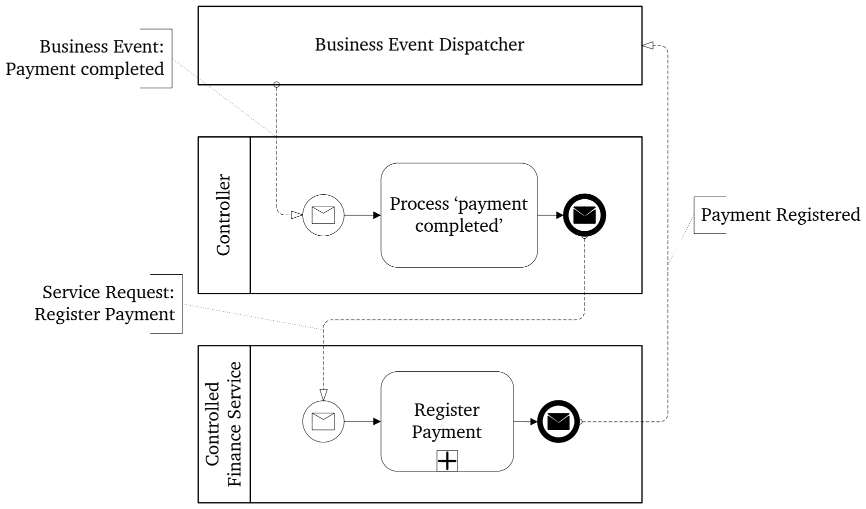

Business Event Dispatcher
Business Event: Payment completed
Controller
Process 'payment completed'
Payment Registered
Service Request: Register Payment
Controlled Finance Service
Register Payment

Figure 3.10: BPMN representation of the Finance Service as a CONTROLLED SERVICE PROVIDER

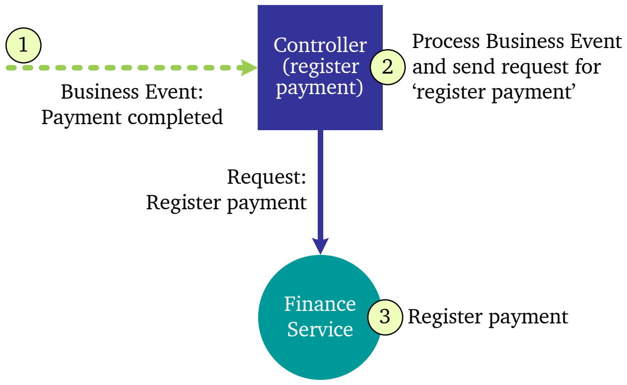

1
Business Event: Payment completed
Controller (register payment)
2
Process Business Event and send request for 'register payment'
Request: Register payment
Finance Service
3
Register payment

Figure 3.11: A simplified representation of the Finance Service as a CONTROLLED SERVICE PROVIDER

can be either an INDEPENDENT CONTROLLER (presented in Subsection 3.2.4) or a CONTROLLING SERVICE PROVIDER (presented in Subsection 3.2.5). Furthermore, this controller can be part of a COORDINATOR (see Subsection 3.2.6).

The COMMAND MESSAGE proposed by Hohpe and Woolf (2003) is a related pattern, because a CONTROLLED SERVICE PROVIDER, typically, receives a COMMAND MESSAGE that is a request for starting the execution of a certain business (process) task.

In a similar way a CONTROLLED SERVICE PROVIDER can be linked to the EXPLICIT INVOCATION PATTERN proposed by Avgeriou and Zdun (2005) since a CONTROLLED SERVICE PROVIDER needs to be invoked explicitly in order to start the execution of a certain business task.

3.2.4 Independent Controller

Pattern-specific context

One has selected a CONTROLLED SERVICE PROVIDER without choosing a specific controller.

Solution

Use a CONTROLLED SERVICE PROVIDER that receives requests from an INDEPENDENT CONTROLLER. An INDEPENDENT CONTROLLER is not a Service Provider in the service composition, but only controls the execution of a CONTROLLED SERVICE PROVIDER.

Figure 3.12 shows a BPMN representation of a CONTROLLED SERVICE PROVIDER controlled by an INDEPENDENT CONTROLLER. This solution is similar to the solution described in the CONTROLLED SERVICE PROVIDER pattern. However, while the solution described in the CONTROLLED SERVICE PROVIDER pattern refers to a *generic* kind of controller, this solution specifies that the CONTROLLED SERVICE PROVIDER is specifically controlled by an INDEPENDENT CONTROLLER.

In Figure 3.9 a simplified representation of a CONTROLLED SERVICE PROVIDER is shown.

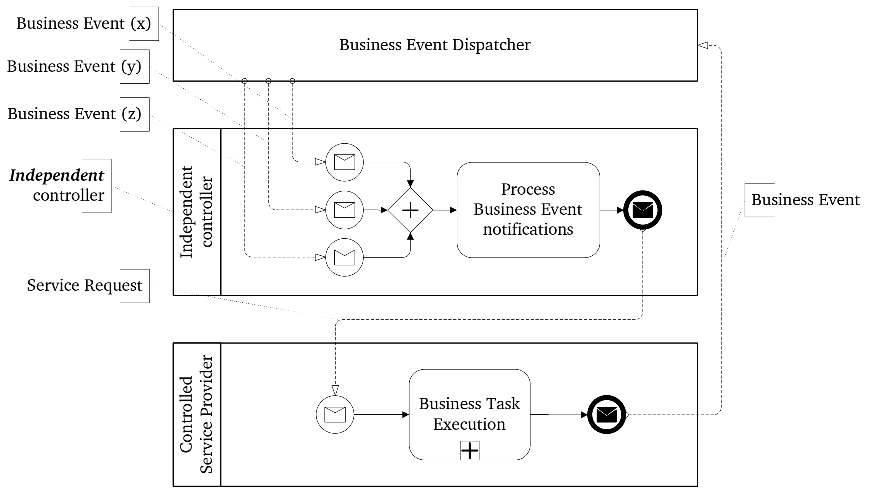


Figure 3.12: A BPMN representation of a CONTROLLED SERVICE PROVIDER controlled by an INDEPENDENT CONTROLLER

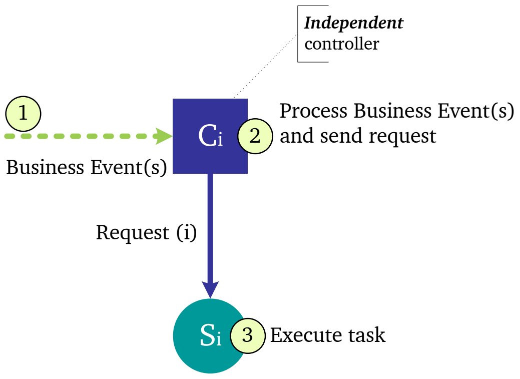


Figure 3.13: A simplified representation of a CONTROLLED SERVICE controlled by an INDEPENDENT CONTROLLER

Consequences

As described in the CONTROLLED SERVICE PROVIDER pattern the CONTROLLED SERVICE PROVIDER does not hold business process logic. Therefore, a change in the business process does not require the modification of the CONTROLLED SERVICE PROVIDER, which increases the process flexibility of the service-based system. Furthermore, when the CONTROLLED SERVICE PROVIDER is controlled by an INDEPENDENT CONTROLLER, the business process logic (or at least part of it) is clearly separated and not included in any Service Provider. Hence, a CONTROLLED SERVICE PROVIDER controlled by an INDEPENDENT CONTROLLER contributes to a higher level of process flexibility.

As explained in the common forces (see Subsection 3.2.1) a controller requires business event information in order to decide whether or not a request needs to be sent to the CONTROLLED SERVICE PROVIDER. This business event information needs to be sent to the INDEPENDENT CONTROLLER, which increases the complexity of the complete coordination scenario. However, if the INDEPENDENT CONTROLLER needs business event information that is also required by another INDEPENDENT CONTROLLER, then a COORDINATOR (see Subsection 3.2.6) can reduce the complexity.

The CONTROLLED SERVICE PROVIDER needs to accept requests from an INDEPENDENT CONTROLLER. If the CONTROLLED SERVICE PROVIDER only accepts requests from certain Service Providers, a CONTROLLING SERVICE PROVIDER as controller is more appropriate (see Subsection 3.2.5).

Example

In the coordination scenario specified in Figure 3.2 all Service Providers are controlled by an INDEPENDENT CONTROLLER, namely the independent coordinator. For example, the CONTROLLED Finance Service (see Figures 3.10 and 3.11) is controlled by an INDEPENDENT CONTROLLER.

3.2.5 Controlling Service Provider

Pattern-specific context

One has selected a CONTROLLED SERVICE PROVIDER without choosing a specific controller.

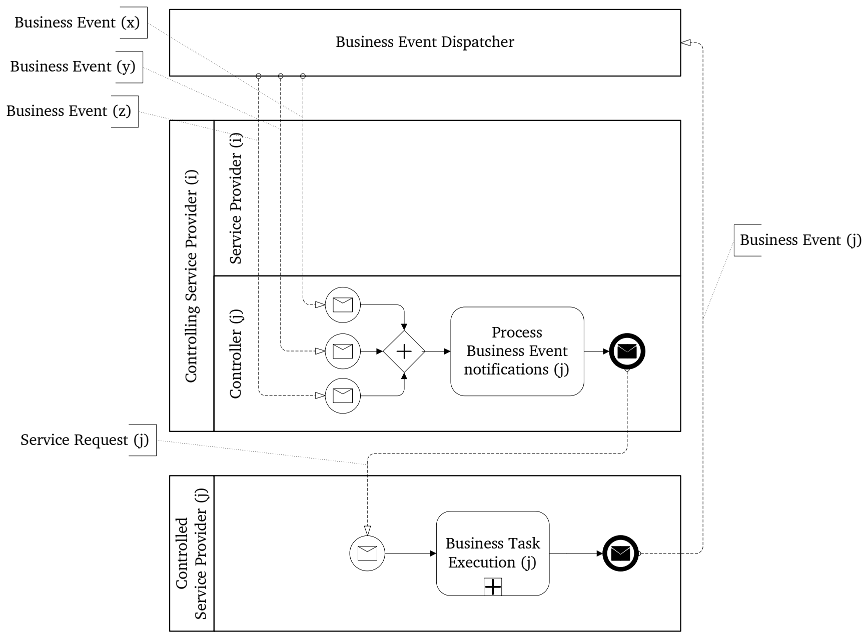

Business Event (x)
Business Event (y)
Business Event (z)
Business Event Dispatcher
Controlling Service Provider (i)
Service Provider (i)
Controller (j)
Process Business Event notifications (j)
Business Event (j)
Service Request (j)
Controlled Service Provider (j)
Business Task Execution (j)

Figure 3.14: A BPMN representation of a CONTROLLED SERVICE PROVIDER controlled by a CONTROLLING SERVICE PROVIDER

Solution

Use a CONTROLLED SERVICE PROVIDER controlled by a CONTROLLING SERVICE PROVIDER. The latter Service Provider is a Service Provider that plays the role of controller for the CONTROLLED SERVICE PROVIDER.

Figure 3.14 shows a BPMN representation of a CONTROLLED SERVICE PROVIDER controlled by a CONTROLLING SERVICE PROVIDER. A CONTROLLED SERVICE PROVIDER controlled by a CONTROLLING SERVICE PROVIDER receives requests from another Service Provider. The latter Service Provider reacts to business event information and sends the necessary requests to the CONTROLLED SERVICE PROVIDER.

In the simplified representation of a CONTROLLED SERVICE PROVIDER controlled by a CONTROLLING SERVICE PROVIDER (see Figure 3.15 the controller C_j is included in the Service Provider S_i to indicate that Service i controls Service j.

Note that this pattern does not tell anything about how the CONTROL-

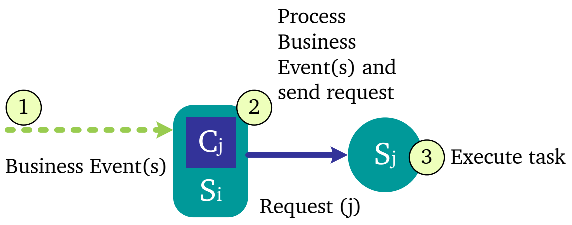


Figure 3.15: A simplified representation of a CONTROLLED SERVICE controlled by a CONTROLLING SERVICE PROVIDER

LING SERVICE PROVIDER (Service S_i) is controlled. The CONTROLLING SERVICE PROVIDER can be either a CONTROLLED SERVICE PROVIDER or a SELF-CONTROLLED SERVICE PROVIDER. This is the reason why we left the Service Provider lane in Figure 3.14 empty.

Consequences

A CONTROLLED SERVICE PROVIDER (controlled by a CONTROLLING SERVICE PROVIDER) does not hold business process logic. Therefore, a change in the business process does not require the modification of the CONTROLLED SERVICE PROVIDER. However, if a CONTROLLING SERVICE PROVIDER is used as controller, a change in the business process requires the modification of the CONTROLLING SERVICE PROVIDER (Service i in Figures 3.14 and 3.15).

If the event information, that is needed to decide whether a request should be sent to the CONTROLLED SERVICE PROVIDER, is available at the CONTROLLING SERVICE PROVIDER, then a CONTROLLED SERVICE PROVIDER controlled by a CONTROLLING SERVICE PROVIDER results into a less complex coordination scenario. This is thanks to the fact that no event information needs to be transferred, since both event source and event consumer (Controller C_j in Figures 3.14 and 3.15) belong to the same service (i.e. the CONTROLLING SERVICE PROVIDER or Service S_i in Figures 3.14 and 3.15).

The CONTROLLING SERVICE PROVIDER (Service S_i in Figures 3.14 and 3.15) needs to be an accepted client for the CONTROLLED SERVICE PROVIDER. If the CONTROLLED SERVICE PROVIDER only accepts requests from an independent party, an INDEPENDENT CONTROLLER is more appropriate (see Subsection 3.2.4).

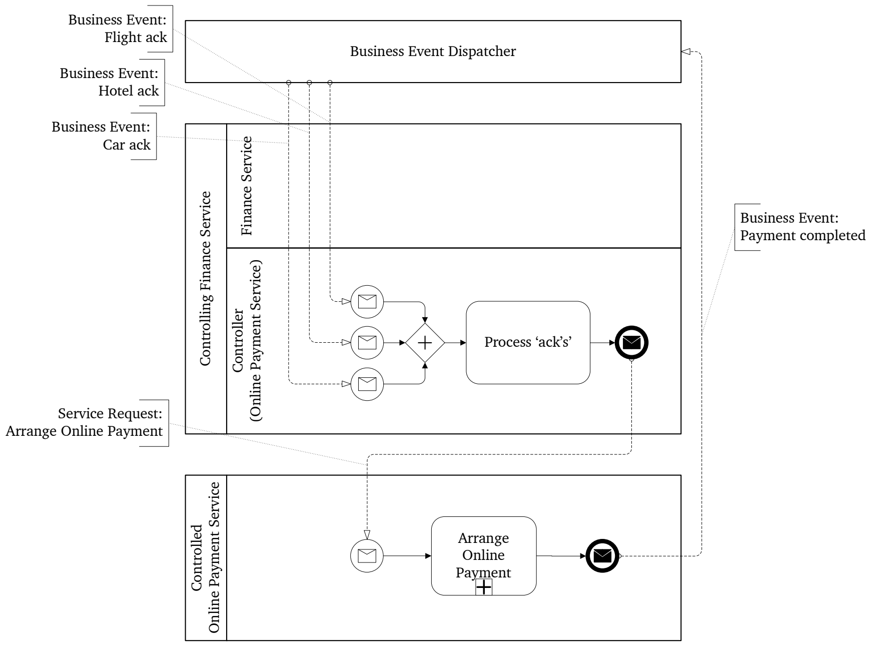

Business Event: Flight ack
Business Event: Hotel ack
Business Event: Car ack
Business Event Dispatcher
Controlling Finance Service
Finance Service
Controller (Online Payment Service)
Process 'ack's'
Business Event: Payment completed
Service Request: Arrange Online Payment
Controlled Online Payment Service
Arrange Online Payment

Figure 3.16: A BPMN representation of the Finance Service controlling the Online Payment Service

Example

In the coordination scenario represented in Figure 3.4 the Online Payment Service is a CONTROLLED SERVICE PROVIDER controlled by a CONTROLLING SERVICE PROVIDER, because it is the Finance Service that sends an 'arrange online payment' request to the Online Payment Service (see Figures 3.16 and 3.17). This coordination scenario is possibly preferable when the Online Payment Service only accepts requests (to arrange online payments) from the Finance Service.

Related patterns

A CONTROLLED SERVICE PROVIDER controlled by a CONTROLLING SERVICE PROVIDER can be related to *decentralized coordination*, as described in Subsection 2.2.5 of Chapter 2. This is so because in a *decentralized coordination* in which Service Providers collaborate together, coordination responsibilities are distributed among Service Providers.

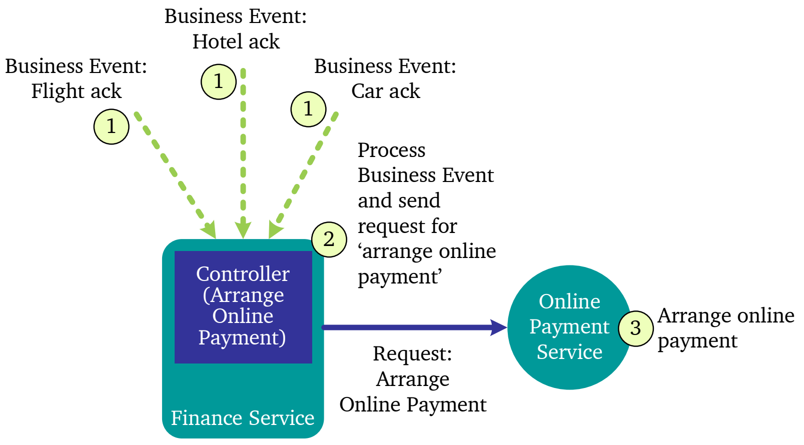

Business Event: Hotel ack
Business Event: Flight ack
Business Event: Car ack
1
1
1
Process Business Event and send request for 'arrange online payment'
2
Controller (Arrange Online Payment)
Finance Service
Request: Arrange Online Payment
Online Payment Service
3
Arrange online payment

Figure 3.17: A simplified representation of a the Finance Service controlling the Online Payment Service

3.2.6 Coordinator

Pattern-specific context

One has selected a CONTROLLED SERVICE PROVIDER (either controlled by an INDEPENDENT CONTROLLER or a CONTROLLING SERVICE PROVIDER).

Solution

Use a CONTROLLED SERVICE PROVIDER controlled by a COORDINATOR, which is a service that controls multiple Service Providers.

Figure 3.18 shows a BPMN representation of a CONTROLLED SERVICE PROVIDER controlled by a COORDINATOR. The coordinator also has the business process logic that specifies when another business task needs to be executed (i.e. the rules to determine if a request needs to be sent to another Service Provider).

In the simplified representation of the CONTROLLED SERVICE PROVIDER controlled by a COORDINATOR in Figure 3.19 this is represented by merging the controller C_j with another controller C_i. Note that this pattern does not tell anything about the location of Service S_i, which means that Service S_i can be either a SELF-CONTROLLED SERVICE PROVIDER or a CONTROLLED SERVICE PROVIDER. This the reason why in Figure 3.18 the COORDINATOR includes

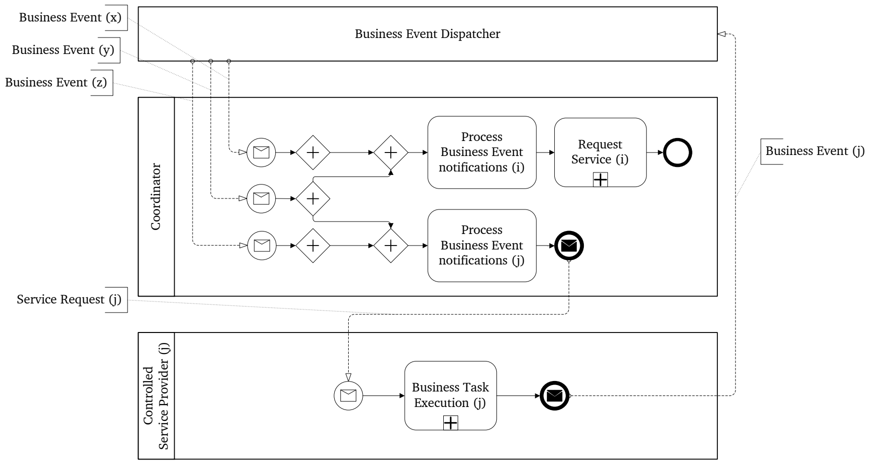

Business Event (x)
Business Event (y)
Business Event (z)
Business Event Dispatcher
Coordinator
Process Business Event notifications (i)
Request Service (i)
Process Business Event notifications (j)
Business Event (j)
Service Request (j)
Controlled Service Provider (j)
Business Task Execution (j)

Figure 3.18: A BPMN representation of a CONTROLLED SERVICE PROVIDER controlled by a COORDINATOR

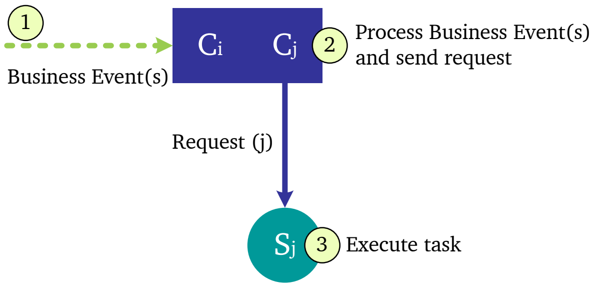


Figure 3.19: A simplified representation of a CONTROLLED SERVICE controlled by a COORDINATOR

a collapsed subprocess - named ’Request Service (i)’ - for the consumption of Service S_i. In case that S_i is a SELF-CONTROLLED SERVICE PROVIDER we say that S_i is controlled by a COORDINATING SERVICE PROVIDER. If S_i is a CONTROLLED SERVICE PROVIDER we refer to the coordinator as an INDEPENDENT COORDINATOR.

Consequences

The COORDINATOR holds business process logic related to multiple business process tasks. For each business process task business event information is needed to decide when the business process task should be executed. In case there is a common need for certain event information, the COORDINATOR pattern decreases the complexity of the overall coordination scenario. This is true because that kind of event information needs to be transferred only once from the source to the consumer.

In the COORDINATOR pattern there is *one* entity containing logic for multiple business process tasks. This eases the process monitoring, because one would need only *one* interaction for retrieving information concerning *multiple* tasks. Furthermore, it potentially also contributes to a higher level of process flexibility, because the COORDINATOR pattern limits the number of components that need to be modified in case of a change in the business process.

Example

In each coordination scenario discussed in the introduction (see Figures 3.2, 3.3, 3.4 and 3.5) the entity that sends the ’book car’ request to the Car Rental Booking Service, also has the responsibility to send requests to the Hotel Booking Service and Flight Booking Service. Since for each responsibility there is a common need for specific business event information (no unpaid invoices), the COORDINATOR pattern results into a preferable coordination scenario. In Figures 3.20 and 3.21 this pattern is applied to the Car Rental Booking Service.

Related patterns

This pattern is related to the ORCHESTRATION and PROCESS CENTRALIZATION patterns as described by Erl (2009), because the main idea behind a COORDINATOR is that two or more controllers are merged into one controlling entity referred to as a COORDINATOR.

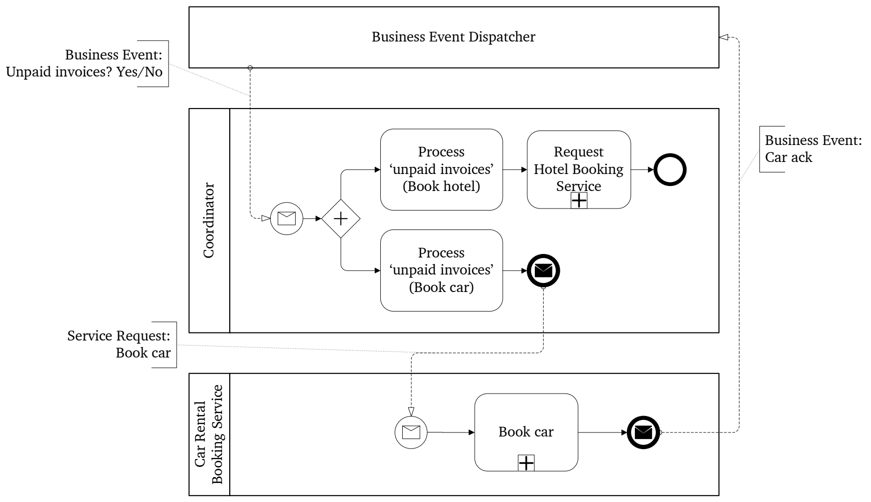


Figure 3.20: A BPMN representation of a CONTROLLED SERVICE PROVIDER controlled by a COORDINATOR

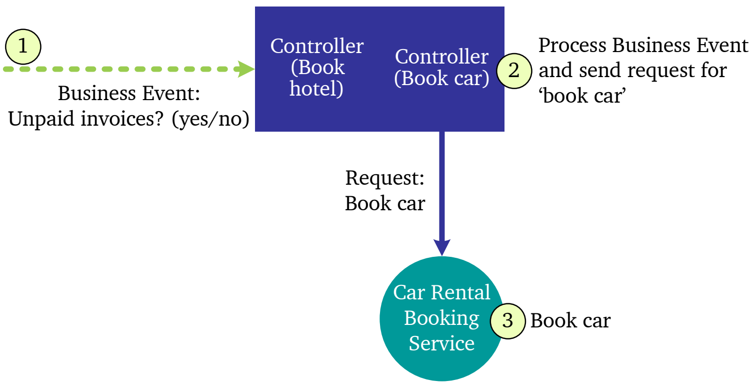


Figure 3.21: A simplified representation of a CONTROLLED SERVICE controlled by a COORDINATOR

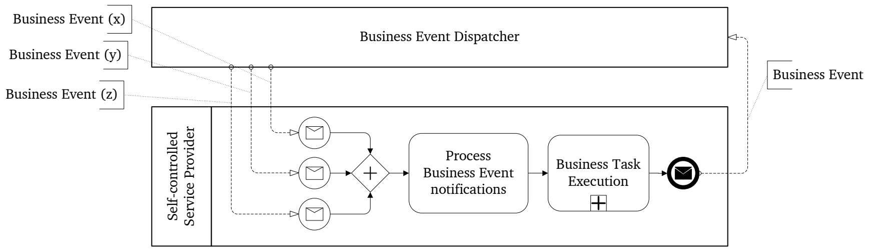

Business Event (x)
Business Event (y)
Business Event (z)
Business Event Dispatcher
Business Event
Self-controlled Service Provider
Process Business Event notifications
Business Task Execution

Figure 3.22: A BPMN representation of a SELF-CONTROLLED SERVICE PROVIDER

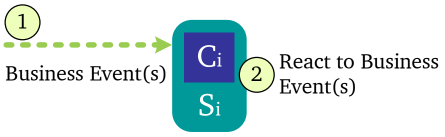


Figure 3.23: A simplified representation of a SELF-CONTROLLED SERVICE

In similar way, this pattern is related to *centralized coordination*, as described in Subsection 2.2.5 of Chapter 2, because coordination responsibilities are grouped together in one COORDINATOR.

3.2.7 Self-controlled Service Provider

Solution

Use a SELF-CONTROLLED SERVICE PROVIDER that reacts to business events so that business process tasks are executed at the right time, as specified in the business process.

Figure 3.22 shows a BPMN representation of a SELF-CONTROLLED SERVICE PROVIDER. A SELF-CONTROLLED SERVICE PROVIDER does not start the task execution upon request. Instead, it simply receives and processes certain business event information, provided by a business event dispatcher. If all conditions as specified in the business process are met (e.g. business events x,y and z have occurred), the SELF-CONTROLLED SERVICE PROVIDER reacts to the events by starting the execution of the business process task.

In Figure 3.23 a simplified representation of a SELF-CONTROLLED SERVICE PROVIDER is shown.

Consequences

Since a SELF-CONTROLLED SERVICE PROVIDER holds (partial) business process logic, the execution of business process task can start immediately when necessary. A SELF-CONTROLLED SERVICE PROVIDER directly reacts to event information. No additional request messages need to be sent to the SELF-CONTROLLED SERVICE PROVIDER, which has a relatively positive influence on the overall performance.

A SELF-CONTROLLED SERVICE PROVIDER (negatively) effects the process flexibility of the service-based system. A SELF-CONTROLLED SERVICE includes partial business process logic, in the form of a set of rules to determine if a business process task needs to executed, and this means that a change in the business process potentially requires the modification of the SELF-CONTROLLED SERVICE PROVIDER.

As discussed earlier a SELF-CONTROLLED SERVICE PROVIDER reacts to event information. Hence, a SELF-CONTROLLED SERVICE PROVIDER can be considered more autonomous. Furthermore, it is assumed that a SELF-CONTROLLED SERVICE PROVIDER can be relatively easier replaced in a service-based system, because no explicit requests are sent to the SELF-CONTROLLED SERVICE PROVIDER. The idea is that event information sent to the SELF-CONTROLLED SERVICE PROVIDER can be simply sent to an alternative SELF-CONTROLLED SERVICE PROVIDER. In that sense, the service-based system has a loose coupling with the SELF-CONTROLLED SERVICE PROVIDER.

Since a SELF-CONTROLLED SERVICE PROVIDER reacts to event information a complete coordination scenario requires the correct transfer of event information to the SELF-CONTROLLED SERVICE PROVIDER. If this event information comes from several other services (e.g. in case of multiple sequence dependencies between the SELF-CONTROLLED SERVICE PROVIDER and multiple other services providers, the overall coordination scenario is relatively complex.

If one wants to check if a certain task execution has already started, an interaction with the SELF-CONTROLLED SERVICE PROVIDER is necessary. Hence, the more SELF-CONTROLLED SERVICE PROVIDERS constitute a service-based system, the more interactions with services are needed when one wants to monitor the business process progress. However, if business event notifications are sent using a PUBLISH-SUBSCRIBE architecture (Avgeriou & Zdun, 2005; Eugster et al., 2003), the business process state can still be relatively easily derived by interacting with the entity that disseminates the business event notifications and observing which business events have occurred.

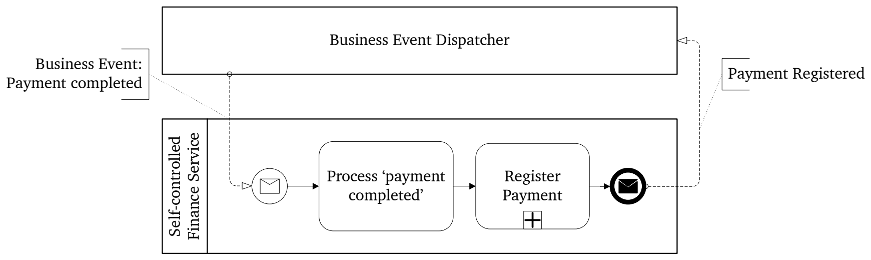


Figure 3.24: A BPMN representation of a SELF-CONTROLLED SERVICE PROVIDER

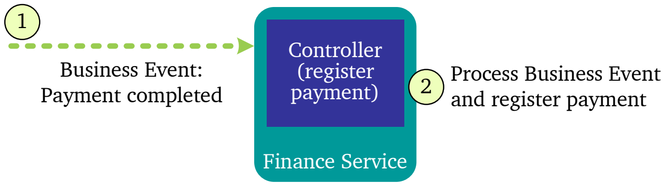


Figure 3.25: A simplified representation of a SELF-CONTROLLED SERVICE

Access restrictions are not relevant when dealing with a SELF-CONTROLLED SERVICE PROVIDER, because, per definition, a SELF-CONTROLLED SERVICE PROVIDER does not accept requests.

Example

The SELF-CONTROLLED SERVICE PROVIDER pattern is applied in the example represented in Figure 3.5. In this example the SELF-CONTROLLED Finance Service receives business event information concerning the payment. In particular, it receives a 'payment completed' message from the Online Payment Service. Subsequently, it is the responsibility of the Finance Service to react to this event information as specified in the business process. This means that the Finance Service contains process logic for the 'register payment' task and registers a payment when it receives event information concerning that payment (see Figures 3.24 and 3.25). In this scenario, this creates a loose coupling between the Online Payment Service and the Finance Service.

Related patterns

The EVENT MESSAGE as described by Hohpe and Woolf (2003) is a related pattern, because a SELF-CONTROLLED SERVICE PROVIDER, typically, reacts to EVENT MESSAGES by starting the execution of a certain business (process) task.

In a similar way a SELF-CONTROLLED SERVICE PROVIDER can be linked to the IMPLICIT INVOCATION PATTERN proposed by Avgeriou and Zdun (2005) since a SELF-CONTROLLED SERVICE PROVIDER is not invoked explicitly. In order to start the execution of a certain business task the SELF-CONTROLLED SERVICE PROVIDER needs to be implicitly invoked through mechanisms such as a PUBLISH-SUBSCRIBE system (Avgeriou & Zdun, 2005; Eugster et al., 2003).

This pattern is also related to what is often referred to as *implicit invocation* or *event-based* architectures (Shaw & Garlan, 1996), in which business event notifications replace explicit invocations.

3.3 Applying the patterns in practice

3.3.1 Design guidelines

As explained in the introduction to the pattern language (see Subsection 3.2.1), a coordination scenario specifies for each business process task which entity is responsible for sending a service request to the Service Provider supporting that task. The patterns presented in this chapter (see Section 3.2 describe different ways of distributing such a responsibility. This means that in order to build a complete coordination scenario one needs to apply the patterns for each business process task.

In order to design a coordination scenario that is optimized to a certain set of criteria, it is important to take all consequences that come with a pattern into account. However, one should understand that an optimized solution specific for a certain business process task, does not completely determine the optimization of the complete coordination scenario. For example, if one applies a pattern that contributes to an increased level of process flexibility (e.g. a CONTROLLED SERVICE PROVIDER controlled by an INDEPENDENT CONTROLLER), the global coordination scenario can still have a moderate or even low level of process flexibility (e.g. in case of multiple INDEPENDENT CONTROLLERS instead of a few or a single COORDINATOR(S)). In particular, this is the case for forces such as performance, process flexibility, complexity and process monitoring, which are dependent from the global solution.

Therefore, designing a coordination scenario by means of the patterns is preferably an iterative process in which forces are constantly balanced. In case of an under optimized coordination scenario, it is important to carefully look at the pattern consequences and find additional improvements.

Based on these thoughts and the pattern language overview (see Subsection 3.2.2) we can come up with the following design process:

1. Taking the identified pattern consequences into account, determine for each business process task and corresponding Service Provider whether a CONTROLLED SERVICE PROVIDER or a SELF-CONTROLLED SERVICE PROVIDER is the most appropriate solution.
2. For each CONTROLLED SERVICE PROVIDER determine whether it is better to control the Service Provider by an INDEPENDENT CONTROLLER or a CONTROLLING SERVICE PROVIDER.
3. Try to make additional improvements by combining controllers into COORDINATORS.

3.3.2 Coordination in workflow patterns

The workflow patterns proposed by Van der Aalst et al. (2003) are often used for the evaluation of business process modeling languages (e.g. BPMN (Wohed, Van der Aalst, Dumas, Ter Hofstede, & Russell, 2006) or UML (OMG, 2010b) Activity Diagrams (Wohed, Van der Aalst, Dumas, Ter Hofstede, & Russell, 2005)), because these patterns are considered as fundamental process constructs. Therefore, we try to prove the value of our patterns by showing that our patterns can be used to coordinate the sequence of activities specified in a workflow pattern. In particular, we will focus on the basic control flow patterns (SEQUENCE, PARALLEL SPLIT, SYNCHRONIZATION, EXCLUSIVE CHOICE, SIMPLE MERGE) and advanced branching and synchronization patterns (MULTI-CHOICE, SYNCHRONIZING MERGE, MULTI-MERGE, DISCRIMINATOR).

First we will discuss the coordination of the sequence pattern. The other basic workflow patterns can be classified as either a split pattern (PARALLEL SPLIT, EXCLUSIVE CHOICE, MULTI-CHOICE) or a join pattern (SYNCHRONIZATION, SIMPLE MERGE, SYNCHRONIZING MERGE, MULTI-MERGE, DISCRIMINATOR). Therefore, we will limit our discussion to the coordination of one split pattern (PARALLEL SPLIT) and one join pattern (SYNCHRONIZATION). Analogously, one can construct coordination scenarios for the remaining patterns.

Sequence pattern

The SEQUENCE workflow pattern consists of two tasks, Task 1 and Task 2, whereby Task 2 can only start when Task 1 is completed. If Services 1 and 2 support these tasks, the pattern requires that Service 2 must start the execution of Task 2 when Service 1 has completed the execution of Task 1. We assume that event information concerning the completion of Task 1 is available at Service 1.

For both Task 1 and Task 2 one needs to decide which entity is responsible for sending a request to the Service Provider supporting that task. If we follow the design guidelines described in Subsection 3.3.1, this means that we first need to choose between a CONTROLLED SERVICE PROVIDER and a SELF-CONTROLLED SERVICE PROVIDER for both Service 1 and Service 2. Then, in case of a CONTROLLED SERVICE PROVIDER we need to specify if the Service Provider is controlled by an INDEPENDENT CONTROLLER or a CONTROLLING SERVICE PROVIDER. Additionally, the controller can be part of COORDINATOR. Hence, for both Service 1 and Service 2 there exist five different solutions: a SELF-CONTROLLED SERVICE PROVIDER, a Service Provider controlled by an INDEPENDENT CONTROLLER, a Service Provider controlled by a CONTROLLING SERVICE PROVIDER, a Service Provider controlled by an INDEPENDENT COORDINATOR or a Service Provider controlled by a COORDINATING SERVICE PROVIDER. Theoretically, this would mean that there are twenty-five ways of coordinating the sequence pattern. However, as shown in Table 3.1 some combinations are not possible, while other combinations result into equal coordination scenarios.

As shown in Table 3.1 eleven combinations are not possible. This is easy to understand when considering a CONTROLLED SERVICE PROVIDER controlled by a COORDINATING SERVICE PROVIDER. Suppose that Service 2 is controlled by a COORDINATING Service 1. This means that Service 1 sends requests to both Service 1 and another service. The latter service is Service 1 itself, because there are only two services involved in the coordination of the SEQUENCE pattern. This implies that Service 1 needs to be a SELF-CONTROLLED SERVICE PROVIDER, which directly drops eight combinations (see last column and row in Table 3.1).

In a similar way, it is possible to explain why three other combinations are not possible. In particular, if one service is a CONTROLLED SERVICE PROVIDER controlled by an INDEPENDENT COORDINATOR, then the other service needs per definition to be also controlled by that same INDEPENDENT COORDINATOR. Hence, a Service Provider controlled by an INDEPENDENT COORDINATOR can only be combined with a Service Provider controlled by the same INDEPENDENT COORDINATOR.

	Service 2 / Service 1	(1)	(2)	(3)	(4)	(5)
(1)	SELF-CONTROLLED SP			implies that Service 2 is controlled by COORDINATING SP	Not possible	▨
(2)	SP controlled by an INDEPENDENT CONTROLLER				implies that both services are controlled by one INDEPENDENT COORDINATOR	Not possible
(3)	SP controlled by a CONTROLLING SP	implies that Service 1 is controlled by COORDINATING SP			Not possible	Not possible
(4)	SP controlled by an INDEPENDENT COORDINATOR	Not possible	implies that both services are controlled by one INDEPENDENT COORDINATOR	Not possible	implies that both services are controlled by one INDEPENDENT COORDINATOR	Not possible
(5)	SP controlled by a COORDINATING SP	▨	Not possible	Not possible	Not possible	Not possible

Table 3.1: Combining the patterns to coordinate a SEQUENCE. Shaded cells with the same grayscale level result into the same coordination scenario.

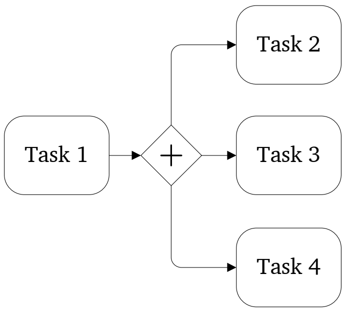


Figure 3.26: PARALLEL SPLIT pattern represented in BPMN

A concrete coordination scenario can be evaluated by studying the consequences of the patterns that are combined into the coordination scenario. For example, if one would like to limit the overall complexity of the coordination scenario, it is advisable to design Service 2 as a CONTROLLED SERVICE PROVIDER controlled by a CONTROLLING Service 1. Since Service 1 has event information that is needed by the controller of Service 2, the overall complexity of the coordination scenario is decreased if Service 1 is the controller of Service 2.

In the introductory example of this chapter (see Section 3.1) there is one SEQUENCE workflow pattern (see Figure 3.1): 'Checking for unpaid invoices' (Task 2) must be executed after the customer's request is processed (Task 1). In the coordination scenario represented in Figure 3.3 this sequence is coordinated using a CONTROLLED Finance Service that is controlled by a CONTROLLING Customer Service.

Split pattern (parallel split)

In the PARALLEL SPLIT workflow pattern the execution of several tasks needs to be started in parallel after the completion of a certain task. In what follows we refer to the latter task as the *before-split task*, while the other tasks, which are executed in parallel, are referred to as the *after-split tasks*. Analogously, we refer to the service supporting the before-split task as the *before-split service* and services supporting after-split tasks are referred to as *after-split services*. In figure 3.26 the parallel split pattern is visualized in BPMN (OMG, 2010a). Task 1 is the before-split task, while Task 2, 3 and 4 are the after-split tasks.

Similar to the discussion on the SEQUENCE workflow pattern one could

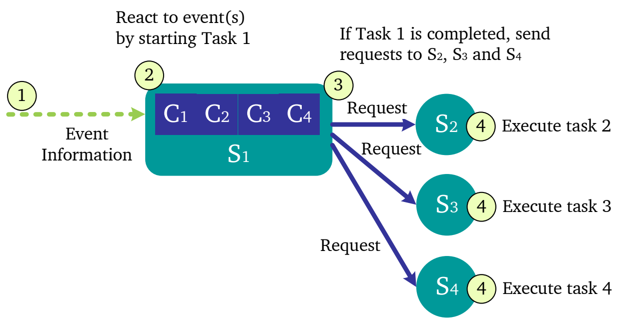


Figure 3.27: Coordinated parallel split using a SELF-CONTROLLED (BEFORE-SPLIT) SERVICE and CONTROLLED (AFTER-SPLIT) SERVICES controlled by a COORDINATING (BEFORE-SPLIT) SERVICE

study all possible combinations of our patterns to construct an appropriate coordination scenario for the PARALLEL SPLIT workflow pattern. However, since a parallel split consists of at least three tasks, one can, theoretically, consider 125 coordination scenarios. Therefore, we prefer to follow the design guidelines presented in Subsection 3.3.1. This means that first a choice should be made between a SELF-CONTROLLED SERVICE PROVIDER and a CONTROLLED SERVICE PROVIDER. Subsequently, one can consider the INDEPENDENT CONTROLLER and CONTROLLING SERVICE PROVIDER patterns as potential controllers for a CONTROLLED SERVICE PROVIDER. Additionally, a COORDINATOR can further balance the forces and optimize a concrete coordination scenario. For example, as discussed in the pattern consequences (see Subsection 3.2.6) a COORDINATOR can be valuable in case of common event information needs. Since this is clearly the case in the split pattern (*all* tasks after the split need to be executed when the task before the split is completed), it is advisable to construct a coordination scenario in which after-split services are controlled by a COORDINATOR. As such, the event information available at the before-split service only needs to be transferred once to the entity that is orchestrating the after-split services. Additionally, one can motivate the use of a CONTROLLING SERVICE PROVIDER as controllers for the after-split services. In particular, this would mean that the before-split service takes the role of controller for all after-split services (see Figure 3.27).

As explained in Subsection 3.2.7, SELF-CONTROLLED SERVICE PROVIDERS have a higher degree of autonomy and the use of such services tends to lead to a loose coupling between services. Supposing that after-split services are SELF-CONTROLLED SERVICE PROVIDERS, the loose coupling results into a sort of improved flexibility because an after-split service can be easily added or removed from the service-based system without having to change the system drastically. This is in contrast to the solution described above, which is based on CONTROLLED SERVICE PROVIDERS and which requires a change in the controlling (and coordinating) entity for each to be added or removed after-split service.

Join pattern (synchronization)

In the SYNCHRONIZATION workflow pattern the execution of a task needs to be started when all tasks in a set of other tasks are completed. In what follows we refer to the first task as the after-join task, while the other tasks, which needs to be completed before the after-join task can start, are referred to as the before-join tasks. Analogously, we refer to the services supporting the before-join tasks as the before-join services and the service supporting the after-join task is referred to as the after-join service. In figure 3.28 the SYNCHRONIZATION pattern is visualized in BPMN (OMG, 2010a). Tasks 1, 2 and 3 are the before-join tasks, while Task 4 is the after-join task.

Similar to the coordination of the PARALLEL SPLIT workflow pattern one can compose a large set of coordination scenarios for the JOIN workflow pattern by combining our patterns in different ways. For each business process task one first needs to make a choice between a SELF-CONTROLLED SERVICE PROVIDER and CONTROLLED SERVICE PROVIDER. Subsequently, one can consider the INDEPENDENT CONTROLLER and CONTROLLING SERVICE PROVIDER patterns as potential controllers for a CONTROLLED SERVICE PROVIDER. Additionally, a COORDINATOR can further balance the forces and optimize a concrete coordination scenario.

In contrast to the SPLIT pattern, there are no common event information needs in the JOIN pattern, which implies that a COORDINATOR does not decrease the overall coordination complexity. Similarly, designing the after-join service as a Service Provider controlled by CONTROLLING SERVICE PROVIDER does not simplify the coordination scenario, because the after-join service needs event information available at several services (all before-join services).

3.4 Evaluation

Since every business process consists of sequence dependencies, the best way of evaluating the patterns presented in this chapter is by using the patterns for sequence management as a means to automatically generate coordination scenarios (in the form of BPEL (OASIS, 2007) processes) from a business

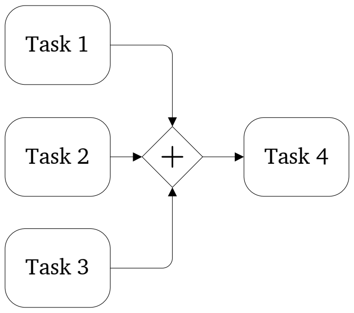


Figure 3.28: SYNCHRONIZATION pattern represented in BPMN

process specification (in the form of a BPMN process (OMG, 2010a)). Such an implementation (presented in Chapter 6) also shows the practical utility of the pattern language, which is an essential evaluation aspect in design science (Hevner et al., 2004).

In design science, it is also important to critically analyze the artifact that is created (Hevner et al., 2004). In Subsection 3.4.1 we describe how the pattern language for managing sequence dependencies went through a critical analysis.

Finally, in Subsection 3.4.2 we describe how the pattern language provides building blocks for every possible coordination scenario and hence answers our first research question (see Section 1.2 in Chapter 1)

3.4.1 Shepherding and writers' workshop

The patterns presented in this Chapter were critically analyzed in two phases. In the first phase the patterns went through a one-month shepherding process. This was an iterative process of review and revision, in which we collaborated with an experienced shepherd to significantly improve the value of the patterns. In the second phase the patterns were validated in a so called Writers' Workshop (Gabriel, 2002) on a EuroPLoP conference[3] (Monsieur, Snoeck, & Lemahieu, 2010b). This resulted into a lot of feedback and constructive suggestions from other pattern authors about how to improve the quality and validity of the patterns. During this workshop, all participating authors were able to give each other feedback on their work in a peer review session. We remained silent while the others discussed the patterns and explained additional insights and views they possess about patterns. This validation

[3]http://hillside.net/europlop

step concluded a first build-evaluate cycle (Hevner et al., 2004). The revised version of the patterns were presented in this chapter (see Section 3.2).

3.4.2 Completeness of the pattern language

As stated in our first research question (see Section 1.2 in Chapter 1), we aim to find a *systematic way* of composing coordination scenarios. The patterns presented in this chapter support this goal by providing fundamental building blocks that can be combined to construct every possible coordination scenario. In this subsection we first show that every interaction in a service composition can be composed by patterns described in this chapter. Subsequently, we show that the pattern language is also compatible with coordination styles such as centralized or decentralized coordination.

Combining patterns into interactions

As explained in our service composition meta-model (see Section 2.2.2 in Chapter 2) a service composition consists of a group of services that interact with each other using business requests and/or business event notifications. The services in a service composition either directly support business tasks or only function as 'glue' in the composition. The first kind of component services are referred to as *task* or *activity services* (Haesen, 2009). In what follows the services that function as the 'glue' are referred to as 'other services'. This means that in a composition one can make a distinction between eight ($= 2^3$) combinations of services and message types that are used in an interaction between two services S_x and S_y (see Table 3.2).

As shown in Table 3.2 five combinations of services and message types can be directly composed from the patterns presented in this chapter (see combinations 1, 2, 4, 5 and 6 in Table 3.2). In combinations 3 and 7 business requests are sent to an 'other service', which is not defined because per definition business requests are always sent to task services. The patterns can not be used as building blocks for the interaction described in combination 8, but event transfer between 'other services' is not part of the pattern language's scope. However, all patterns can be combined with any kind of event-based systems (e.g. a publish-subscribe system (Avgeriou & Zdun, 2005; Eugster et al., 2003)).

	S_x	S_y	Message Type	Patterns
(1)	Task Service	Task Service	Business Request	S_y is controlled by a CONTROLLING SERVICE PROVIDER (S_x)
(2)	Task Service	Task Service	Business Event Notification	S_y is a SELF-CONTROLLED SERVICE
(3)	Task Service	Other Service	Business Request	not defined
(4)	Task Service	Other Service	Business Event Notification	S_y is an INDEPENDENT CONTROLLER
(5)	Other Service	Task Service	Business Request	S_y is controlled by an INDEPENDENT CONTROLLER
(6)	Other Service	Task Service	Business Event Notification	S_y is a SELF-CONTROLLED SERVICE
(7)	Other Service	Task Service	Business Request	not defined
(8)	Other Service	Other Service	Business Event Notification	out-of-scope

Table 3.2: Combining requests and event notifications in service compositions using the patterns

Combining patterns into coordination styles

In Chapter 2 we have described orchestration and choreography as composition styles (see Subsection 2.2.4).

In an orchestration as composition style one service (the coordinator) interacts with all component services. There are no (explicit) interactions between component services; each interaction in the orchestration is between the coordinator and a component service. Typically, only the coordinator has knowledge of the business process and sends business requests to the component services. This composition style can be easily followed by applying the COORDINATOR pattern, which describes how one service sends multiple business requests to Service Providers.

In a choreography a business process is executed and coordinated by several peer-to-peer interactions among a group of collaborating services. Typically, several component services have knowledge about (part of) the business process and send business requests to each other. This style can be easily achieved by applying the CONTROLLING SERVICE PROVIDER pattern in which a task service *controls* (i.e. sends business requests) to another task service (i.e. the CONTROLLED SERVICE PROVIDER).

3.5 Conclusion

In this chapter we have presented a pattern language for managing sequence dependencies. As explained in the introduction to the pattern language (see Subsection 3.2.1) managing a sequence dependency is about distributing the business process knowledge among participants in a service composition. More specifically, a concrete coordination scenario needs to specify for each task in the business process which entity is responsible for sending a business request to the Service Provider supporting the business process task. The pattern language described in this chapter distinguishes between two solutions for manage a sequence dependency. Either one applies the CONTROLLED SERVICE pattern (see Subsection 3.2.3) or one goes for the SELF-CONTROLLED SERVICE (see Subsection 3.2.7). A CONTROLLED SERVICE can be controlled by either an INDEPENDENT CONTROLLER (see Subsection 3.2.4) or a CONTROLLING SERVICE PROVIDER (see Subsection 3.2.5). Additionally, this INDEPENDENT CONTROLLER or CONTROLLING SERVICE PROVIDER can be part of a COORDINATOR (see Subsection 3.2.6).

For each pattern we identified a set of consequences (e.g. with respect to loose coupling, flexibility, etc.). Based on these consequence we have presented concrete guidelines on how to combine the patterns to compose optimized coordination scenarios (see Subsection 3.3.1). In Subsection 3.3.2 we have applied the patterns and guidelines for managing sequence dependencies to the basic control flow patterns (Van der Aalst et al., 2003).

In Subsection 3.4.2 we first showed that every interaction in a service composition can be composed by the patterns presented in this chapter. Subsequently, we demonstrated that the pattern language is also compatible with coordination styles such as centralized or decentralized coordination.

Data is a precious thing and will last longer than the systems themselves.

– **Tim Berners-Lee** (1955),
British computer scientist,
Inventor of the World Wide Web

4 Managing data dependencies

In this chapter we present our pattern language that can support service composers when designing coordination logic that manages data dependencies in a service composition.

The chapter starts with a concrete example that shows the problem of managing data dependencies (see Section 4.1). Subsequently, in Section 4.2 the different patterns are presented. In Section 4.3 it is explained how the patterns presented in this chapter can be combined to manage data dependencies in an optimized way. The section also includes concrete design guidelines, including three decision trees that help to select the most appropriate patterns. In Section 4.4 the practical utility of the pattern language is demonstrated using three design science evaluation methods, which includes an application of the pattern language in a real-life business case. Subsequently, in Section 4.5 it is shown that every possible coordination scenario can be composed using the patterns presented in this chapter.

Finally, the chapter ends with a brief conclusion (see Section 4.6).

4.1 Introductory example

In this section we present a small example of service composition in hospitals, which we will use as a running example. The example is substantially the same as the running example described in Subsection 2.3.1 of Chapter 2. However, in this Section we present the example in a broader context so that it is clear how the problem of managing data dependencies can be situated in the process of implementing and coordinating a complete business process.

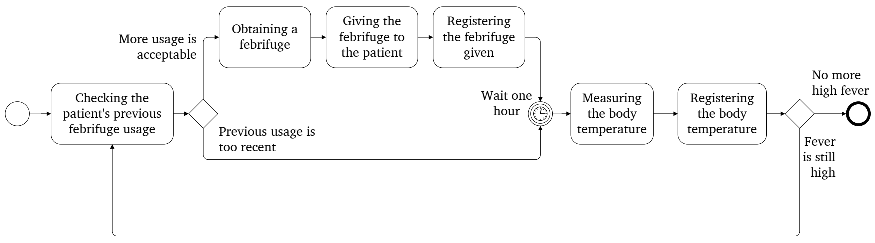

More usage is acceptable
Obtaining a febrifuge
Giving the febrifuge to the patient
Registering the febrifuge given
Checking the patient's previous febrifuge usage
Previous usage is too recent
Wait one hour
Measuring the body temperature
Registering the body temperature
No more high fever
Fever is still high

Figure 4.1: A business process for taking care of patients with high fever (represented using BPMN (OMG, 2010a))

In a hospital nurses provide several (business) services to patients, such as taking care of patients with high fever. In what follows, this service is referred to as the 'treating fever service'. A business process that supports this service, which can be used as a blueprint for a service-based system, could consist of the following tasks: checking the patient's previous febrifuge usage in the medical records, obtaining a febrifuge, giving the febrifuge to the patient, measuring the body temperature and registering the body temperature. In Figure 4.1 this business process is represented using BPMN (OMG, 2010a).

The business process can be implemented by consuming several services. As we will describe below, the treating fever service is composed of four main services: the medical records service, the pharmacist's service, the doctor's service and the nurse's service. In this example we assume that the nurse plays the role of service composer. The medical records service must be consumed for retrieving information concerning previous usage of febrifuges, registering the febrifuge given and registering the current body temperature. Obtaining a febrifuge also requires the consumption of services. In particular the nurse should request a febrifuge from the pharmacist, who of course also provides several services to the hospital staff. Hence, the nurse can be considered as a service composer that needs to consume the service of a pharmacist. Both aspirin and paracetamol are fever reducers. However, aspirin has the unpleasant side effect that it can cause stomach bleeding in certain circumstances. Therefore, the pharmacist needs information concerning the risk for stomach bleeding, before he or she can deliver an appropriate febrifuge. The risk for stomach bleeding is only known by the patient's doctor. This means that the doctor provides a second service that needs to be consumed in order to support the task of obtaining a febrifuge. The remaining tasks in the business process require consumption of the nurse's service(s). In this example, this means that the nurse can complete the tasks 'giving the febrifuge to the patient' and 'measuring the body temperature' tasks without the consumption of other services.

Although being a rather simple service composition, it already demonstrates the need for coordination. For example, coordination is required to ensure that the registration of a given febrifuge (consuming the medical records service) only occurs when a febrifuge is successfully given to a patient (consuming the nurse's service). Service coordination guarantees that sequence constraints as specified in the business process are met by constructing an appropriate control flow.

Dealing with data needs is usually also part of service coordination (see also Section 2.1 in Chapter 2). Services can require certain input data, which may in turn be the output from another service. Service coordination controls when which service is invoked, how input data is delivered and what to do with a service's output (Janssen & Feenstra, 2008). As such service coordination yields an appropriate data flow. For example, the pharmacist needs the information concerning the risk for stomach bleeding, which is held by the doctor. Hence, there is a data dependency between the pharmacist and the doctor. As a result, service interactions with the pharmacist and doctor must be coordinated, such that the pharmacist obtains the right information at the right time and in the right format.

Although this is a rather small example of a service composition in which needs to be dealt with data dependencies, many coordination scenarios are possible. In Chapter 2 we already presented several ways of managing the data dependency between the pharmacist and the doctor (e.g. figures 2.2(a) and 2.2(b)). However, as explained in detail in Chapter 2 (see Section 2.3) most approaches for dealing with data dependencies allow finding alternative data flows, but do not provide a systematic way of building different coordination scenarios (that can be used to construct every possible coordination scenario) nor do they analyze the advantages and disadvantages of alternatives (Monsieur, Snoeck, & Lemahieu, 2010a). Only a few studies about data dependency management take into account other aspects than performance that could influence the choice of a specific data flow such as data confidentiality, loose coupling or robustness to change. This can result in badly or suboptimally coordinated service compositions and service-based systems. Ultimately, these shortcomings hamper the effective and efficient design and automatic generation of coordination logic.

In this chapter we study all possible styles of coordinating services so that a service's data needs are fulfilled in an appropriate way. We give answers to questions like: What are the fundamental differences between two coordination scenarios (e.g. figure 2.2(a) versus figure 2.2(b))? What are advantages and disadvantages of specific coordination scenarios? How can a service composer construct an optimal coordination scenario?

4.2 Pattern language

4.2.1 Introduction

In this section we present three patterns that can be used to construct coordination scenarios that manage data dependencies. Each pattern deals with a specific recurring problem in data dependency management, in common context, and under influence of the same set of forces.

Common context

Analysing data dependency management requires precisely defined concepts. Therefore, we first introduce a terminology, which allows one to clearly name the participants that are relevant in a service composition that includes one or more data dependencies. We refer to an entity that consumes a particular service as a *Service Requester*. In a service composition this entity typically holds a (partial) description of the control flow, because it knows when to consume a specific service. Hence, in the context of the patterns for managing sequence dependencies presented in Chapter 3, a Service Requester is a controller. The entities of which the services are consumed for the realization of a composite service are called *Service Providers*. Possibly, a Service Provider needs certain data for processing the Service Requester's request. We refer to this kind of Service Provider as a *needy* Service Provider. The service that can provide the data needed is referred to as the *Data Provider*[1]

In summary, the previous definitions imply that a data dependency as defined in Chapter 2 (see Section 2.1) is always related to a Needy Service Provider and a Data Provider. In the context of the hospital example we can consider the nurse as a Service Requester, that sends a request to the service provided by the pharmacist. Hence, the pharmacist plays the role of a Service Provider. Since the pharmacist needs information that is known by the doctor, the pharmacist can be considered as a Needy Service Provider and the doctor can be labeled as a Data Provider. In this example the Data Provider is also the data owner. However, as defined above, a Data Provider can be the data owner or a data mediator that forwards data requests to other Data Providers. In this example a possible data mediator could be the doctor's assistant.

[1]In our pattern language we abstract from the fact that a Data Provider can be either the entity that owns the data or the entity that functions as a data mediator (Gamma et al., 1995) that can forward data requests to the right entities (e.g. other Data Providers or data owners). A Data Provider can correspond to a group of data mediators and data owner, in which any request and/or transmission of data can flow via either the data mediator or the data owner.

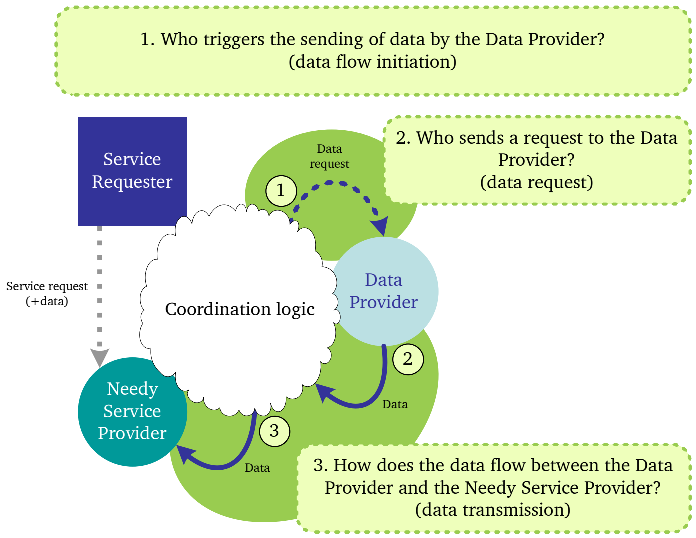

1. Who triggers the sending of data by the Data Provider?
(data flow initiation)
Service Requester
Data request
1
2. Who sends a request to the Data Provider?
(data request)
Service request (+data)
Coordination logic
Data Provider
2
Data
Needy Service Provider
3
Data
3. How does the data flow between the Data Provider and the Needy Service Provider?
(data transmission)

Figure 4.2: Three questions that need to be answered by a specific coordination scenario that manages a data dependency

Three different problems

In general, a specific coordination scenario that manages a data dependency should answer three questions (see figure 4.2):

1. Who triggers the sending of data by the Data Provider? (*data flow initiation*)
   (e.g. Who triggers that the information on the risk for stomach bleeding must be sent by the doctor?)

2. Who sends a request to the Data Provider? (*data request*)
   (e.g. Who sends the request for the risk for stomach bleeding to the doctor?)

3. How does the data flow between the Data Provider and the Needy Service Provider? (*data transmission*)
   (e.g. How does the risk information get from the doctor to the pharmacist?)

For each question we have described a pattern that helps to answer the question in the most optimal way by taking advantages and disadvantages into account. Answers to these questions can replace the cloud in figure

4.2. As such these three patterns form the three building blocks for scenarios that manage a data dependency between a Needy Service Provider and a Data Provider (cfr. research question 1). While previous work on alternative data flows in service compositions is mainly focused on optimizing the performance (i.e. reducing communication overhead, etc.) (see Section 2.3 in Chapter 2), our approach analyzes the advantages and disadvantages of each design alternative by considering *multiple* evaluation criteria or pattern forces, including robustness to change, loose coupling and data confidentiality. In each pattern several solutions to the same problem are described[2]. The criteria are used to evaluate the solutions in each pattern. In each pattern several solutions are presented for a specific problem. In each solution presented in one pattern the forces are balanced differently, resulting into different solution evaluations. By giving a weight to the evaluation criteria, service composers can determine the optimal coordination scenario (research question 2).

Common forces (evaluation criteria)

In contrast to previous work on alternative data flows in service compositions, which is mainly focused on optimizing the performance (see Section 2.3 in Chapter 2), our approach analyzes the advantages and disadvantages of each design alternative by considering multiple evaluation criteria or pattern forces. In this thesis we do not aim to exhaustively identify all relevant forces. Rather, we refer to common and frequently used forces in the literature. We summarize eight evaluation criteria (EC) that were discussed in the literature (e.g. (Balasooriya et al., 2005; Barros et al., 2005; Erl, 2007; Goethals, 2008; Haesen et al., 2006; Zirpins et al., 2004)):

EC1 *Robustness to change*: In a service-oriented environment it is critical that the propagation of changes due to the modification of the interface of a Service Provider is minimized. Consumers prefer to rely on a Service Provider that only rarely changes its interface. A change in the data requirements should minimally change the way the Service Provider is consumed. This evaluation criterion is related to the service design principle called service reusability, because a service can be considered more reusable if it has a relatively simple and stable interface (Erl, 2007).

[2]Our pattern language for managing sequence dependencies contains different patterns that all describe one solution to one and the same problem. In contrast, our pattern language for managing data dependencies consists of three patterns, each addressing one specific problem. Furthermore, each pattern describes different solutions to one (pattern-specific) problem. In the literature one can find both kinds of patterns (Paikens & Arnicans, 2008).

EC2 *Adjustability*: A specific coordination scenario consists of a set of service interactions so that data available at the Data Provider is sent to the Service Provider. This criterion is about the ability to change which data is sent to the Service Provider in function of a specific service request. For example, in coordination scenarios to manage the data dependency between the pharmacist and the doctor this criterion can be used to make a distinction between coordination scenarios in which information regarding a *specific* patient is sent to the pharmacist and coordination scenarios in which information regarding *multiple* patients are sent to the pharmacist. Depending on the specific business context, a certain level of adjustability can be desired. For example, in the hospital setting efficiency issues can motivate coordination scenarios with high adjustability, so that pharmacists only receive information they really need (e.g. patient specific information instead of information regarding multiple patients).

EC3 *Coupling with Data Provider*: In some situations the data needed is not always provided by the same Data Provider. Each time a service takes over the role of Data Provider the party that is sending data requests to the Data Provider needs to be notified and modified properly. Similarly, each change in the interface of the Data Provider, requires a change in the implementation of the party that is interacting with the Data Provider. Therefore, a common principle in service design called loose coupling is often applied (Erl, 2007). This means that preferably a Service Provider's implementation does not have to rely on several other services.

EC4 *Data provider accessibility*: Sometimes it is possible that the Service Provider does not know *which* service can provide the required data (e.g. the pharmacist does not know who is the patient's doctor). In other cases it can occur that the Service Provider does not have *access* to the specific Data Provider (e.g. the pharmacist does not have the phone number of the patient's doctor or is not allowed to call the doctor directly). However, it can also occur that only the Service Provider has access to the right Data Provider (e.g. only the pharmacist can request information concerning a potential risk for stomach bleeding).

EC5 *Confidentiality of data requirements*: In order to complete its internal processing, a Service Provider needs data. It can occur that these data requirements are confidential (e.g. suppose that nurses cannot have insight into the pharmacist's internal decision processes), which means that only a limited set of services or even only the Service Provider itself knows which data is needed in the process of delivering its service. This evaluation criterion is related to the service design principle called service abstraction, because it is about hiding information about the Service Provider's data requirements (Erl, 2007).

EC6 *Data confidentiality*: When requesting a Data Provider to send the required data to an entity, it is important to realize that the provided data can be confidential and therefore there can exist a need to limit the number of entities that the Data Provider can share the data with. For example, a Data Provider can demand that the provided data is only sent to the entity (e.g. a Service Provider) that needs the data and that it cannot be shared with other Service Providers or the Service Requester (Goethals, 2008).

EC7 *Data reusability*: In some business cases data provided by a Data Provider is used by more than one Service Provider. In such situations an optimal coordination scenario limits the number of data requests that are sent to the Data Provider.

EC8 *Data format*: When the Data Provider replies, the data that is provided is possibly not in a form that is expected by the Service Provider. For example, the data format needs to be adapted, or the data should be made anonymous. In short, in some cases data transformations are desirable before the data is received by the Service Provider. Dealing with different data formats is a common challenge when information is shared among services (Goethals, 2008).

4.2.2 Pattern overview

To manage a data dependency, we use three patterns: DATA FLOW INITIATION (see Subsection 4.2.3), DIRECT-INDIRECT REQUEST (see Subsection 4.2.4) and DIRECT-INDIRECT DATA TRANSMISSION (see Subsection 4.2.5). Each pattern consists of several types of solutions, among which a service composer can choose by considering the evaluation criteria and solutions' consequences. In Figure 4.3 each pattern is visualized in a box (dashed line border) containing both the pattern name and sub-boxes referring to different types of solutions in that pattern. The relationships between the three patterns are indicated by arrows. An application of the DATA FLOW INITIATION pattern can function as a first necessary step in managing data dependencies. The next steps toward a coordination scenario is indicated by means of the arrows in Figure 4.3. An arrow pointing from a pattern sub-box A to a pattern box B indicates that the pattern represented by B should be applied next when a pattern is applied in the way represented by sub-box A. For example, there is an arrow that indicates that the DIRECT-INDIRECT REQUEST pattern should be applied when an active Service Provider is chosen after applying the DATA FLOW INITIATION

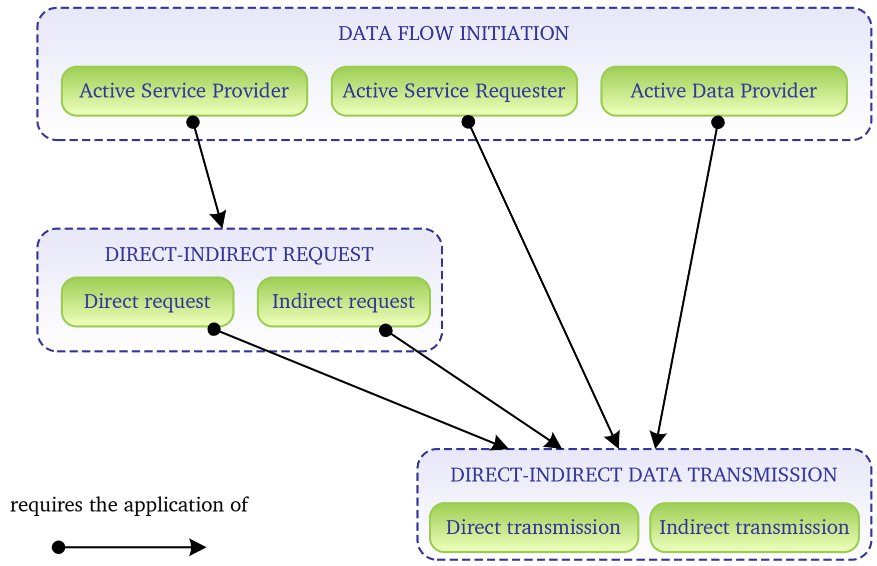


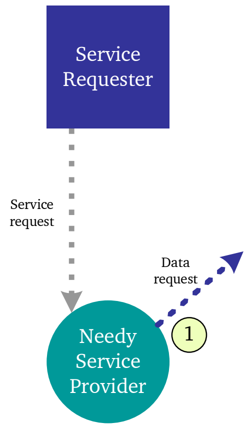
Figure 4.3: Relationships between the three patterns

pattern. Although theoretically the three patterns can be applied in any order, the order shown in Figure 4.3 is the most intuitive one and will therefore be used in the rest of this chapter.

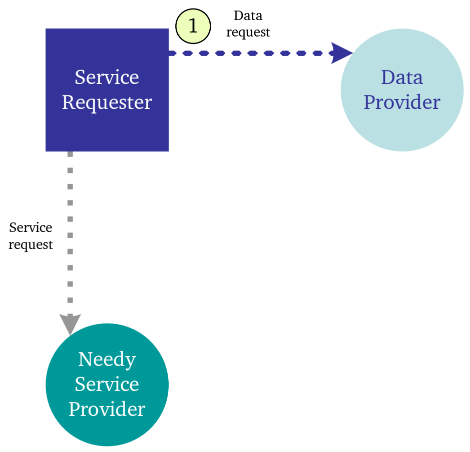
4.2.3 Data flow initiation

Problem

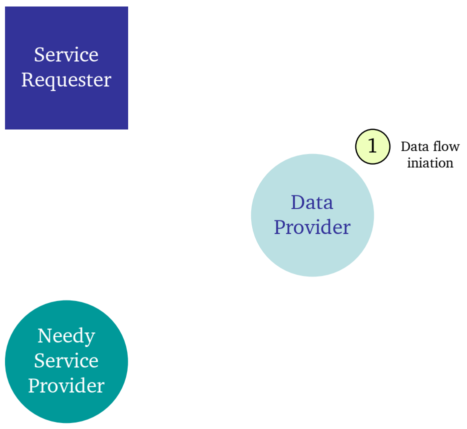
If a Service Requester sends a request to a Service Provider, it can occur that the Service Provider does not possess sufficient data for completing its internal processing. Therefore additional input data should be collected. This task is accomplished by the data flow. For example, when a nurse asks the pharmacist a febrifuge for a certain patient, it can occur that the pharmacist needs more input data (e.g. the risk for stomach bleeding). This raises an important question regarding the data collection process: **Who triggers the sending of data by a Data Provider?**



(a) Active Service Provider



(b) Active Service Requester



(c) Active Data Provider

Figure 4.4: Data flow initiation

Solutions

There are three possible data flow initiators in a coordination scenario (see Figures 4.4(a),4.4(b), and 4.4(c)). An ACTIVE SERVICE PROVIDER initiates the data flow by sending out a data request (see step one in Figure 4.4(a)). It is not specified to which entity the Service Provider sends out its data requests. This problem is discussed in one of the other patterns (see DIRECT-INDIRECT REQUEST in subsection 4.2.4). An ACTIVE SERVICE REQUESTER initiates the data flow by sending a data request to the Data Provider (see step one in Figure 4.4(b)). While in coordination scenarios with an ACTIVE SERVICE PROVIDER or an ACTIVE SERVICE REQUESTER the Data Provider sends out data upon request of an entity (i.e. upon the request of the Service Provider or Service Requester), in a scenario with an ACTIVE DATA PROVIDER it is the Data Provider itself that triggers the sending of data (see Figure 4.4(c)).

Consequences (evaluation of the solutions)

For each force discussed in Section 4.2.1 we can evaluate each solution presented in Section 4.2.3.

EC1 *Robustness to change*: In case of an ACTIVE SERVICE REQUESTER or an ACTIVE DATA PROVIDER every change in the Service Provider's data requirements results in a change in the implementation of the Service Requester or Data Provider. In contrast, in the ACTIVE SERVICE PROVIDER scenario these changes are only reflected in modified data requests sent by the Service Provider itself. Therefore, consumption of ACTIVE SERVICE PROVIDERS is considered to be rather stable.

EC2 *Adjustability*: An ACTIVE SERVICE REQUESTER sends both a service request to the Service Provider and a data request to the Data Provider. Hence, it is obvious that an ACTIVE SERVICE REQUESTER can adjust the data request to a specific service request. An ACTIVE SERVICE PROVIDER can also adjust the data request to a specific service request, because it receives, per definition, service requests from the Service Requester. In contrast, in ACTIVE DATA PROVIDER scenario control and data flow are always separated (i.e. neither the Service Requester nor the Service Provider is sending data requests to the Data Provider), which means the data sent by the Data Provider can not be changed in function of a specific service request.

EC3 *Coupling with Data Provider*: It is clear that an ACTIVE SERVICE PROVIDER is coupled with the external world, because it needs to send out data requests to known external parties. In contrast, in a case of an ACTIVE SERVICE REQUESTER and ACTIVE DATA PROVIDER the Service Provider simply expects that the data is provided at some point in time. In such scenarios Service Providers do not have to initiate interactions with external parties (for input data purposes). An ACTIVE SERVICE REQUESTER has a coupling with the Data Provider, but this can possibly be considered more acceptable because Service Requesters are also strongly coupled with Service Providers that need to be triggered. An ACTIVE DATA PROVIDER implies a looser coupled Service Requester and Service Provider. However, as a consequence an ACTIVE DATA PROVIDER is more coupled with the external world, because it autonomously sends out data instead of sending data upon request.

EC4 *Data provider accessibility*: Since an ACTIVE SERVICE REQUESTER needs to send a data request to the Data Provider, an ACTIVE SERVICE REQUESTER is not appropriate when only the Service Provider has access to the Data Provider.

EC5 *Confidentiality of data requirements*: As discussed in the previous evaluation criterion, an ACTIVE SERVICE REQUESTER needs to send a data request to the Data Provider. However, if the data requirements are confidential and are only known by the Service Provider itself, an ACTIVE SERVICE REQUESTER is not appropriate. As discussed in the adjustability criterion, an ACTIVE DATA PROVIDER cannot send data that is adjusted to a specific service request. As a consequence, an ACTIVE DATA PROVIDER often needs to send a larger amount of data (e.g. information regarding multiple patients). This can be favorable because in this way the specific data requirements are not known by the Data Provider.

EC6-8 *Data confidentiality, reusability and format*: This pattern only deals with the initiation of the data flow (see problem definition in 4.2.3). It does not describe anything about the data itself or the data transmission between services. Therefore, evaluation criteria EC6-8 are not relevant for the evaluation of this pattern.

Table 4.1 summarizes all consequences of the DATA FLOW INITIATION pattern.

Relationship with other patterns

An ACTIVE SERVICE PROVIDER sends out data requests in order to receive the missing input data (see step one in Figure 4.4(a)). The DIRECT-INDIRECT REQUEST pattern shows who contacts the actual Data Provider with the request (see 4.2.4). An ACTIVE SERVICE REQUESTER sends data requests to

	Active SR	Active SP	Active DP
Robustness to change	-	+	-
Adjustability	+	+	-
Coupling with Data Provider	SR coupled	SP or SR coupled	no coupling
Data provider accessibility	SR needs access	SP or SR needs access	no access required
Confidentiality of data requirements	-	*depends on request*	+
Data confidentiality		*depend on data transmission*	
Data reusability		*depend on data transmission*	
Data format		*depend on data transmission*	

SR = Service Requester
SP = Service Provider
DP = Data Provider

Table 4.1: Summary of the consequences of DATA FLOW INITIATION

the Data Provider (see step one in figure 4.4(b)). As a consequence the Data Provider sends out data. In case of an ACTIVE DATA PROVIDER the Data Provider itself decides if it needs to send out data. The DIRECT-INDIRECT TRANSMISSION pattern shows how the data flows from the Data Provider to the Needy Service Provider (see Subsection 4.2.5).

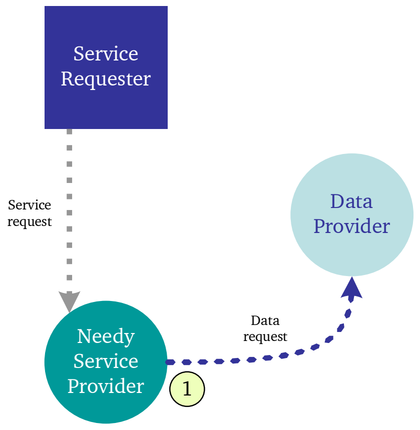
4.2.4 Direct-Indirect request

Problem

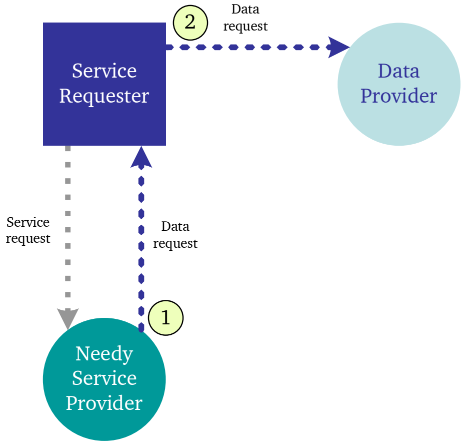
If a Service Provider is active, then the Service Provider sends out data requests in order to receive missing input data. This raises the following question: **Where can an active Service Provider send its data requests to?** For example, if a pharmacist wants to inform himself about the risk for stomach bleeding, the pharmacist needs to know who he can ask this question to. Should he ask the nurse or can he ask the doctor?

Solutions

An ACTIVE SERVICE PROVIDER can send its data requests to two entities, as shown in Figures 4.5(a) and 4.5(b). Firstly, an ACTIVE SERVICE PROVIDER can send a *direct request*, which means that the data request is sent directly to the Data Provider. Secondly, an ACTIVE SERVICE PROVIDER can send its data request to the Service Requester (see step two in Figure 4.5(b)), which is supposed to forward the data request to the appropriate Data Provider (see step three in figure 4.5(b)). This alternative is referred as an *indirect request*.



(a) Direct request



(b) Indirect request

Figure 4.5: Direct request versus indirect request

Consequences (evaluation of the solutions)

For each force discussed in section 4.2.1 we can evaluate each solution presented in Section 4.2.4:

EC1-2 *Robustness to change* and *adjustability*: This pattern deals with ACTIVE SERVICE PROVIDERS (see problem definition in 4.2.3). Hence, both solutions presented in this pattern, which both include ACTIVE SERVICE PROVIDERS, score equally on these evaluation criteria. As discussed in the evaluation of the DATA FLOW INITIATION pattern (see section 4.2.3), ACTIVE SERVICE PROVIDERS lead to robust and adjustable coordination scenarios.

EC3 *Coupling with Data Provider*: In the direct request scenario there is a strong coupling between the Service Provider and the Data Provider. Sending data requests to the Service Requester, as in the indirect request scenario, removes this coupling. However, note that in the indirect request scenario there is a coupling between the Service Requester and the Data Provider. Perhaps this can be considered more acceptable because Service Requesters are also strongly coupled with Service Providers that need to be triggered. In the direct request scenario only the Service Provider needs to know which service plays the role of Data Provider, while the indirect request scenario requires that the Data Provider is known by the Service Requester.

EC4 *Data provider accessibility*: The direct request scenario requires that the Data Provider is known by the Service Provider, while in the indirect request scenario only the Service Requester needs to know which service plays the role of Data Provider. Similarly, the direct request scenario requires that the Data Provider can be accessed by the Service Provider, while in the indirect request scenario only the Service Requester needs to have access to the Data Provider.

EC5 *Confidentiality of data requirements*: In the indirect request scenario the Service Requester needs to send a data request to the Data Provider. However, if the data requirements are confidential and are only known by the Service Provider itself, an active Service Provider with indirect request is not appropriate.

EC6-8 *Data confidentiality, reusability and format*: This pattern only deals with data requests (see problem definition in 4.2.4). It does not describe anything about the data itself or the data communication between services. Therefore, evaluation criteria EC6-8 are not relevant for the evaluation of this pattern.

Table 4.2 summarizes all consequences of the DIRECT-INDIRECT pattern.

	Direct request	Indirect request
Robustness to change	(+)	(+)
Adjustability	(+)	(+)
Coupling with Data Provider	SP coupled	SR coupled
Data provider accessibility	SP needs access	SR needs access
Confidentiality of data requirements	+	-
Data confidentiality	*depend on data transmission*	
Data reusability	*depend on data transmission*	
Data format	*depend on data transmission*	

SR = Service Requester
SP = Service Provider
DP = Data Provider
(+) is inherited from Active SP

Table 4.2: Summary of the consequences of DIRECT-INDIRECT REQUEST

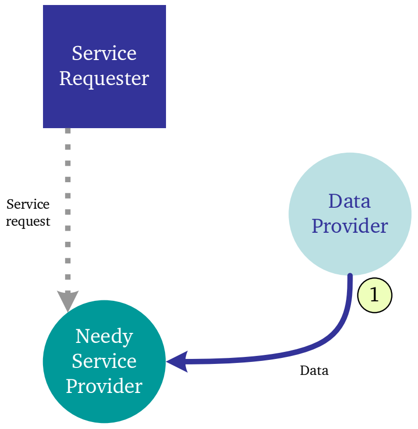
Relationship with other patterns

In both scenarios the Data Provider receives a data request (see step one in Figure 4.5(a) and step two in Figure 4.5(b)). As a consequence, data should be received by the Service Provider. The DIRECT-INDIRECT TRANSMISSION pattern shows how the data flows from the Data Provider to the Needy Service Provider (see Subsection 4.2.5).

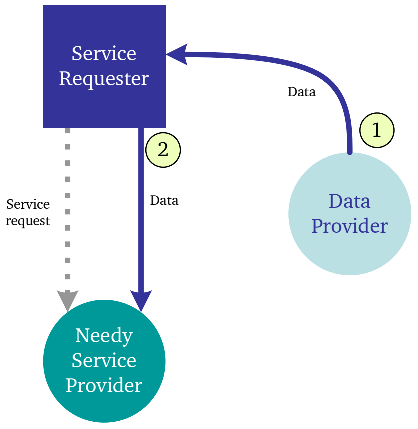
4.2.5 Direct-Indirect transmission

Problem

If a Data Provider received a data request, the requested data should be received by the Service Provider. If an ACTIVE DATA PROVIDER is used in a coordination scenario, the Service Provider should also receive data sent by the Data Provider. In both cases, the following question pops up: **How does the data flow from the Data Provider to the Service Provider?**



(a) DIRECT DATA TRANSMISSION



(b) INDIRECT DATA TRANSMISSION

Figure 4.6: DIRECT DATA TRANSMISSION versus INDIRECT DATA TRANSMISSION

Solutions

Data can flow from the Data Provider to the Service Provider in two ways. Firstly, the Data Provider can initiate a *direct data transmission,* which means that the data is sent directly to the Service Provider (see step one in Figure 4.6(a)). Secondly, the data can be transmitted from the Data Provider to the Service Requester (see step one in Figure 4.6(b)) and subsequently to the Service Provider (see step two in Figure 4.6(b)). This alternative is referred to as an *indirect data transmission* (see Figure 4.6(b)).

Evaluation of the solutions

For each force discussed in Section 4.2.1 we can evaluate each solution presented in Section 4.2.5.

EC1-4 *Robustness to change, adjustability, coupling with Data Provider* and *Data Provider accessibility*: When evaluating a complete coordination scenario using these evaluation criteria, this pattern has no influence on the evaluation. In other words, an evaluation of the solutions in this pattern totally depends on the specific solutions chosen in the other two patterns. For an evaluation regarding these criteria for the DATA FLOW INITIATION pattern and the DIRECT-INDIRECT REQUEST pattern, we refer to sections 4.2.3 and 4.2.4.

EC5 *Confidentiality of data requirements*: In the INDIRECT DATA TRANSMISSION scenario, all data that needs to be transmitted to the Service Provider is passed through the Service Requester. However, if the Service Provider's data requirements are confidential and can only be known by the Service Provider itself, INDIRECT DATA TRANSMISSION is not appropriate.

EC6 *Data confidentiality*: When the provided data is confidential, DIRECT DATA TRANSMISSION is the best scenario, since INDIRECT DATA TRANSMISSION implies that the data is passed through the Service Requester before it is received by the Service Provider.

EC7 *Data reusability*: INDIRECT DATA TRANSMISSION facilitates the reuse of the provided data. For example, the Service Requester only receives the specific data once, before distributing the same data to several Service Providers it interacts with.

EC8 *Data format*: INDIRECT DATA TRANSMISSION allows data transformations, since all data that need to be transmitted to the Data Provider is passed through the Service Requester. As such, the Service Requester can be responsible for data transformations. However, in a DIRECT DATA TRANSMISSION scenario the Service Requester is not involved when the data needs to be transmitted from the Data Provider to the Service Provider. As a consequence, intermediary data transformations are not possible.

Table 4.3 summarizes all consequences of the DIRECT-INDIRECT DATA TRANSMISSION pattern.

	Direct transmission	Indirect transmission
Robustness to change	*depend on initiation/request*	
Adjustability	*depend on initiation/request*	
Coupling with Data Provider	*depend on initiation/request*	
Data provider accessibility	*depend on initiation/request*	
Confidentiality of data requirements	*depends on initiation/request*	-
Data confidentiality	+	-
Data reusability	-	+
Data format	-	+

SR = Service Requester
SP = Service Provider
DP = Data Provider

Table 4.3: Summary of the consequences of DIRECT-INDIRECT DATA TRANSMISSION

4.3 Applying the patterns to construct coordination scenarios

4.3.1 Combining the patterns into coordination scenarios

As described in the introduction to our pattern language (see Subsection 4.2.1) the three patterns discussed above are building blocks that need to be combined to build coordination scenarios.

Since the DATA FLOW INITIATION pattern has three solutions, and the DIRECT-INDIRECT REQUEST pattern and DIRECT-INDIRECT TRANSMISSION pattern have two solutions, it is, in theory, possible to combine the patterns in twelve different ways. However, by following the pattern relationships that were discussed in Subsection 4.2.2 and shown in figure 4.3 only eight combinations are possible. This makes sense, because, per definition, only in a coordination scenario with an ACTIVE SERVICE PROVIDER it is relevant to decide whether the data request should be sent in a direct or indirect way.

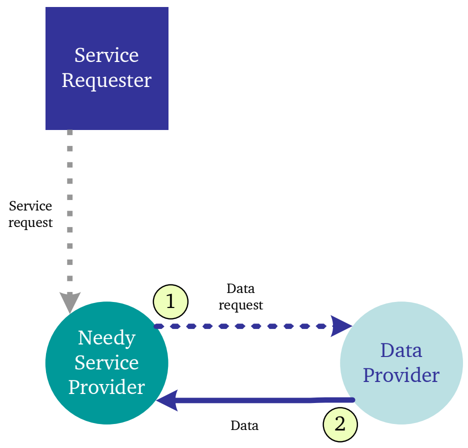


Figure 4.7: ACTIVE SERVICE PROVIDER with DIRECT REQUEST and DIRECT DATA TRANSMISSION

An ACTIVE SERVICE REQUESTER sends data requests to the Data Provider in a direct manner. ACTIVE DATA PROVIDERS do not receive data requests. In figures 4.7 to 4.14 the eight possible combinations are represented. Capitalized words in the figures' captions indicate how the patterns are applied.

4.3.2 Design guidelines for applying the patterns

As mentioned before, the consequences that are discussed in each pattern help service composers to decide on which solution is the most appropriate. As such service composers are guided in constructing a coordination scenario that is optimized according to the forces described in Subsection 4.2.1. In doing so, it can be helpful to consult the Tables 4.1, 4.2 and 4.3, that summarize all consequences related to solutions described by the patterns. However, although these tables are perfectly suitable for getting a quick view on how a specific solution deals with the forces, the tables do not explicitly guide one to the best solution. Therefore, based on the tables, we derived three decision trees, one for each pattern or table. Using decision trees the best solution described in a pattern can be easily found by simply following the different branches in the tree. In Figures 4.15, 4.16 and 4.17 decision trees are shown for DATA FLOW INITIATION, DIRECT-INDIRECT REQUEST and DIRECT-INDIRECT DATA TRANSMISSION, respectively. In the example described in the next subsection (see Subsectio 4.3.3) we show how these trees can be used when constructing an optimized coordination scenario.

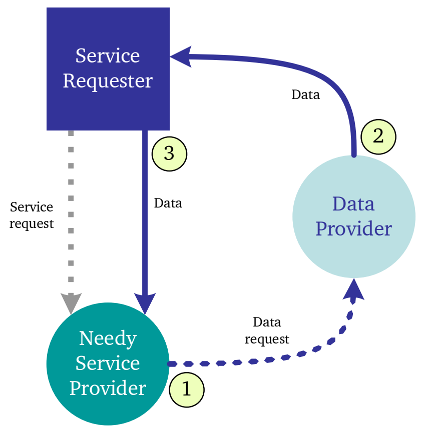


Figure 4.8: ACTIVE SERVICE PROVIDER with DIRECT REQUEST and INDIRECT DATA TRANSMISSION

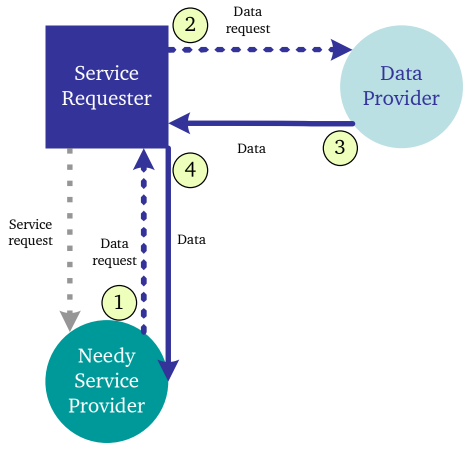


Figure 4.9: ACTIVE SERVICE PROVIDER with INDIRECT REQUEST and INDIRECT DATA TRANSMISSION

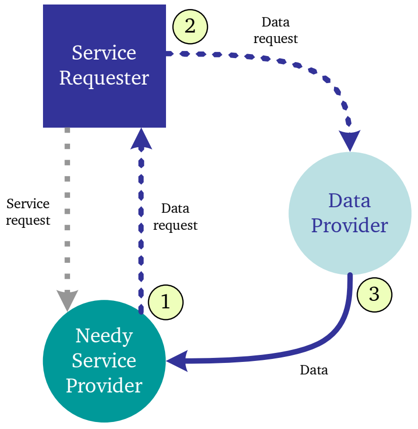


Figure 4.10: ACTIVE SERVICE PROVIDER with INDIRECT REQUEST and DIRECT DATA TRANSMISSION

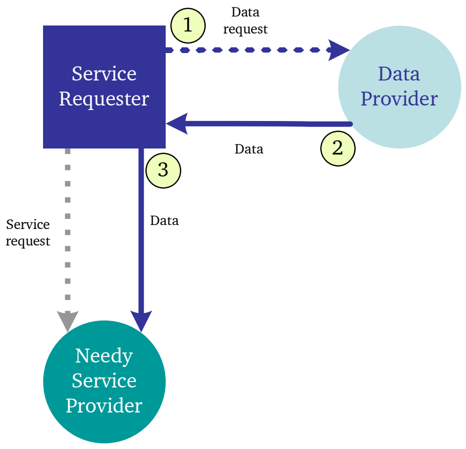


Figure 4.11: ACTIVE SERVICE REQUESTER with INDIRECT DATA TRANSMISSION

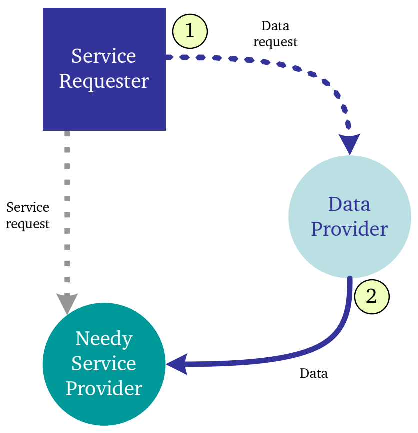


Figure 4.12: ACTIVE SERVICE REQUESTER with DIRECT DATA TRANSMISSION

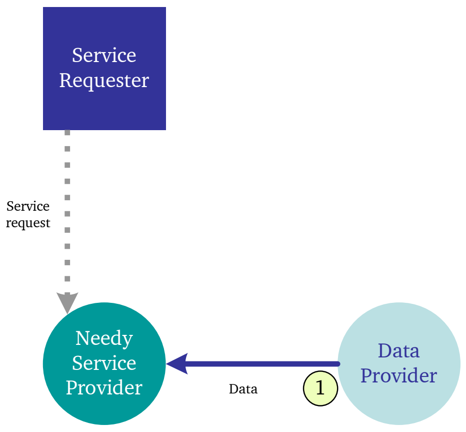


Figure 4.13: ACTIVE DATA PROVIDER with DIRECT DATA TRANSMISSION

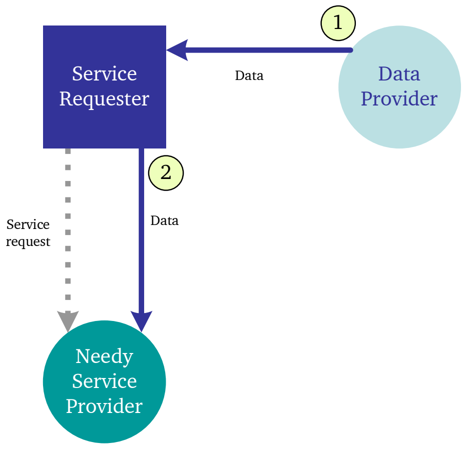


Figure 4.14: Active Data Provider with INDIRECT DATA TRANSMISSION

4.3.3 Applying the patterns and guidelines to the hospital example

By following the pattern relationships as described in 4.2.2 and shown in figure 4.3 we can construct an appropriate coordination scenario for the hospital example (see Section 4.1 for problem description):

- DATA FLOW INITIATION: Since neither nurses nor doctors want to understand which input data is required by the pharmacist, it is probably more desirable to choose an ACTIVE PHARMACIST. Nurses simply want to use some services provided by the pharmacist. Furthermore, it is not preferred that changes in data requirements result in changes to how the nurses work (or consume the pharmacist's services). Hence, in the decision tree shown in Figure 4.15 first the *robustness to change is necessary* branch needs to be followed. Subsequently, since the pharmacist does not know which doctor is treating the patient, the *SP has no access* and *SP not coupled* branches lead to an ACTIVE PHARMACIST.

- DIRECT-INDIRECT REQUEST: This pattern needs to be applied, because pharmacists are considered as active Service Providers. Since the pharmacist does not know which doctor is treating the patient, the *SP not coupled* and *SP has no access* branches needs to be followed in the decision tree shown in Figure 4.16. Furthermore, nurses are supposed to be *trusted* partners in a hospital setting, which means that the pharmacist's data requirements can be shared with nurses. Hence,

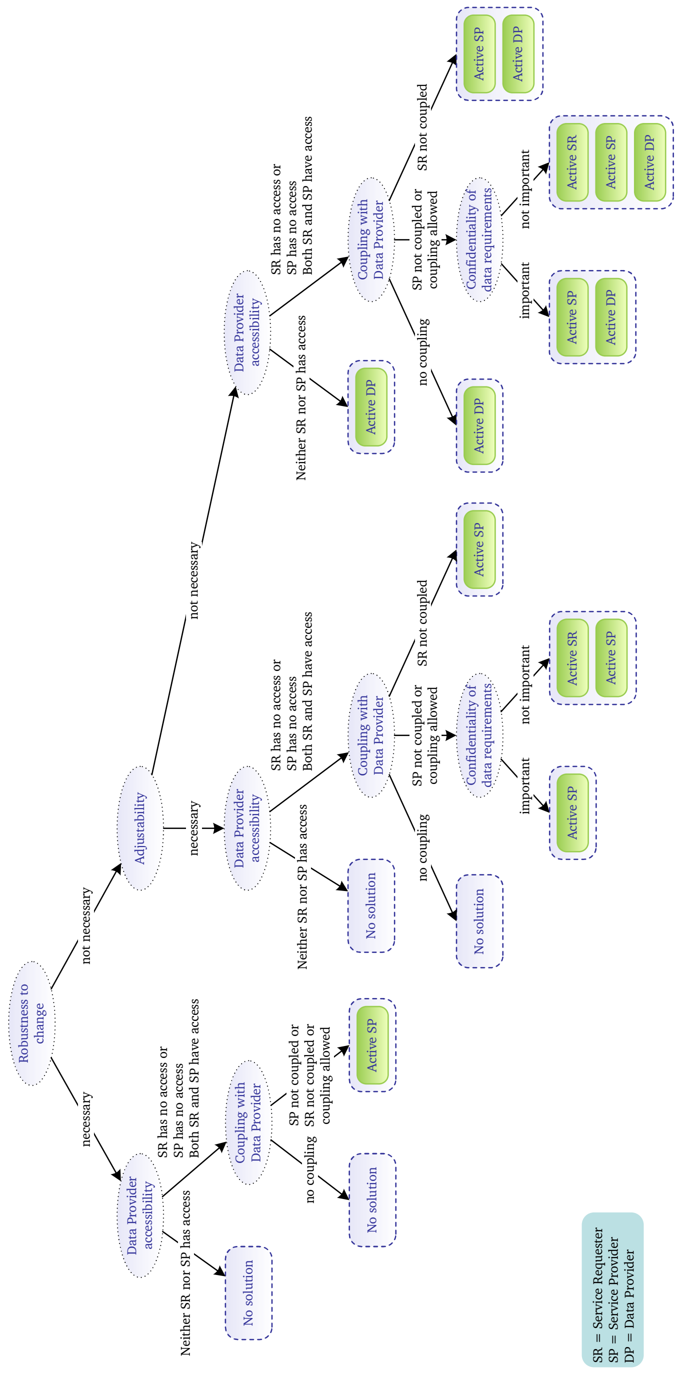


Figure 4.15: Decision tree DATA FLOW INITIATION

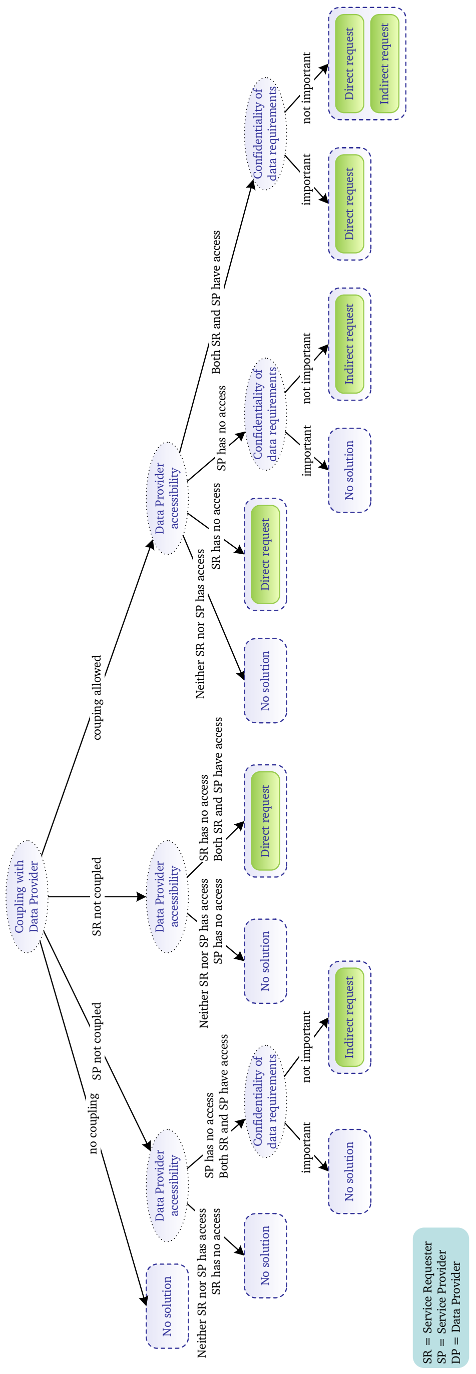


Figure 4.16: Decision tree DATA REQUEST

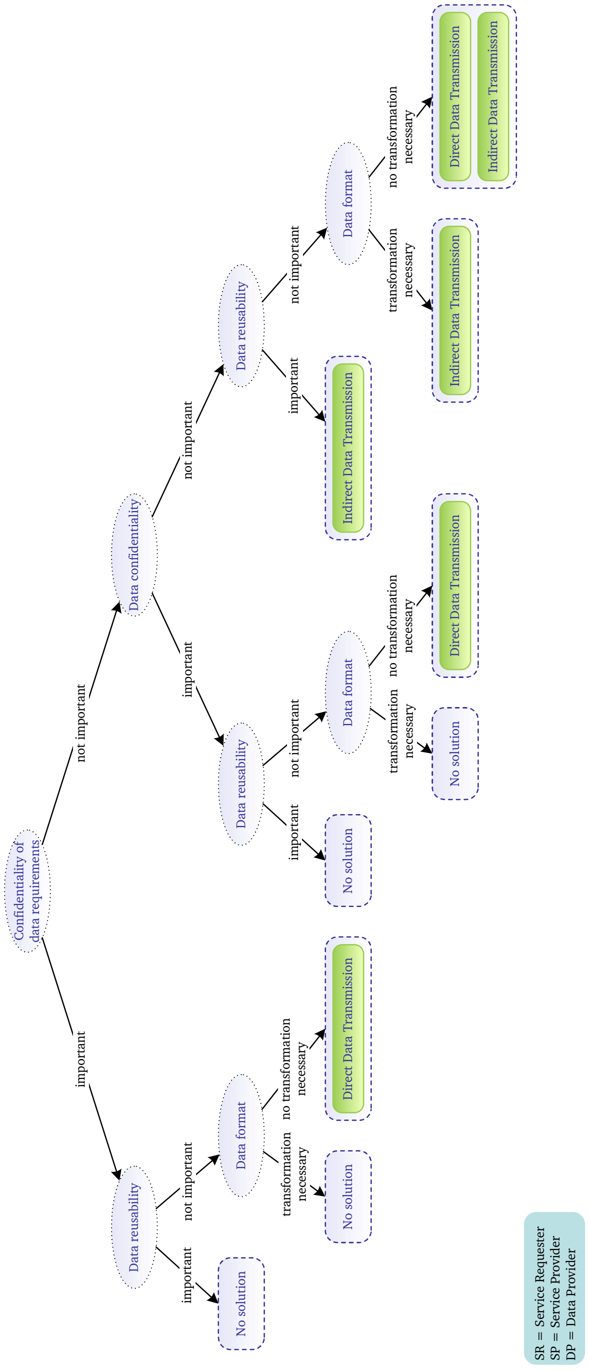


Figure 4.17: Decision tree DATA TRANSMISSION

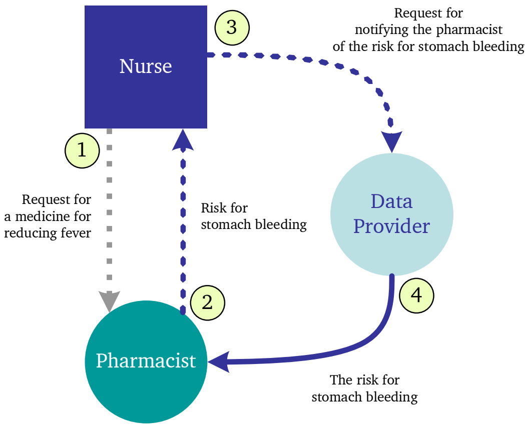

Nurse
3
Request for notifying the pharmacist of the risk for stomach bleeding
1
Request for a medicine for reducing fever
Risk for stomach bleeding
Data Provider
2
4
Pharmacist
The risk for stomach bleeding

Figure 4.18: ACTIVE PHARMACIST with INDIRECT REQUEST and DIRECT DATA TRANSMISSION

the *confidentiality of data requirements* leads to the INDIRECT REQUEST as the most appropriate solution. This means that the pharmacist asks the nurse for more information concerning the risk for stomach bleeding (see step two in Figure 4.18). Subsequently, the nurse can forward the request to the right doctor (see step three in Figure 4.18).

- DIRECT-INDIRECT DATA TRANSMISSION: Suppose the risk for stomach bleeding is quite confidential information that can not be shared with the nurse. Furthermore, as explained above the pharmacist's data requirements are not important, nor are data reusability and data format. Then, the decision tree in Figure 4.17 leads to the DIRECT DATA TRANSMISSION scenario as the best solution. Hence, the doctor should send the information concerning the risk for stomach bleeding directly to the pharmacist (see step four in Figure 4.18).

The complete solution for this example is shown in figure 4.18.

4.4 Demonstrating the practical utility

As described in Subsection 4.2.1 the use of our patterns, including the evaluation criteria, must help to determine the optimal coordination scenario (cfr. research question 2). In this Section the practical utility of our pattern language is demonstrated, using three design science evaluation methods, as proposed by Hevner et al. (2004):

1. *Analytical*: One of the first versions of the patterns was critically analyzed in two phases. In the first phase the patterns went through a one-month shepherding process, while in the second phase we participated in a Writer's Workshop (see Subsection 4.4.1).
2. *Observational*: After the analytical validation, we studied the artifact in depth in a business environment by means of a real-life case study with a Belgian bank and insurance company (see Subsection 4.4.2).
3. *Descriptive*: By using information from the knowledge base (i.e. relevant research) we built a convincing argument for the artifact's utility (see Subsection 4.4.3).

4.4.1 Analytical validation: a shepherding process and a writers' workshop

The patterns, including the evaluation criteria that help service composers to construct an optimal coordination scenario, were critically analyzed in two phases. In the first phase the patterns and evaluation criteria went through a one-month shepherding process. This was an iterative process of review and revision, in which we collaborated with experts from the IT world to significantly improve the value of the patterns. In the second phase the patterns were validated in a so called Writers' Workshop (Gabriel, 2002) on a PLoP conference[3] (Monsieur, Snoeck, & Lemahieu, 2009). This gave us a lot of feedback and constructive suggestions from other pattern authors about how to improve the quality and validity of the patterns. During this workshop, all participating authors were able to give each other feedback on their work in a peer review session. We remained silent while the others discussed the patterns and explained additional insights and views they possess about patterns. This validation step concluded the first build-evaluate cycle (Hevner et al., 2004) at the end of which the revised version of the patterns presented in this chapter was developed.

4.4.2 Observational evaluation: real-life insurance case

During the second build-evaluate cycle, we validated the patterns by means of a real-life business case at the Belgian KBC Banking & Insurance company (Haesen et al., 2006). The case can be considered a situation in which a consumer wants his house to be insured together with the house content.

[3]http://www.hillside.net/plop

A simplified version of the business process consists of the following tasks: processing the customer's request, presenting an offer to the customer, making the contract, sending the insurance policy to the customer, and payment by the customer. In the context of this thesis we only consider the first task, which deals with the processing of customer requests. The system supporting this task, which we refer to as the *insurance request management (IRM) service*, is composed of several (component) services: insurance quote service, sales service, customer information service, blacklist service and external information service. The main service that is consumed for this task is the *insurance quote service*. This service accepts or rejects the request and needs to calculate the insurance premium in case of acceptance. This yields a two-staged approach. The acceptance step investigates whether or not to accept the request for insurance. The second step is the tarification step which generates a price offer. The first step requires a substantial amount of data in order to evaluate all possible reasons for rejection. On the other hand, a minimum of data may be sufficient to provide the customer with a first, rough estimation of the price, purely for informational purposes. These data requirements explain why, besides the insurance quote service, several other services are involved when composing the IRM service. The insurance quote service needs to be combined with other services because of specific (data) needs:

1. **Information about existing customers**: The person who wants to have an object insured can be an existing or a new customer of the insurance company. For an existing customer most data will be available at the *customer information service*.

2. **Information about new customers**: All data about a new customer will have to be retrieved by the *sales service,* which interactively questions the customer.

3. **Data concerning descriptions of expensive items**: The premium of the house content depends on the fact whether the customer possesses exclusive and expensive goods, such as jewelry or special stamp collections. The premium increases proportionally to the value of those possessions. The data used to calculate the premium for the house content can be altered after the construction of the application. For example the premium for a stamp collection may initially only depend on the number of stamps in the collection. After examining past insurance claims, the insurance company may wish to consider also the exact kind of stamps for the premium calculation. This means further communication with the customer or interactions with an *external information service* containing price information of expensive objects, are needed.

4. **Information about blacklisted customers or fraudulent family members**: Before the insurance request is accepted, the insurance quote service needs information about possible fraudulent family members. Furthermore, the insurance quote service needs to check whether the customer is present on any blacklists of untrusted payers. This information can be retrieved from a third party service which is referred to as the blacklist service in the rest of this chapter.

5. **Base insurance quote**: Simplified, an insurance quote can be determined using a base insurance quote which is raised or reduced depending on the specific risk estimations. These risk estimations and the effect on the quote are calculated by the insurance quote service. The base insurance quote, however, is set by the sales service. Hence, in order to calculate the complete insurance quote, the base insurance quote must be retrieved from the sales service.

In summary, we can consider five main data needs, which all require some interaction with a service that needs to be included in the service composition. In terms of the terminology as used in this chapter, the insurance quote service plays the role of a Needy Service Provider, while the other services (e.g. sales service or third party service to check blacklists) play the role of the Data Providers. The entity that composes the IRM service is considered as the Service Requester, since it requests the insurance quote service. Below we show how the patterns presented in this chapter were applied to this real-life example in order to find an optimal solution. It allows us to find the most appropriate coordination scenarios for each data need. The patterns applied are indicated using small caps. Additionally, we refer to the criteria discussed in each pattern (see Subsections 4.2.3, 4.2.4 and 4.2.5) that lead up to the specific choice of coordination styles. Finally, for each data need the resulting coordination scenario is linked to one of the eight combinations that were discussed in Subsection 4.3.1.

1. **Information about existing customers**: Information about existing customers can be retrieved from the customer information service of the insurance company. Since this is rather stable data (see EC1 evaluation of the DATA FLOW INITIATION pattern in Subsection 4.2.3), that is strongly related to the specific insurance quote request (see EC2 evaluation of the DATA FLOW INITIATION pattern in Subsection 4.2.3) and one prefers loose coupling between the insurance quote service and other services (e.g. the customer information service) (see EC3 evaluation of the DATA FLOW INITIATION pattern in Subsection 4.2.3) it is better to choose for an ACTIVE INSURANCE QUOTE SERVICE REQUESTER. This means the insurance quote service simply expects to receive this

information, which is acceptable because it is rather stable data and consumers do not have to worry that the interface and required data are changing frequently. The loose coupling between the insurance quote service and other services is guaranteed, because the Service Requester, which is responsible for triggering the insurance quote service, should send out a request for customer data to the customer information service. Concerning the data transmission, two alternatives remain possible. Depending on the exact customer information that is needed, one could decide on the most appropriate solution. If the customer information needed is rather confidential, DIRECT DATA TRANSMISSION is better than indirect data transmission (see EC6 evaluation in Subsection 4.2.5). However, when the customer information service does not provide the data in the correct form, transformation by the Service Requester is needed, which justifies INDIRECT DATA TRANSMISSION via the Service Requester (see EC8 evaluation in Subsection 4.2.5). The application of these patterns results in the coordination scenarios represented in Figures 4.11 (based on indirect data transmission) and 4.12 (based on direct data transmission).

2. **Information about new customers**: For the information about new customers the patterns are applied in a similar way. The main difference is the service that provides the data, which can be either a sales service or perhaps the Service Requester. In any way, the insurance quote service prefers to have a loose coupling with this Data Provider, which (again) motivates the choice for an ACTIVE INSURANCE QUOTE SERVICE REQUESTER (see EC3 evaluation of the DATA FLOW INITIATION pattern in Subsection 4.2.3). Similar to the previous data need, both DIRECT DATA TRANSMISSION and INDIRECT DATA TRANSMISSION can be useful in certain situations. The application of these patterns results in the coordination scenarios represented in Figures 4.11 (based on indirect data transmission) and 4.12 (based on direct data transmission).

3. **Data concerning descriptions of expensive items**: For some customers additional data might be needed. For example, customers that have large collections of stamps need to be treated in a different way. Since they want to insure their valuable collections, a precise estimate of the value of the collection is needed to calculate the insurance premium. Therefore, an insurance quote service requires detailed descriptions of the stamps. Since this information is only needed in certain cases - only when the customer has exceptionally expensive items in his house - it is better to go for an ACTIVE INSURANCE QUOTE SERVICE (see EC1 evaluation of the DATA FLOW INITIATION pattern in Subsection 4.2.3). Since the insurance quote service does not have

any knowledge on where to get this information, an INDIRECT REQUEST is necessary (see EC4 in Subsection 4.2.4). Next, as with the other types of data above, for this data two sorts of data transmission are possible too. Firstly, in the case of confidential information (e.g. the value of the items is extremely high), it is better to choose DIRECT DATA TRANSMISSION (see EC6 evaluation in Subsection 4.2.5). Secondly, when transformation of the information is a priority above confidentiality, INDIRECT DATA TRANSMISSION via the Service Requester is more appropriate (see EC8 evaluation in Subsection 4.2.5). The application of these patterns results in the coordination scenarios represented in Figures 4.9 (based on indirect data transmission) and 4.10 (based on direct data transmission).

4. **Information about blacklisted customers or fraudulent family members**: Checking whether or not a customer (or a family member) is on any blacklist, is only a thing that the insurance quote service can do, because only this service knows where to get this information. Furthermore, only the insurance quote service has access to the blacklist service. This motivates the use of an ACTIVE INSURANCE QUOTE SERVICE, using DIRECT REQUESTS (see EC4 evaluation in Subsections 4.2.3 and 4.2.4). Since this data is rather confidential, DIRECT DATA TRANSMISSION is also more appropriate in this case (see EC6 evaluation in Subsection 4.2.5). These choices also support the fact that the insurance quote service prefers not to share its business rules (checking blacklists and/or family members) with its consumers (i.e. a Service Requester) (see EC5 in 4.2.3). The application of these patterns results in the coordination scenario represented in Figure 4.7.

5. **Base insurance quote**: For each type of insurance (car, house, etc.) there exists a base insurance quote, that is set by the sales service. Both the insurance quote service and the insurance quote Service Requester prefer a loose coupling with the sales service. Therefore, it is better to choose an ACTIVE SALES SERVICE that sends a set of base insurance quotes to the insurance quote service from time to time (e.g. each time the sales service decides to modify base insurance quotes) (see EC3 evaluation of the DATA FLOW INITIATION pattern in Subsection 4.2.3). Based on potential data transformation requirements, one can make a choice between DIRECT DATA TRANSMISSION and INDIRECT DATA TRANSMISSION. The application of these patterns results in the coordination scenarios represented in Figures 4.13 (based on direct data transmission) and 4.14 (based on indirect data transmission).

Hence, a final solution for the management of the data dependencies is constructed by combining several coordination scenarios, each taking care of a particular set of data. As explained above, the insurance quote Service Requester takes the role of an ACTIVE SERVICE REQUESTER with respect to customer data, while the insurance quote service takes the role of an ACTIVE SERVICE PROVIDER when it comes to information about the insured items or confidential background data about customers.

The conclusion of this validation exercise demonstrates that at least seven out of eight combinations prove to be useful in practice. Moreover, it also demonstrates that there is no 'one size fits all' solution. The ideal solution can only be obtained by considering the specific characteristics of data and applying the suitable pattern for each different set of data. This allows to balance the different requirements and meet several criteria at once. The solution developed by applying the patterns has been implemented at KBC Banking and Insurance as a new version in replacement of the existing version because of its improved stability (robustness to change), its capability of handling confidential data and its satisfying performance level. Furthermore, when data requirements are changing at KBC Banking and Insurance, the use of the patterns, including the guiding criteria, makes it easier to adapt the coordination scenarios than before when coordination logic was designed in an ad-hoc fashion. However, more research is needed to quantitatively evaluate that the use of the patterns contributes to a more efficient and effective development of coordination scenarios.

One combination of patterns, namely the scenario represented in Figure 4.8, has not been used in this real life case. Nevertheless, this does not imply that this pattern is useless. Since it is the result of a logical deduction step on the possible combination of the three basic patterns (ACTIVE SERVICE PROVIDER with DIRECT REQUEST and INDIRECT DATA TRANSMISSION), it has its place in the overview of potential solutions and might still prove useful in future real life cases.

4.4.3 Descriptive evaluation: flexible coordination of service interactions

For the descriptive evaluation of our research we used the 'informed argument' method (Hevner et al., 2004). We use the work by Zirpins et al. (2004) to build our convincing argument that the proposed approach to the management of data dependencies, consisting of patterns and evaluation criteria, is useful. In their article Zirpins et al. (2004) present which structural elements a useful approach should have. Below we will show how our approach follows the same structure and thus can be considered as useful and rigorous.

Zirpins et al. (2004) propose to make a distinction between the *logical* *dependencies* that are modeled by the interaction logic and the *operational coordination* that refers to the procedure or method that is utilized to enforce the logical dependencies. This closely matches the ideas in this chapter. In Chapter 2 (see Section 2.1) we identified logical dependencies in the form of sequence and data dependencies. Similarly, the operational coordination matches our vision on coordination, which is about managing the sequence and data dependencies. Zirpins et al. (2004) argued that while workflow processes represent the logical dependencies of interactions (i.e. causal and data relationships of message exchanges) they often simultaneously act as instructions for their coordination on the execution-level by distributed workflow management systems. As such, the coordination procedure emerges only implicitly as a side-effect of dependencies from the interaction logic and not because of application-specific reasons.

However, there are in most cases multiple alternatives for the enforcement of the abstract interaction logic. A reason for this is the multiplicity of possibilities for splitting the dependencies of the interaction logic into different partitions as well as the variety of alternatives for delegating parts of a partition to executive organizations for operational coordination. Therefore, Zirpins et al. (2004) suggest that a technical solution for service composition should consist of a combination of design and implementation patterns. A *design pattern* corresponds to the interaction logic that only specifies the generic process characteristics, while an *implementation pattern* refers to the refinement of the interaction logic that is needed for the concrete coordination of services. In the context of this thesis, the design pattern consists of *triggering the Service Provider*, *requesting the Data Provider*, *receiving the data from the Data Provider* and *sending the data to the Service Provider*. As discussed above these steps are required to manage the data dependency between a Needy Service Provider and a Data Provider. This design pattern is represented in Figure 4.19, using the notation by Zirpins et al. (2004). The notation, which is geared to usual workflow models, contains communication steps (circles) and transitions (arrows). Communication steps represent the sending of messages to an endpoint (e.g. Service Provider) either originating from (rcv) or going to (snd) a role that represents a participant.

In order to manage a data dependency, a concrete coordination process is needed. Zirpins et al. (2004) argue that implementation patterns are needed in order to build such concrete coordination processes. In this dissertation we propose such implementation patterns. Examples of these implementation patterns are represented in Figures 4.20(a) and 4.20(b), which correspond to respectively the combinations of patterns represented in Figures 4.11 and 4.10. As such, all possible coordination scenarios discussed in Subsection 4.3.1 function as implementation patterns. Zirpins et al. (2004) propose to

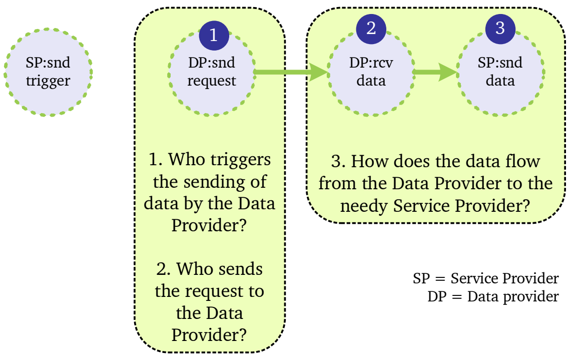

SP:snd trigger
1
DP:snd request
2
DP:rcv data
3
SP:snd data
1. Who triggers the sending of data by the Data Provider?
2. Who sends the request to the Data Provider?
3. How does the data flow from the Data Provider to the needy Service Provider?
SP = Service Provider
DP = Data provider

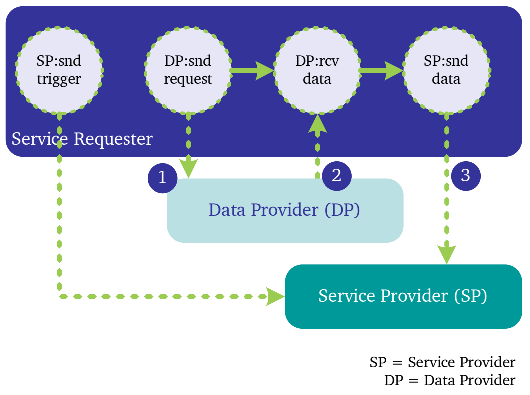
Figure 4.19: Interaction procedure

specify criteria for the choice of the most appropriate coordination pattern in so called coordination policies. A *coordination policy* describes the effect of a coordination variant in terms of specific (non-functional) service properties and thereby controls the choice of alternatives. In that way, the different patterns presented in this chapter, including the evaluation criteria for certain coordination styles and the different combinations that can be made by the patterns, can be considered as coordination policies. They support the transformation of the interaction logic needed for input data provisioning into concrete coordination processes.

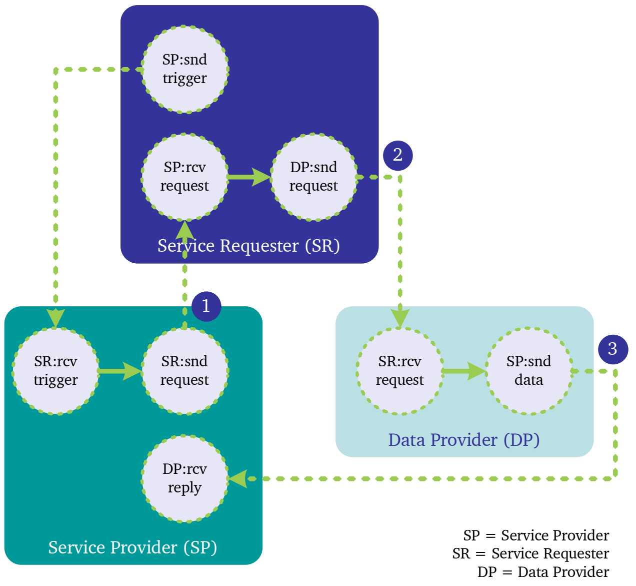
In summary, we can conclude that we have developed both implementation patterns and coordination policies (Zirpins et al., 2004). The implementation patterns allow us to find all possible coordination scenarios, and the coordination policies make it possible to construct the most appropriate coordination scenario in a certain business context. Hence, our approach to the management of data dependencies follows the structure proposed by Zirpins et al. (2004).

4.5 Completeness confirmation

Our patterns are innovative in the sense that these patterns are basic building blocks that can be combined to compose complete 'coordination scenarios'. A coordination scenario then shows the actions taken by all involved parties to get the data from the Data Provider to the Needy Service Provider. Hence, the real value of the patterns depends on an evaluation of the *composition* of the patterns into concrete coordination scenarios. In this section we show that *all*



(a) ACTIVE SERVICE REQUESTER with INDIRECT DATA TRANSMISSION (see figure 4.11)



(b) ACTIVE SERVICE PROVIDER with INDIRECT REQUEST and DIRECT DATA TRANSMISSION (see figure 4.10)

Figure 4.20: Two coordination scenarios visualized as implementation patterns (Zirpins et al., 2004) (the numbers match the numbers used in figures 4.10 and 4.11)

potential coordination scenarios can be composed by combining the patterns (cfr. research question 1). For this completeness confirmation, we did not start from the aforementioned questions that form the basis for our three patterns (see introduction to the pattern language in Subsection 4.2.1). Rather, we declaratively specified what a coordination scenario should accomplish and in which message exchanges a Service Requester, Service Provider and Data Provider can be involved. For example, it is easy to understand that in every coordination scenario the Service Provider must receive data from another entity. More conditions that should be met by a coordination scenario are discussed in the rest of this section.

We have used Prolog (Clocksin & Mellish, 2003; Wielemaker, 2003), a general purpose logic programming language, to show the completeness of our patterns. This declarative language has its roots in formal logic. Typically, a Prolog program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over these relations. This allows us to declaratively specify what a coordination scenario should accomplish, so that an execution of the Prolog program (i.e. a query that calculates or derives all solutions) results into all possible coordination scenarios.

Appendix A contains the complete Prolog program. In this subsection we first highlight the main relations specified in the Prolog program (see Subsection 4.5.1). Subsequently, we present the result of the Prolog program execution (see Subsection 4.5.2).

4.5.1 Formalizing a coordination scenario

A coordination scenario must follow several restrictions that can be easily derived from the definitions of a Service Requester, Service Provider and Data Provider. In total, there are four constraints that must be satisfied by a coordination scenario:

C1 The coordination scenario must be a proper interaction scenario:

- **C1.1** The Service Provider can only send data requests or receive data.
- **C1.2** The Data Provider can only receive data requests or send data.

C2 The Data Provider must provide data (i.e. the Data Provider must send data to an entity).

C3 The Service Provider must receive data from another entity.

C4 The resulting data flow must be complete:

C4.1 The Service Requester must forward any data request to the Data Provider.

C4.2 An entity can only send data if this entity is the Data Provider or has received data from another entity.

Based on these constraints we composed a predicate that can be used to query all possible coordination scenarios (see listing 4.1). It clearly contains all constraints discussed above (see comments in listing 4.1).

```
coordination_scenario(CoordinationScenario) :-
  % C1: A coordination scenario needs to be a proper interaction
      scenario
  findall(X, interaction_scenario(X), ListOfInteractionScenarios),
  member(CoordinationScenario, ListOfInteractionScenarios),
  % C2: the Service Provider must receive data from another entity
  member((_, service_provider, data), CoordinationScenario),
  % C3: the Data Provider must send data to an entity
  member((data_provider, _, data), CoordinationScenario),
  % C4: The resulting data flow must be complete
  complete_data_flow(CoordinationScenario),
  % requests or data can only be sent once per participant
  not(multiple_requests_or_data_sent(CoordinationScenario)),
  % requests or data can only be received once per participant
  not(multiple_requests_or_data_received(CoordinationScenario)).
```

Listing 4.1: The coordination_scenario predicate

For a complete Prolog specification of the C1 and C4 constraints, we refer to Appendix A. The C1 constraint is mainly based on the possible_message_-exchange predicate that describes valid interactions with service and Data Providers, as specified in C1.1 and C1.2 (see listing 4.2).

```
% C1.1: The Service Provider can only send data requests or receive
    data
possible_message_exchange(service_provider, Y, data_request) :-
  message_exchange(service_provider, Y, data_request).
possible_message_exchange(X, service_provider, data) :-
  message_exchange(X, service_provider, data).

% C1.2: The Data Provider can only receive data requests or send
    data
possible_message_exchange(data_provider, Y, data) :-
  message_exchange(data_provider, Y, data).
possible_message_exchange(X, data_provider, data_request) :-
  message_exchange(X, data_provider, data_request).
```

Listing 4.2: Restricting possible message exchanges for a Service Provider and a Data Provider

The C4 constraint is specified in the complete_data_flow predicate as the negation of the incomplete_data_flow predicate. The latter predicate simply describes coordination scenarios in which C4.1 or C4.2 are not true (see Listing 4.3).

```
% C4: The data flow is complete
% Specification as the negation of not C4.1 or not C4.2
complete_data_flow(Coordination_messages) :-
  not(incomplete_data_flow(Coordination_messages)).

% C4.1: The Service Requester must forward any data request to the
     Data Provider.
% Specification of a coordination in which C4.1 is not true
incomplete_data_flow(Coordination_messages) :-
  member((service_provider,service_requester,data_request),
       Coordination_messages),
  not(member((service_requester,data_provider,data_request),
       Coordination_messages)).

% C4.2: An entity can only send data if this entity is the Data
     Provider or has received data from another entity.
% Specification of a coordination in which C4.2 is not true
incomplete_data_flow(Coordination_messages) :-
  member((Participant1,Participant2,data),Coordination_messages),
  not(member((_,Participant1,data),Coordination_messages)),
  Participant1\=data_provider.
```

Listing 4.3: The complete_data_flow predicate

4.5.2 Executing the Prolog program

In this subsection we show that the set of solutions generated from constraints C1 to C4 (i.e. all coordination scenarios that met the conditions C1-4 above) exactly matches the coordination scenarios that can be composed by combining our patterns.

When executing the Prolog program, we can query all potential coordination scenarios using the coordination_scenario predicate. Listing 4.4 shows such a query. The Prolog program enumerates all valid coordination scenarios (see variable X). The coordination scenarios found exactly match the coordination scenarios that can be composed by combining our patterns, which shows the completeness of our pattern language (cfr. research question 1). Line 2, 3, 4, 5, 6, 7, 8 and 9 in Listing 4.4 match respectively scenarios 7 (Figure 4.13), 1 (Figure 4.7), 4 (Figure 4.10), 6 (Figure 4.12), 2 (Figure 4.8), 3 (Figure 4.9), 5 (Figure 4.11) and 8 (Figure 4.14).

```
1 ?- coordination_scenario(X).
2 X=[(data_provider,service_provider,data)];
```

```
X=[(data_provider,service_provider,data),(service_provider,
    data_provider,data_request)];
X=[(data_provider,service_provider,data),(service_provider,
    service_requester,data_request),(service_requester,data_provider
    ,data_request)];
X=[(data_provider,service_provider,data),(service_requester,
    data_provider,data_request)];
X=[(data_provider,service_requester,data),(service_provider,
    data_provider,data_request),(service_requester,service_provider,
    data)];
X=[(data_provider,service_requester,data),(service_provider,
    service_requester,data_request),(service_requester,data_provider
    ,data_request),(service_requester,service_provider,data)];
X=[(data_provider,service_requester,data),(service_requester,
    data_provider,data_request),(service_requester,service_provider,
    data)];
X=[(data_provider,service_requester,data),(service_requester,
    service_provider,data)];
false.
```

Listing 4.4: Querying the Prolog program

4.6 Conclusion

In this chapter we have presented a pattern language for managing data dependencies. The pattern language consists of three main patterns (DATA FLOW INITIATION, DIRECT-INDIRECT REQUEST and DIRECT-INDIRECT DATA TRANSMISSION) that can be combined into eight different coordination scenarios to manage a data dependency. Each pattern is about a specific problem in the management of a data dependency and describes several solutions to that problem.

In Section 4.5 we showed that *all* potential coordination scenarios for managing a data dependency can be composed by combining the patterns that were presented in that chapter. Furthermore, based on a detailed evaluation of the solutions presented in a pattern, we were able to derive concrete design guidelines on how to combine the patterns (see Section 4.3). Three decision trees were presented that help to select the most appropriate patterns when designing an optimized coordination scenario.

The whole is more than the sum of its parts

– **Aristotle** (384 BC-322 BC),
Greek philosopher

5 Combining the pattern languages

In the previous two chapters pattern languages for managing sequence and data dependencies were presented. However, in practice, both types of dependencies are present in service compositions. Therefore, in this chapter it is discussed how the two pattern languages presented in this dissertation can be combined to construct complete coordination scenarios.

This chapter starts with a discussion on which types of data dependencies occur in typical service compositions (see Section 5.1). Subsequently, the patterns presented in Chapter 4 are applied to these different types of data dependencies to obtain the coordination scenarios presented in Section 5.2. Based on these coordination scenarios Section 5.3 describes how the pattern languages for managing sequence and data dependencies can be combined.

5.1 Data dependencies service compositions

In our pattern language for managing data dependencies (see Section 4.2 in Chapter 4), we defined three participants in each coordination scenario that manages a data dependency: a Service Requester, a Needy Service Provider and a Data Provider.

In a service composition we can make a distinction between two sorts of *needy services* (i.e. services that require certain data for some reason). First, the execution of a certain business process task can require data. In terms of our pattern language for managing sequence dependencies (see Section 3.2 in Chapter 3) this means that a *Service Provider* supporting that business process task, requires data in order to be able to execute the business process task. Second, sometimes data is required in order to decide if a certain business process task needs to be executed. In terms of our pattern language for managing sequence dependencies (see Section 3.2 in Chapter 3) this means that a *controller* can possibly require data to decide whether or not this controller needs to send a business request to the Service Provider it controls. For the sake of simplicity, we currently make abstraction of whether this controller is an INDEPENDENT CONTROLLER, CONTROLLING SERVICE PROVIDER or a part of a SELF-CONTROLLED SERVICE PROVIDER. Hence, for example, it can occur that a SELF-CONTROLLED SERVICE PROVIDER requires data for both deciding on the execution of business process task and the execution itself. Similarly, a CONTROLLING SERVICE PROVIDER possibly requires data for two reasons.

In a service composition we recognize two sorts of *Data Providers*. First, it is possible that a *Service Provider* provides data as the result of executing a certain business task. We refer to this data as *task data output* or *data output*. Second, it is also possible that a service only provides data that is needed in the service composition. Such a service, which does not support the execution of any business process task, is referred to as *(external) Data Provider*.

Based on the fact that there are two kinds of needy services and two sorts of Data Providers, we identify four types of data dependencies in service compositions (see Table 5.1):

Data Dependency We refer to a data dependency between a Service Provider and a (external) Data Provider as a *(external) data dependency*. This means that the Data Provider needs to be consumed so that the Service Provider can execute the business process task it supports.

Output Input Association This type of data dependency refers to the case in which a Service Provider needs the data output from another business process task in order to execute a task. In other words, an *output input association* associates the output of one business process task to the input of another business process task.

Decision Data Dependency A *decision data dependency* specifies that data available at a Data Provider is required to decide whether a certain business process task needs to be executed.

Decision Data Association A *decision data association* is defined as the case in which task output data is required to decide whether a certain business process task needs to executed.

Data Provider / **Needy Service (reason)**	(external) Data Provider	Service Provider
Service Provider (task execution)	(external) Data Dependency	Output Input Association
Controller (deciding on task execution)	Decision Data Dependency	Decision Data Association

Table 5.1: Four types of data dependencies in service compositions

5.2 Managing data dependencies in service compositions

For all four data dependencies described in the previous Subsection (see Subsection 5.1) our pattern language for managing data dependencies (see Chapter 4) can be applied to find all possible coordination scenarios. In order to find the possible set of scenarios, it is important to identify the role of Service Requester (as defined in our pattern language for managing data dependencies (see pattern language introduction in Subsection 4.2.1 in Chapter 4) in each type of data dependency.

Data Dependency In case of a data dependency, the Service Requester role is played by the controller of the Service Provider. This means that, in theory, all combinations and thus eight scenarios are possible (see Figures 4.7 to 4.14 in Chapter 4). However, for the sake of simplicity, in the rest of this Chapter is assumed that a service composition does not contain ACTIVE (EXTERNAL) DATA PROVIDERS, which means that we only consider six ways of managing a(n) (external) data dependency (see Figures 4.7 to 4.14 in Chapter 4)).

Output Input Association In a coordination scenario that manages an output input association the Service Requester role is played by the controller of the Needy Service Provider. Since the data involved in an output input association is the result of a business process task execution, only coordination scenarios including an ACTIVE DATA PROVIDER are possible. Indeed, the Service Provider, which plays the role of Data Provider in an output input association, actively sends task data output. Hence, only two scenarios are possible: an ACTIVE DATA PROVIDER with DIRECT (see Figure 5.1) or INDIRECT DATA TRANSMISSION (see Figure 5.2).

Decision Data Dependency In a coordination scenario that manages a *decision data dependency* the role of Service Requester and Needy Service Provider are both played by the controller. This is so because the con-

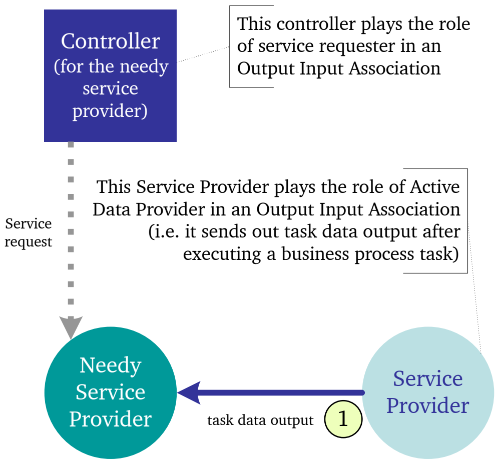

Controller (for the needy service provider)
This controller plays the role of service requester in an Output Input Association
Service request
This Service Provider plays the role of Active Data Provider in an Output Input Association (i.e. it sends out task data output after executing a business process task)
Needy Service Provider
task data output
1
Service Provider

Figure 5.1: Managing an Output Input Association using an ACTIVE DATA PROVIDER (i.e. Service Provider actively sends data) with a DIRECT DATA TRANSMISSION

troller autonomously decides to process business event notifications and then needs additional data to decide whether a business process task needs to be executed. Since we do not consider ACTIVE (EXTERNAL) DATA PROVIDERS, only one coordination scenario is possible (see Figure 5.3).

Decision Data Association Similar to the management of output input associations the data involved in a decision data association is the result of a business process task execution. This means that only coordination scenarios including an ACTIVE DATA PROVIDER are possible. Since the role of Service Requester and Needy Service Provider are both played by the controller, only one coordination scenario remains possible (i.e. an ACTIVE DATA PROVIDER with DIRECT DATA TRANSMISSION) (see Figure 5.4).

5.3 Combining sequence and data dependency management

In the previous subsection we applied our pattern language for managing data dependencies to all four data dependencies that we described in Subsection 5.1. As mentioned in that subsection discussing the four types of

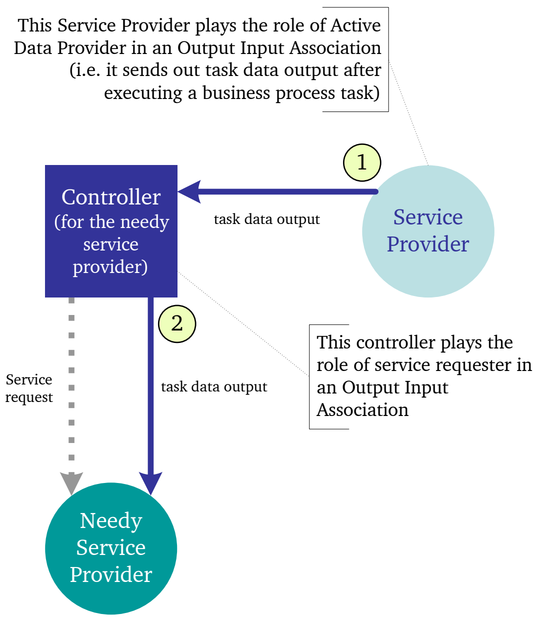

This Service Provider plays the role of Active Data Provider in an Output Input Association (i.e. it sends out task data output after executing a business process task)
1
Controller (for the needy service provider)
task data output
Service Provider
2
Service request
task data output
This controller plays the role of service requester in an Output Input Association
Needy Service Provider

Figure 5.2: Managing an Output Input Association using an ACTIVE DATA PROVIDER (i.e. Service Provider actively sends data) with an INDIRECT DATA TRANSMISSION

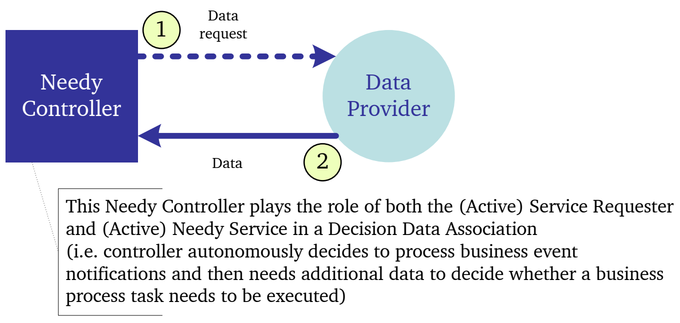

1
Data request
Needy Controller
Data Provider
Data
2
This Needy Controller plays the role of both the (Active) Service Requester and (Active) Needy Service in a Decision Data Association (i.e. controller autonomously decides to process business event notifications and then needs additional data to decide whether a business process task needs to be executed)

Figure 5.3: Managing a Decision Data Dependency

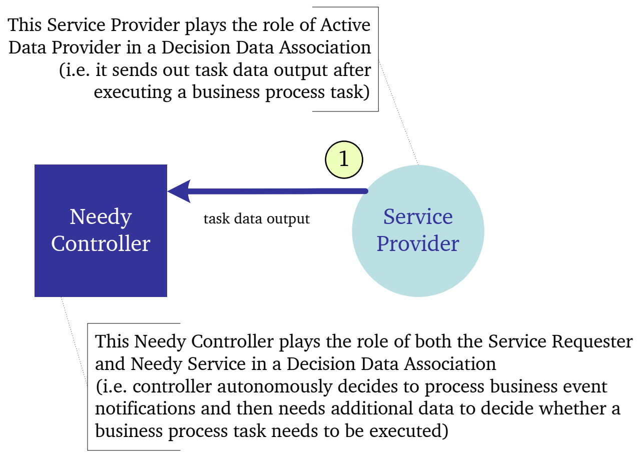

This Service Provider plays the role of Active Data Provider in a Decision Data Association (i.e. it sends out task data output after executing a business process task)
1
Needy Controller
task data output
Service Provider
This Needy Controller plays the role of both the Service Requester and Needy Service in a Decision Data Association (i.e. controller autonomously decides to process business event notifications and then needs additional data to decide whether a business process task needs to be executed)

Figure 5.4: Managing a Decision Data Association

data dependencies (see Subsection 5.1) we made abstraction of whether a controller is an INDEPENDENT CONTROLLER, CONTROLLING SERVICE PROVIDER or a part of a SELF-CONTROLLED SERVICE PROVIDER. This is exactly what the pattern language for managing sequence dependencies is about: distributing the responsibility of controlling the business process tasks.

For all coordination scenarios described in the previous subsection, the pattern language for managing sequence dependencies can be directly applied. However, there are some combinations of patterns for managing sequence and data dependencies that provide potentially additional advantages that were not present separately in each pattern language. In particular, there are potential advantages that arise when the COORDINATOR pattern (see Chapter 3) is combined with coordination scenarios that manage Output Input Associations and/or Decision Data Associations (see Subsection 5.1). We distinguish between three kinds of such pattern combinations.

A first combination that can create additional value is the use of the COORDINATOR pattern combined with two Decision Data associations in which the *same task data output* is associated with (possibly) different controllers. In Figure 5.5 such a combination is shown. The main motivating idea behind this combination is the fact that the controllers are merged into one COORDINATOR so that the task data output only should be sent once. This combination is applicable when the same task data output is required to decide if different business tasks needs to be executed (e.g. in Figure 5.5 both Controller 1 and Controller 2 need the same task data output to decide whether or not the business process tasks that these controllers control need to be executed).

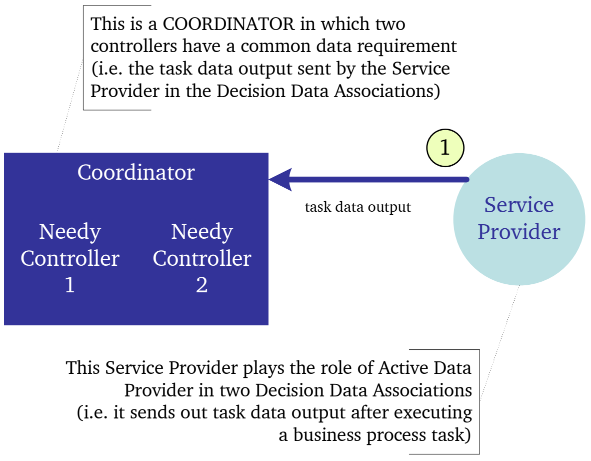

This is a COORDINATOR in which two controllers have a common data requirement (i.e. the task data output sent by the Service Provider in the Decision Data Associations)
Coordinator
Needy Controller 1
Needy Controller 2
1
task data output
Service Provider
This Service Provider plays the role of Active Data Provider in two Decision Data Associations (i.e. it sends out task data output after executing a business process task)

Figure 5.5: The use of the COORDINATOR pattern combined with two Decision Data Associations that are related to the same task data output

The second potentially valuable combination consists of the use of the COORDINATOR pattern combined with two or more Output Input Associations that are related to the *same* task data output and that are all managed using an INDIRECT DATA TRANSMISSION scenario. In Figure 5.6 such a combination is shown. Similar to the previous case the controllers are merged into one COORDINATOR so that the task data output only should be sent once. This combination is applicable when the same task data output is required by two or more different Service Providers in order to execute a business process task (e.g. in Figure 5.6 both Needy Service Provider 1 and Needy Service Provider 2 need the same task data output to execute the business process task that is supported by that Service Provider).

The third potentially valuable combination consists of a COORDINATOR combined with a Decision Data Association and an Output Input Association that are associated to the same task output data and where the Output Input Association is managed using an INDIRECT DATA TRANSMISSION scenario. Figure 5.7 shows such a combination. Similar to the two previous cases the controllers are merged into one COORDINATOR so that the task data output only should be sent once. This combination is applicable when the same task data output is required by both a Service Provider in order to execute a business process task and a controller to decide whether a business process task needs to be executed (e.g. in Figure 5.7 both controllers need the same task data output).

Note that the advantages described above were situated in coordination

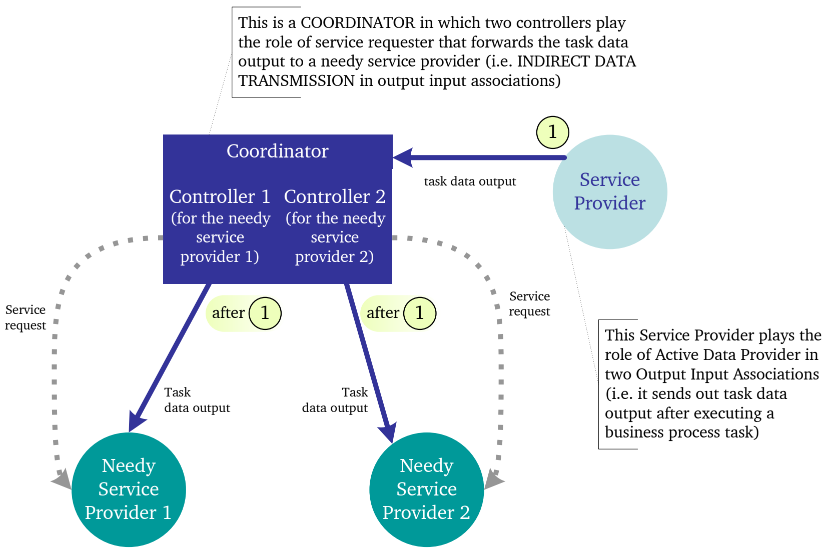

This is a COORDINATOR in which two controllers play the role of service requester that forwards the task data output to a needy service provider (i.e. INDIRECT DATA TRANSMISSION in output input associations)
Coordinator
Controller 1 (for the needy service provider 1)
Controller 2 (for the needy service provider 2)
1
task data output
Service Provider
Service request
after 1
after 1
Service request
This Service Provider plays the role of Active Data Provider in two Output Input Associations (i.e. it sends out task data output after executing a business process task)
Task data output
Task data output
Needy Service Provider 1
Needy Service Provider 2

Figure 5.6: The use of the COORDINATOR pattern combined with two Output Input Associations that are related to the same task data output

scenarios in which Service Providers played the role of Data Providers in Output Input or Decision Data Associations. We explicitly choose not to describe such advantages when an *external* Data Provider is involved (i.e. in case of an external data dependency or decision data dependency). This is so because we assumed that data provided by a Data Provider is not necessarily always the same. For example, when two needy controllers that together form a COORDINATOR, both request the same data from a common Data Provider, it is not necessarily possible to limit the data requests sent by the COORDINATOR. This is so because possibly the controllers request the data at a different moment in time and as a consequence the Data Provider sends not the same data to the two controllers. This is in contrast with data provided by a Service Provider (i.e. task data output), which is per definition only provided once and cannot change over time.

5.4 Conclusion

In this chapter we have described how the two pattern languages presented in this dissertation can be combined to construct complete coordination scenarios.

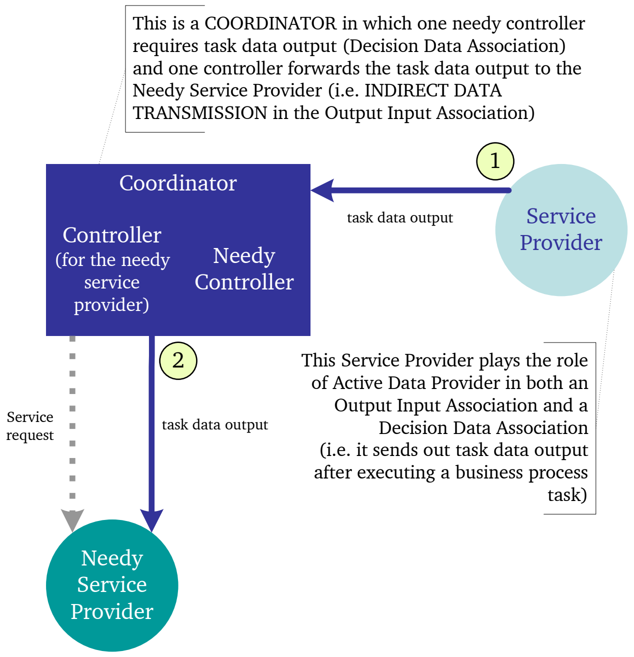

This is a COORDINATOR in which one needy controller requires task data output (Decision Data Association) and one controller forwards the task data output to the Needy Service Provider (i.e. INDIRECT DATA TRANSMISSION in the Output Input Association)
Coordinator
Controller (for the needy service provider)
Needy Controller
1
task data output
Service Provider
2
Service request
task data output
This Service Provider plays the role of Active Data Provider in both an Output Input Association and a Decision Data Association (i.e. it sends out task data output after executing a business process task)
Needy Service Provider

Figure 5.7: The use of the COORDINATOR pattern combined with an Output Input Association and a Decision Data Association that are related to the same task data output

In Section 5.1 we have described four types of data dependencies that can occur in typical service compositions. Subsequently, we have applied the pattern language for managing data dependencies to each of these types (see Section 5.1). Subsequently, the patterns presented in Chapter 4 are applied to these different types of data dependencies to obtain the coordination scenarios presented in Section 5.2. Based on these coordination scenarios Section 5.3 describes how the pattern languages for managing sequence and data dependencies can be combined. In particular, it is shown how the use of patterns for managing sequence dependencies (e.g. COORDINATOR) can be beneficial in the context of data dependency management.

Knowing is not enough; we must apply.
Willing is not enough; we must do.
– **Johann Wolfgang von Goethe** (1749–1832),
German writer

6 Tool support for pattern-based coordination

In this chapter we describe how we have developed a tool that demonstrates that it is possible to semi-automate the construction of a coordination scenario. In the tool developers pick out specific patterns for sequence and data dependencies management and then the tool automatically generates a complete coordination scenario (in the form of BPEL (OASIS, 2007) processes) from a business process specification (e.g. a BPMN model (OMG, 2010a)).

The chapter starts with a high-level overview of our pattern-based approach for service composition and coordination (see Section 6.1). Subsequently, in Section 6.2 it is discussed which input models are required to automatically generate coordination scenarios. Section 6.3 presents the complete approach and describes the tool that we have developed to automate the construction of BPEL-based coordination scenarios. In Section 6.4 the tool is applied to an extended version of the travel agency process that was introduced in Chapter 3. Finally, the chapter concludes in Section 6.5.

6.1 From business process modeling to service composition

In this section we give a high-level overview on our pattern-based approach for service composition and coordination. The section starts with a brief discussion on existing BPMN-to-BPEL transformations.

6.1.1 Existing BPMN-to-BPEL transformations

In the literature one can find several papers discussing a translation of BPMN (OMG, 2010a) processes to BPEL (OASIS, 2007) processes (Recker & Mendling, 2006; White, 2005; Fjellheim, Milliner, Dumas, & Vayssière, 2007; Ouyang, Dumas, Hofstede, & Van der Aalst, 2006). Several techniques described in these studies are also implemented in BPM suites tools from major BPM players such as IBM, Oracle, etc. who use these translation approaches for a better interoperability between their modeling and development tool.

However, these techniques do not allow to automatically construct coordination scenarios. Either it is up to the business analyst to include coordination aspects in their business process model or it is up to the developer to extend the transformed process model with a coordination model. For example, a business process task modeled in BPMN typically is only translated in a BPEL invoke activity (e.g. White, 2005). As such it often is implicitly assumed that the business process task is also completed when the message is sent to the service supporting that task (i.e. the invoke activity is done). However, for example, if the completion of a business process task needs to be confirmed by the reception of a business event notification message, either business analysts or developers explicitly need to specify 'receive' tasks or activities.

Some researchers acknowledge this shortcoming and therefore propose a translation of BPMN process models to event-based coordination models. Such translations identify all business events (e.g. the completion of a previous business process task) that need to be occurred and generate appropriate 'receive' activities for receiving the necessary business event notifications (e.g. Fjellheim et al., 2007). However, one can still see two weak points in these studies. First, these approaches fail to manage data dependencies in a systematic or automatic way. Receiving task data output and sending task data input still need to be explicitly modeled by the analyst or added by the developer. If business analysts need to model data-related tasks, business process models become too complex. If developers need to specify such activities, it becomes a challenge to adapt the implemented process to every change in the business process. Indeed, developers not only need to change the business process itself, but also need to modify coordination logic and potentially need to repeatedly implement the same implementation patterns (e.g. a specific coordination pattern for managing a data dependency). Second, as with most BPMN-to-BPEL transformations, these approaches target a centralized coordination. The transformation typically results into one BPEL process that needs to be deployed to a central BPEL engine. If another composition style is desired (e.g. decentralized coordination), analysts are supposed to model several BPMN processes.

6.1.2 Pattern-based service composition and coordination

Our pattern-based service composition and coordination starts from the idea that the services in a service composition support business process tasks. This means that services need to be able to receive business requests and send out business event notifications about the execution of the business process task it supports (e.g. the completion of the task). Furthermore, a service supporting a business process also needs the ability to receive all data that is necessary to execute a certain business process. Moreover, if the business process task results into task data output, then the service is supposed to send out this data. All these expectations concerning a service supporting a business process task, can be described in an orchestration (e.g. a BPEL process). A skeleton for this orchestration can be generated automatically based on the business process. Hence, developers only need to add logic so that, for example, business requests are translated in the appropriate actions to start a business process execution. Once, the orchestration for a service is complete, using our pattern-based service composition and coordination approach all coordination scenarios can be generated.

In a concrete coordination scenario business requests need to be sent to the service supporting the business process task. This responsibility is delegated to the controllers. Such a controller can be defined using an orchestration too (e.g. a BPEL process). COORDINATORS are also defined in one orchestration, so that the reception of business event notifications and data can be shared among all controllers that together form the coordinator. If a particular service that supports the execution of a business process task is a SELF-CONTROLLED SERVICE PROVIDER, then the controller BPEL process can be simply deployed to the same BPEL engine that runs the service provider's BPEL process for receiving business requests etc.

The main advantage of this approach is that all orchestrations forming the global interaction and coordination scenario are automatically generated from a business process specification after selecting specific coordination patterns. Therefore, it is relatively easy to apply other patterns for managing sequence and data dependencies so that a different coordination scenario is composed.

In the next sections we show how we automatically generate BPEL processes, including the corresponding WSDL (W3C, 2001) interfaces for these processes, that together form a specific coordination scenario.

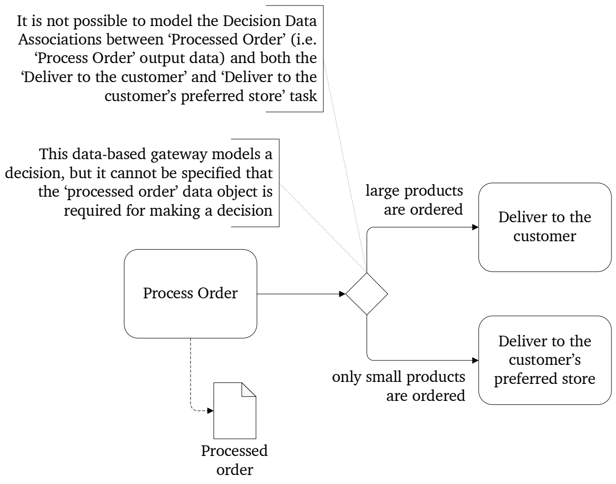

It is not possible to model the Decision Data Associations between 'Processed Order' (i.e. 'Process Order' output data) and both the 'Deliver to the customer' and 'Deliver to the customer's preferred store' task
This data-based gateway models a decision, but it cannot be specified that the 'processed order' data object is required for making a decision
large products are ordered
Deliver to the customer
Process Order
Deliver to the customer's preferred store
only small products are ordered
Processed order

Figure 6.1: Modeling Decision Data Associations in BPMN (OMG, 2010a)

6.2 Input for pattern-based service composition and coordination

6.2.1 Representing sequence and data dependencies in BPMN

Since BPMN (OMG, 2010a) is a standard language for modeling business processes it is perfectly suited for representing sequence dependencies. However, as already explained briefly in Chapter 2 (see Subsection 2.5.1) modeling all kind of data dependencies is not possible.

Modeling Decision Data Dependencies or Decision Data Associations is the most problematic in BPMN. As explained in detail in Chapter 2 (see Subsection 2.5.1) BPMN does not provide language constructs for modeling associations between data objects and sequence flow conditions. This implies that it is not possible to model that certain data is required for deciding whether a certain task should be executed or not, regardless of whether this data is available at an external data provider or is the result of the execution of another business process task. For example, Figure 6.1 shows a business process fragment in which two Decision Data Associations (i.e. the 'processed order' is required to decide on the execution of the two delivery tasks) could not be modeled using BPMN constructs.

External Data Dependencies and Output Input Associations can be modeled using *data objects* that are associated to BPMN tasks as *data input* or *data* *output*. An Output Input Association can be modeled by linking an activity's *data output* to another activity's *data input* to indicate that the output data of a certain task is the input for another task (e.g. see Figures 2.5(a) and 2.5(b) in Chapter 2). External Data Dependencies can also be described in BPMN using a special *input* marker (i.e. a non-colored arrow) in the data object that represents the data input (e.g. see Figure 2.6 in Chapter 2). Nonetheless, it should be noted that additional specification is necessary to model the data provider's interface when one wants to automatically generate a complete coordination scenario.

6.2.2 An additional data dependencies model

As explained in the previous subsection (see Subsection 6.2.1), Decision Data Dependencies and Decision Data Associations cannot be represented in a BPMN model. Furthermore, External Data Dependencies can only partially be represented in the BPMN model, because BPMN does not provide the language constructs to specify the data provider in an External Data Dependency.

Therefore, we have defined an XML Schema (W3C, 2004) for describing a data dependencies model that consists of External Data Dependencies (including data providers), Decision Data Dependencies and Decision Data Associations.

The complex types for an External Data Dependency and a Decision Data Dependency both are derived from a data dependency complex type that is defined as shown in Listing 6.1. The required attributes *dataProviderId* and *bpmnTaskId* refer to the IDs of the data provider (as defined in the data dependencies model) and the business process task (as defined in the BPMN model). The optional attribute *dataInputAssociationId* refers to the ID of a *dataInputAssociation* element as defined in a BPMN model. Although we assume that a data provider defines the data structure of the data object that it provides, we optionally relate a data dependency to a data object defined in a BPMN model. Currently, we do not check any consistency between both data structures. However, in case a coordination scenario that includes an INDIRECT DATA TRANSMISSION is used we assume that the data object structure as defined in the BPMN model is the data object structure that the controller or service provider needs. In that case, the data object structure as defined in the data provider is only used in the communication with the data provider.

```
<complexType name="tDataDependency">
  <attribute name="id" type="ID" use="required"/>
  <attribute name="name" type="string" use="required"/>
  <attribute name="dataProviderId" type="NCName" use="required"/>
  <attribute name="bpmnTaskId" type="NCName" use="required"/>
  <attribute name="dataInputAssociationId" type="NCName" use="
      optional"/>
</complexType>
```

Listing 6.1: The complex type for data dependencies

The complex type for a data provider is shown in Listing 6.2. It has attributes for specifying the location of the WSDL file, request and receive operations and BPEL partnerLink information. Using this information it is defined how a service can interact with that data provider.

```
<complexType name="tDataProvider">
  <attribute name="id" type="ID" use="required"/>
  <attribute name="name" type="string" use="required"/>
  <attribute name="wsdlLocation" type="string" use="required"/>
  <attribute name="requestOperationName" type="string" use="required
      "/>
  <attribute name="requestRoleName" type="string" use="required"/>
  <attribute name="receiveOperationName" type="string" use="required
      "/>
  <attribute name="partnerLinkTypeName" type="string" use="required"
      />
  <attribute name="receiveRoleName" type="string" use="required"/>
</complexType>
```

Listing 6.2: The complex type for a data provider

The complex type for a Decision Data Association is shown in Listing 6.3. It has two important attributes. The *sourceRef* attribute refers to the ID of the data object (as defined in the BPMN model) that is required to make the decision. The *targetRef* attribute refers to the ID of the business process task (as defined in the BPMN model) involved in a Decision Data Association.

```
<complexType name="tDecisionDataAssociation">
  <attribute name="id" type="ID" use="required"/>
  <attribute name="sourceRef" type="NCName" use="required"/>
  <attribute name="targetRef" type="NCName" use="required"/>
</complexType>
```

Listing 6.3: The complex type for a Decision Data Association

The complete XML Schema for our data dependencies model can be found in Appendix B (see Section B.1).

6.2.3 Representing a coordination model

In order to generate a coordination scenario from a business process specification and a data dependencies model, the user needs to pick patterns for managing the sequence and data dependencies. These choices can be specified in a so called *coordination model*, for which we defined an XML Schema (W3C, 2004).

A coordination model consists of two main elements: a *sequenceDependenciesManagement* and *a dataDependenciesManagement* element.

The complex type for a *sequenceDependenciesManagement* element has one attribute that specifies the location of the BPMN model. In addition it can have several *coordinator* elements as child elements (see Listing 6.4).

```
<complexType name="tSequenceDependenciesManagement">
  <sequence minOccurs="1" maxOccurs="1">
    <element name="coordinator" type="cm:tCoordinator" minOccurs="1"
          maxOccurs="unbounded"/>
  </sequence>
  <attribute name="bpmnModel" type="string" use="required"></
      attribute>
</complexType>
```

Listing 6.4: The complex type for sequence dependencies management

In line with the COORDINATOR pattern, a *coordinator* is defined by a set of BPMN tasks (via the IDs as defined in the BPMN model) (i.e. it controls several business process tasks) and a name (see Listing 6.5).

```
<complexType name="tCoordinator">
  <sequence minOccurs="1" maxOccurs="1">
    <element name="bpmnTaskId" type="NCName" minOccurs="1"maxOccurs=
         "unbounded">
    </element>
  </sequence>
  <attribute name="name" type="string" use="required"></attribute>
</complexType>
```

Listing 6.5: The complex type for a COORDINATOR

The second element in a coordination model is the *dataDependenciesManagement* element, which has one attribute to specify the data dependencies model and several *dataDepedencyManagement* and *dataOutputIsDataInputManagement* elements (see Listing 6.6).

```
<complexType name="tDataDependenciesManagement">
  <sequence>
    <element name="dataDependencyManagement" type="
         cm:tDataDependencyManagement" minOccurs="0" maxOccurs="
         unbounded"/>
    <element name="dataOutputIsDataInputManagement" type="
         cm:tDataOutputIsInputManagement" minOccurs="0" maxOccurs="
         unbounded"/>
  </sequence>
  <attribute name="dataDependenciesModel" type="string" use="
      required"/>
</complexType>
```

Listing 6.6: The complex type for data dependencies management

A *dataDependencyManagement* element specifies how an External Data Dependency is managed. In particular it defines which coordination scenario (of the six that are represented in Figures 4.7 to 4.12 in Chapter 4) is used via the *pattern* attribute (see Listing 6.7). This attribute is of the type *dataDependencyManagementPattern* which is nothing else than an enumeration that let users pick specific patterns for managing the (external) Data Dependency (see Listing 6.8).

```
<complexType name="tDataDependencyManagement">
  <attribute name="id" type="ID"/>
  <attribute name="dataDependencyId" type="NCName"/>
  <attribute name="pattern" type="cm:dataDependencyManagementPattern
      "/>
</complexType>
```

Listing 6.7: The complex type for Data Dependency management

```
<simpleType name="dataDependencyManagementPattern">
  <restriction base="string">
    <enumeration value="active_service_provider_with_
direct_request_and_direct_data_transmission" />
    <enumeration value="active_service_provider_with_
direct_request_and_indirect_data_transmission" />
    <enumeration value="active_service_provider_with_
indirect_request_and_direct_data_transmission" />
    <enumeration value="active_service_provider_with_
indirect_request_and_indirect_data_transmission" />
    <enumeration value="active_service_requester_with_
direct_data_transmission" />
    <enumeration value="active_service_requester_with_
indirect_data_transmission" />
  </restriction>
</simpleType>
```

Listing 6.8: The simple type that defines the different coordination scenarios for managing an External Data Dependency

A *dataOutputIsInputManagement* element specifies how a Data Input Output Association is managed. In such an element the Data Input Output Association is specified by referencing the corresponding BPMN outputAssociation and inputAssocition elements (as defined in the BPMN model) (see Listing 6.9). The *pattern* attribute, which is of the *dataOutputIsInputManagementPattern* type, specifies how the Data Input Output Association is managed. Similar to the *dataDependencyManagementPattern* simple type this type is defined by an enumeration that let users pick specific patterns for managing the Output Input Association (see Listing 6.10).

```
<complexType name="tDataOutputIsInputManagement">
  <attribute name="id" type="ID" use="required"/>
  <attribute name="outputAssociationId" type="NCName" use="required"
      />
  <attribute name="inputAssociationId" type="NCName" use="required"/
      >
  <attribute name="pattern" type="
      cm:dataOutputIsInputManagementPattern" use="required"/>
</complexType>
```

Listing 6.9: The complex type for Data Output Input Association management

```
<simpleType name="dataOutputIsInputManagementPattern">
  <restriction base="string">
    <enumeration value="direct_data_transmission" />
    <enumeration value="indirect_data_transmission" />
  </restriction>
</simpleType>
```

Listing 6.10: The simple type that defines the different coordination scenarios for managing a Data Output Input Association

Note that the coordination model does not contain elements for specifying the patterns required for the management of Decision Data Dependencies or Decision Data Associations. This makes sense because as we have described in Subsection 5.2 these type of data dependencies can only be managed in one way.

The complete XML Schema for our coordination model can be found in Appendix B (see Section B.2).

6.3 Pattern-based service composition and coordination

In this section we describe the inner working of the tool we developed to automatically generate coordination scenarios based on a business process specification and a set of patterns (see Figure 6.2).

In Figure 6.3 a UML (OMG, 2010b) class diagram is represented, showing the most important Java classes that form the tool's core. As one can see in this class diagram, there are three sorts of classes: BPEL-related classes (i.e. BPEL4EventDispatcher, BPEL4TaskExecutionService, BPEL4Coordinator), WSDL-related classes (i.e. WSDL4EventDispatcher, WSDL4TaskExecutionService

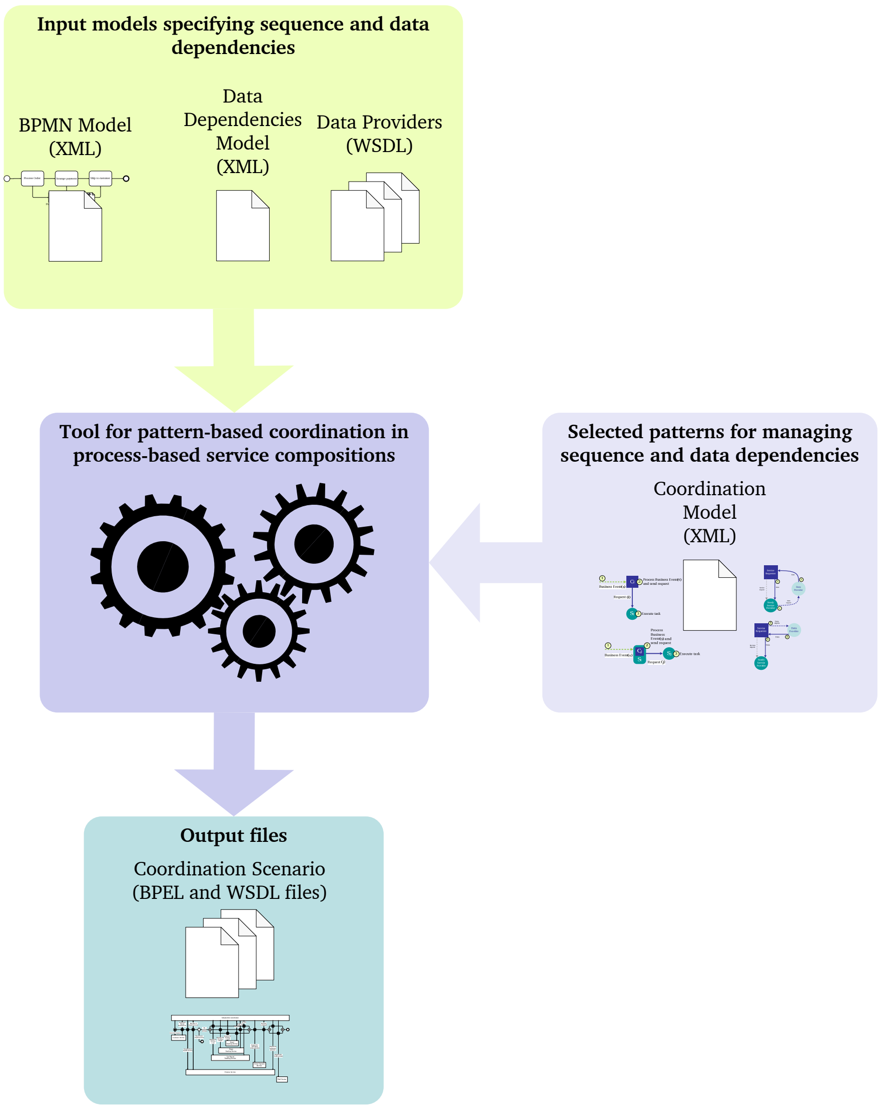

Input models specifying sequence and data dependencies
BPMN Model (XML)
Data Dependencies Model (XML)
Data Providers (WSDL)
Tool for pattern-based coordination in process-based service compositions
Selected patterns for managing sequence and data dependencies
Coordination Model (XML)
Output files
Coordination Scenario (BPEL and WSDL files)

Figure 6.2: Input and output for pattern-based coordination

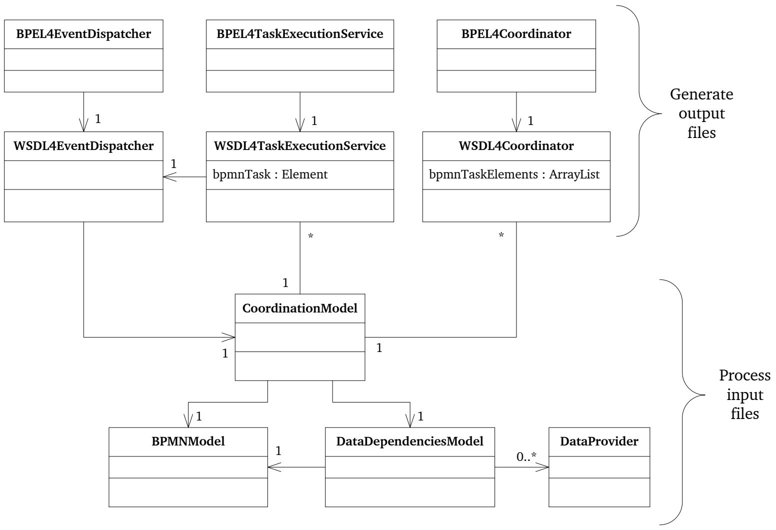


Figure 6.3: A UML (OMG, 2010b) class diagram showing the core of our tool

and WSDL4Coordinator), and model-related classes (i.e. CoordinationModel, BPMNModel, DataDependenciesModel and DataProvider). The model-related classes, discussed in Subsection 6.3.1, are used for processing the input models, while the BPEL-related and WSDL-related classes, described in Subsection 6.3.2, are used for generating the output files.

6.3.1 Processing the input models

In this subsection we show how the input models are processed. Before discussing the processing of each input model, this subsection starts with a general discussion of how the XML-based input models are processed in the tool.

XML processing using XOM and XQuery

Since both the input for our tool (i.e. BPMN Model, data dependencies model, data providers and coordination model) and the generated output (i.e. BPEL process and WSDL descriptions) are XML files, an API for XML processing is intensively used in the tool's implementation. We made use of the XOM parser[1] for parsing and constructing XML files. The main reason why we used this API is that there exists another toolkit called Nux[2] that provides XQuery (W3C, 2007b) support for XOM. XQuery is a query language that allows to easily query XML documents in a declarative way. For example, finding the BPMN *sequenceFlow* elements that target a certain BPMN *task* element can be relatively easy done using the combination of Java and XQuery code shown in Listing 6.11. Being able to query BPMN models was extremely valuable in order to find all relevant business events that need to occur before a business process task needs to be started.

```
String query = "//ns:sequenceFlow[@targetRef='" + bpmnTaskElement.
    getAttributeValue("id") + "']/@sourceRef";
Nodes results = XQueryUtil.xquery(getParsedBPMNFile(),
    NAMESPACE_DECLARATION + query);
```

Listing 6.11: The use of XQuery in Java to find flow elements that target a certain BPMN *task* element

Processing the BPMN model

For the processing of a BPMN model we have developed one Java class (BPMNModel in Figure 6.3) that reads and searches through a BPMN model using the XOM parser and XQuery. The most important and frequently used methods in this class are methods for retrieving BPMN tasks, calculating the business events that need to occur before a task needs to be executed, getting data (output and input) objects and corresponding types, and collecting Data Output Input Associations.

Processing the data dependencies model

Processing a data dependencies model (see Subsection 6.2.2 for a description of this model) occurs in the DataDependenciesModel class (see Figure 6.3). Together with the DataProvider class these classes provide several methods to retrieve External Data Dependencies, Data Providers, Decision Data Dependencies and Decision Data Associations. The DataDependenciesModel also holds a reference to a BPMN Model so that, for example, data objects as defined in a BPMN model and that are referenced in data dependency definitions, can be rapidly retrieved.

[1]http://www.xom.nu

[2]http://acs.lbl.gov/software/nux/

Processing the coordination model

The CoordinationModel class (see Figure 6.3) processes a coordination model (see Subsection 6.2.3 for a description of this model) and provides methods to get information about how sequence and data dependencies are supposed to be managed in a generated coordination scenario. The Coordination-Model also holds references to both a BPMN model and a data dependencies model. Furthermore, it provides methods to find which COORDINATOR (see WSDL4Coordinator class described in Subsection 6.3.2) controls a certain business process task. Finally, there are also methods included to obtain a service provider's WSDL description (see the WSDL4TaskExecutionService class described in Subsection 6.3.2).

6.3.2 Generating the output files

As described in Subsection 6.1.2 our pattern-based approach for service composition and coordination consists of the generation of BPEL processes for both the service providers that support business process tasks and the COORDINATORS that hold business process knowledge to decide when a certain business process task needs to be executed. The generation of both types of BPEL processes and the corresponding WSDL descriptions for these processes, are presented in the rest of this subsection.

Generating a service provider's WSDL description

A service provider's WSDL description is constructed by generating the following WSDL elements:

Import elements If there exists an External Data Dependency in the data dependencies model that is related to this service provider and the coordination model specifies that this External Data Dependency is managed using a DIRECT DATA TRANSMISSION, then an *import* element is generated to import the data provider's WSDL description, which is required to successfully define the service for receiving the data (see generation of *service* elements).

Types element First of all the WSDL *types* element defines elements for describing business requests. The main child element in such a business request is a *process_id* element that is used to correlate all interactions to the right process instance. In case of an ACTIVE SERVICE PROVIDER with an INDIRECT REQUEST the business request also includes an element specifying the URI[3] of the controller's port to which the ACTIVE SERVICE PROVIDER can send its INDIRECT REQUESTS. In case of an ACTIVE SERVICE REQUESTER with a DIRECT DATA TRANSMISSION the business request also includes a *request_id* element that is used to correlate messages received from a data provider. The *types* element also defines elements for task data output that the service provider needs to receive in order to execute the business process (i.e. in case of Data Output Input Associations). In case of an External Data Dependency that is managed using an INDIRECT DATA TRANSMISSION the *types* also defines elements for receiving the data related to this data dependency. In such a situation we assume that the data received is a data input object defined in the BPMN model. In case of a DIRECT DATA TRANSMISSION no data types need to be defined, because these are already described in the data provider's WSDL description. Finally, when the service provider sends INDIRECT REQUESTS to its controller, the service provider's WSDL description also describes the interface that it expects to find in the controller for sending these INDIRECT REQUESTS and hence also the types of messages that are used in that interface.

Message elements The generation of these elements is similar to the generation of the *types* element that describes several message types.

Port type elements First of all the service provider's WSDL description needs a port type for receiving business requests. In case of an External Data Dependency that is managed using an INDIRECT DATA TRANSMISSION the service provider's WSDL description should also define a port type for receiving the data specified in the data dependency. As explained in the generation of the *types* element, if the service provider sends out INDIRECT REQUESTS to its controller, then the service provider's WSDL description also describes the interface that it expects to find in the controller for sending these INDIRECT REQUESTS. This implies that the port type for sending these INDIRECT REQUESTS is also defined in the WSDL description. Finally, if the service provider needs to receive task data ouput from other service providers, an appropriate port type for receiving such data is also included. Note that only one port type, having multiple operations, is defined when the service provider is involved in multiple Data Output Input Associations.

Binding elements The generation of these elements is similar to the generation of the *portType* elements. For each port type generated an appropriate document-based SOAP (W3C, 2000) binding is generated.

[3]Uniform Resource Identifier

Service elements The generation of these elements is also largely similar to the generation of the port type and binding elements. However, in case of an External Data Dependency managed using a DIRECT DATA TRANSMISSION an additional *service* element is generated. This *service* element defines a port associated to a binding that is defined in the WSDL description of the data provider involved in the data dependency.

Partner Link Type elements The service provider's WSDL description defines partner link types for three purposes. First, a partner link type is generated for the receival of business requests. Second, partner link types for receiving task output data are generated if the service provider is involved in a Data Output Input Association. Finally, if the service provider sends out INDIRECT REQUESTS or receives data via INDIRECT DATA TRANSMISSIONS a partner link type is generated to describe the necessary interaction between the service provider and its controller.

Property and property alias elements The WSDL description of each service provider defines a property that is used to correlate messages to the right process instance. In case of External Data Dependencies that are managed using DIRECT DATA TRANSMISSION an additional property is defined to correlate data that is received to the right data dependency. For each message that needs to be correlated a corresponding *property alias* element is generated.

Generating a service provider's BPEL process

A service provider's BPEL process is constructed by generating the following BPEL elements:

Import elements First of all a service provider's BPEL process needs to import its own WSDL description (see discussion above). Second, the WSDL description for the event dispatcher needs to be imported so that the BPEL process can send out business event notifications. Third, if the service provider produces task data output, it needs to import the WSDL descriptions of the COORDINATORS and service providers that need this task data output. These needy COORDINATORS are involved in an Output Input Association managed using an INDIRECT DATA TRANSMISSION or a Decision Data Association, while the needy service providers are involved in an Output Input Association using a DIRECT DATA TRANSMISSION. Finally, if an External Data Dependency is managed using a DIRECT REQUEST or a DIRECT DATA TRANSMISSION then an *import* element is generated to import the data provider's WSDL description.

Partner Link elements If the service provider is involved in one or more Output Input Association and needs task data output from one or more service providers, then a *partnerLink* element needs to be generated.

Variable elements The BPEL process of every service provider should at least define variables for storing the business request and preparing the business event notification message. If the service provider produces task data output there are also *variable* elements generated for the data messages sent to needy COORDINATORS and needy SERVICE PROVIDERS. Finally, if the service provider is involved in an External Data Dependency, appropriate *variable* elements need to be generated for requesting and/or receiving data.

Correlation set elements The generation of these elements is similar to the generation of *property* and *propertyAlias* elements in the service provider's WSDL description.

A sequence element The concrete process that is described in the BPEL process consists of one *sequence* element. In this *sequence* element three sequential BPEL activities are defined. The first BPEL activity in this *sequence* element is a BPEL *flow* element which we refer to as the *pre-execution flow element*. In this *flow* element for each task data output that needs to be received a *receive* activity is defined. Furthermore, this *flow* element includes another *sequence* element which we refer to as the *'receive business request and manage external data dependencies' sequence element*. As the name suggests, this *sequence* element consists of a *receive* activity for receiving business requests followed by a *flow* element in which for each External Data Dependency the required request and receive actions are defined. The *pre-execution flow element* is followed by the logic to execute the business process task (e.g. invoking another Web service) and a so called *post-execution flow element*. In this last *flow* element the business event notification is sent to the event dispatcher, and any produced task data output is sent to the needy COORDINATORS and needy SERVICE PROVIDERS.

Generating a COORDINATOR's WSDL description

A COORDINATOR's WSDL description is constructed by generating the following WSDL elements:

Import elements A COORDINATOR's WSDL description contains *import* elements for every Decision Data Dependency the COORDINATOR is involved in (i.e. importing the data provider's WSDL description). Furthermore, in case of an External Data Dependency between a business process task controlled by the COORDINATOR and a data provider, that is managed using an INDIRECT DATA TRANSMISSION, an additional *import* element is generated for importing the data provider's WSDL description. In such coordination scenarios the data provider's WSDL description is used in the definition of a *service* element for receiving the data. If the External Data Dependency is managed using an ACTIVE SERVICE PROVIDER with an INDIRECT REQUEST the service provider's WSDL description needs to be imported, because that WSDL description describes the interface that the COORDINATOR should provide to receive INDIRECT REQUESTS.

Types element The *types* element defines types for all business event notifications and task data output that are to be received by the COORDINATOR (either because of an INDIRECT DATA TRANSMISSION in an Output Input Association or because of a Decision Data Dependency).

Message elements The generation of these elements is similar to the *types* element generation.

Port type elements In the COORDINATOR's WSDL description one port type is generated for receiving business event notifications. If the COORDINATOR needs to receive task output data an additional port type is generated (either because of an INDIRECT DATA TRANSMISSION in an Output Input Association or because of a Decision Data Dependency).

Binding elements The generation of these elements is similar to the *port type* elements generation.

Service elements In the WSDL description of every service provider a *service* element for receiving business event notifications is generated. Furthermore, if necessary, a *service* element is generated for receiving task output data (i.e. similar to the generation of the port type element for receiving task output data). For each Decision Data Dependency in which the COORDINATOR is involved, a *service* element is generated for receiving the data. Similarly, a *service* element is generated when an External Data Dependency is managed using an INDIRECT DATA TRANSMISSION. If the External Data Dependency is managed using an ACTIVE SERVICE PROVIDER with an INDIRECT REQUEST the COORDINATOR's WSDL description should also define a service for receiving these INDIRECT REQUESTS.

Partner Link Type elements The generation of these elements is similar to the generation of the port type elements.

Property and property alias elements The WSDL description of each COORDINATOR defines a property that is used to correlate messages to the right process instance. In case of External Data Dependencies that are managed using an INDIRECT DATA TRANSMISSION an additional property is defined to correlate data that is received to the right data dependency. For each message that needs to be correlated a corresponding *property alias* element is generated.

Generating a COORDINATOR's BPEL process

A COORDINATOR's BPEL process is constructed by generating the following BPEL elements:

Import elements First of all a COORDINATOR's BPEL process needs to import its own WSDL description (see discussion above). Second, a COORDINATOR's BPEL process needs to import the WSDL description of every service provider the COORDINATOR controls. Furthermore, the WSDL description of data providers from Decision Data Dependencies and data providers from External Data Dependencies, that are managed using ACTIVE SERVICE PROVIDERS with INDIRECT REQUESTS or ACTIVE SERVICE REQUESTERS or INDIRECT DATA TRANSMISSIONS, need to be imported.

Partner link elements A COORDINATOR's BPEL process typically defines multiple partner links. Every COORDINATOR needs at least partner links for receiving business event notifications and sending business requests to the service providers it controls. Furthermore, for every imported data provider's WSDL description, there is also a partner link related to that data provider (see generation of import elements). In case the COORDINATOR also needs to receive task data output from other service providers, the BPEL process also contains an appropriate partner link for doing so. Similarly, the BPEL process needs to define partner links for forwarding task data output to a needy service provider. Finally, if an External Data Dependency is managed using an INDIRECT DATA TRANSMISSION or an ACTIVE SERVICE PROVIDER with an INDIRECT REQUEST, an additional partner link is generated to enable the data-related interactions between the COORDINATOR and the service provider.

Variable elements First of all every COORDINATOR's BPEL process should have variables for storing business event notifications and business requests. If the COORDINATOR is involved in a Decision Data Dependency also variables for requesting and receiving the data required to make a decision are generated. In case the COORDINATOR also needs to receive task data output from other service providers, the BPEL process also contains variables for storing such data. Similarly, the BPEL process needs to define variables for forwarding such task data output to a needy service provider. Finally, if the service provider is involved in an External Data Dependency, appropriate *variable* elements need to be generated for receiving data (i.e. in case of INDIRECT DATA TRANSMISSION), receiving INDIRECT DATA REQUESTS, and sending data requests (i.e. in case of an ACTIVE SERVICE REQUESTER or an ACTIVE SERVICE PROVIDER with an INDIRECT REQUEST).

Correlation set elements The generation of these elements is similar to the generation of *property* and *propertyAlias* elements in the COORDINATOR's WSDL description.

A flow element The concrete process that is described in the BPEL process consists of one *flow* element. This *flow* element contains BPEL receive activities for receiving both business event notifications and task data output from other service providers. Using BPEL *links* these receive activities are linked to specific controllers for each business process task the COORDINATOR controls. These controllers are defined using *sequence* elements in which first potential Decision Data Dependencies are managed (in a *flow* element) and then a business request is sent to the service provider. Subsequently, potential External Data Dependencies are managed (in a *flow* element).

Generating a BPEL-based event dispatcher

For the sake of simplicity we did not implement an advanced publish-subscribe system for publishing and notifying business events among service composition participants. However, we have implemented a relatively simple BPEL-based event dispatcher.

This BPEL-based event dispatcher consists of a large *flow* element in which for each business event a *sequence* is specified. Such a *sequence* element contains a receive activity for receiving a business event notification and a *flow* element in which the business event notification is sent to all interested parties (i.e. COORDINATORS).

The generation of our BPEL-based event dispatcher directly uses the BPMN model in which for each business process task it is calculated which business event need to occur before that business process task can start its execution.

As an alternative to the BPEL-based event dispatcher one could also reuse existing Web services eventing specifications (e.g. WS-Events (Hewlett-Packard, 2003), WS-Eventing (W3C, 2006) or WS-Notification (OASIS, 2006b; Niblett & Graham, 2005)) or approaches to integrate an SOA and an Event-Driven Architecture (EDA) (Monsieur et al., 2007; Juric, 2010).

6.4 Demonstration in a concrete example

6.4.1 Sequence and data dependencies in the travel agency example

The example discussed throughout this section is an extended version of the introductory example described in Chapter 3 (see Section 3.1). In particular, it extends the travel agency example by adding several data dependencies.

Figure 6.4 shows the adapted BPMN process for this example. The business process starts with a data-based gateway which specifies that the business process should continue with the processing of the customer's travel request only if the customer does not have any unpaid invoices. In case there are still unpaid invoices, the customer should be notified and then the business process ends. In order to decide on which path needs to be followed, an overview on the unpaid invoices is required. In this example, we assume that this information is available at the Finance Data Service.

Once a customer's travel request is processed the business process continues with making all necessary bookings. Based on the processed customer request it is decided whether or not a hotel, flight and/or car should be booked. Furthermore, as specified in the BPMN model using unidirectional associations between the 'processed customer request' data object and the booking tasks, the processed customer request contains all relevant information required to make a hotel booking (e.g. desired number of stars, room preferences, etc.), flight booking (e.g. destination, passport number, etc.) or a car rental booking (e.g. driving license number, etc.).

When all bookings are successfully made, the business process continues with the arrangement of an online payment. If the payment is done, the payment is registered and the tickets are sent to the customer. As specified in the BPMN model, a customer's address is required to send the tickets to the customer. In this example we assume that the customer's address is available at the Customer Data Service.

Based on this description we can identify four different types of data

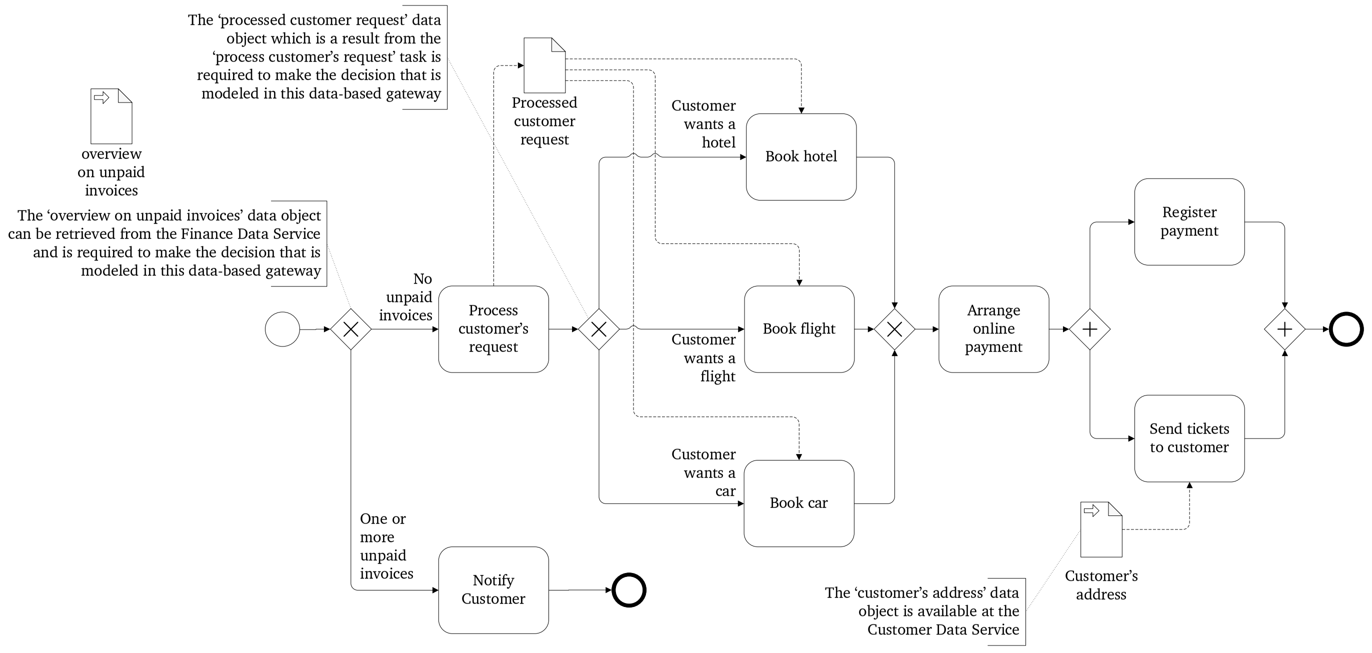

The 'processed customer request' data object which is a result from the 'process customer's request' task is required to make the decision that is modeled in this data-based gateway
overview on unpaid invoices
The 'overview on unpaid invoices' data object can be retrieved from the Finance Data Service and is required to make the decision that is modeled in this data-based gateway
Processed customer request
No unpaid invoices
Process customer's request
Customer wants a hotel
Book hotel
Customer wants a flight
Book flight
Customer wants a car
Book car
One or more unpaid invoices
Notify Customer
Arrange online payment
Register payment
Send tickets to customer
Customer's address
The 'customer's address' data object is available at the Customer Data Service

Figure 6.4: A BPMN (OMG, 2010a) model for the extended travel agency example

dependencies in the business process:

1. Two *Decision Data Dependencies*: one between the Finance Data Service and the 'notify customer' task, and one between the Finance Data Service and the 'process customer's request' task. Since BPMN does not provide the appropriate language constructs, these data dependencies are indicated in Figure 6.4 by the annotation to the first data-based gateway.
2. Three *Decision Data Associations*: the processed customer request is required to decide whether or not the book hotel, book flight and book car task needs to be executed. Since BPMN does not provide the appropriate language constructs, these data dependencies are indicated in Figure 6.4 by the annotation to the second data-based gateway.
3. Three *Data Ouput Input Associations*: the processed customer request is required to execute the book hotel, book flight and book car task. These data dependencies are shown in Figure 6.4 using BPMN data output and data input associations.
4. One *External Data Dependency* between the Customer Data Service and the 'send tickets to customer' task, which describes that the customer's address is required to send the tickets to the customer. This data dependency is partially modeled using the BPMN data input association. An annotation to the data object is used to indicate that the data is available at the Customer Data Service.

6.4.2 Specifying the input models

Representation of the sequence and data dependencies

We have modeled the business process described in the previous subsection (Subsection 6.4.1) in BPMN using Signavio[4], a commercial web-based BPMN modeling application that is built on the Oryx (Decker, Overdick, & Weske, 2008) open-source platform for modeling business processes. In Signavio we exported the BPMN model to an XML format that after adding data structures[5] is directly used as input for our tool[6].

[4]http://www.signavio.com

[5]Data object structures are defined by first adding a *itemDefinition* element that refers to an XML Schema (W3C, 2004) document, and then adding *ItemSubjectRef* attributes (which refer to the *itemDefinition* elements) in the *dataObject* elements.

[6]This BPMN XML file can be downloaded from:
http://merode.econ.kuleuven.be/phd/monsieur/

The Decision Data Dependencies, Decision Data Associations and the complete definition of the External Data Dependency, including the data providers (i.e. the Finance Data Service and Customer Data Service) were formally specified in an XML-based data dependencies model (as described in Subsection 6.2.2 of this chapter) (see Listing 6.12).

```
<dd:dataDependenciesModel
  xmlns:dd="http://servicecoordination.org/dataDependencies"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://servicecoordination.org/
      dataDependencies_DataDependencies.xsd_"
  bpmnLocation="travel_agency.bpmn">

  <dd:dataProviders>
    <dd:dataProvider name="FinanceDataService" receiveOperationName=
        "receiveData"
      receiveRoleName="DataReceiver" requestOperationName="
          requestData"
      requestRoleName="DataProvider" wsdlLocation="
          FinanceDataService.wsdl"
      partnerLinkTypeName="FinanceDataProviderLink" id="
          FinanceDataServiceId" />
    <dd:dataProvider name="CustomerDataService" receiveOperationName
        ="receiveData"
      receiveRoleName="DataReceiver" requestOperationName="
          requestData"
      requestRoleName="DataProvider" wsdlLocation="
          CustomerDataService.wsdl"
      partnerLinkTypeName="CustomerDataProviderLink" id="
          CustomerDataServiceId" />
  </dd:dataProviders>

  <dd:dataDependencies>
    <dd:decisionDataDependency bpmnTaskId="sid-AF13BF30-1281-43FC-93
        DC-1BC1E7E11766"
      dataProviderId="FinanceDataServiceId" name="
          overview_unpaid_invoices-decides-notify_customer"
      id="decisionDD1"/>
    <dd:decisionDataDependency bpmnTaskId="sid-1EF2D490-69E2-466E-
        A425-40CCBADF84C0"
      dataProviderId="FinanceDataServiceId" name="
          overview_unpaid_invoices-decides-process_customer_request"
      id="decisionDD2"/>
    <dd:externalDataDependency dataInputAssociationId="sid-5D2DCB82
        -8F7D-4673-B78D-0CDA5B74F3DB" bpmnTaskId="sid-15FBD3B0-6C87
        -45FF-A8C6-76FB1B6F5F1D"
      dataProviderId="CustomerDataServiceId" name="
          send_tickets_to_customer_requires_customer_address"
      id="externalDD1"/>
  </dd:dataDependencies>

  <dd:dataAssociations>
    <dd:decisionDataAssociation id="decisionDA1" sourceRef="sid
```

```
        −40181B3E−04ED−4F1F−BEF6−9AFC96D5C23D” targetRef=”sid−8
        E812A04−5457−42E3−A860−6357FCB374C6”/>
    <dd:decisionDataAssociation id=”decisionDA2” sourceRef=”sid
        −40181B3E−04ED−4F1F−BEF6−9AFC96D5C23D” targetRef=”sid−
        CE833A54−8CEA−4C17−89DF−822FDDB0D4CB”/>
    <dd:decisionDataAssociation id=”decisionDA3” sourceRef=”sid
        −40181B3E−04ED−4F1F−BEF6−9AFC96D5C23D” targetRef=”sid−
        D63246E4−15DD−4555−B579−E9DECC8AC3F7”/>
  </dd:dataAssociations>
</dd:dataDependenciesModel>
```

Listing 6.12: An XML-based data dependencies model for the travel agency example

Representation of the coordination model

The choice of patterns for managing the sequence and data dependencies in the travel agency example was specified in an XML-based coordination model (as described in Subsection 6.2.3 of this chapter) (see Listing 6.13). The aim of this example is not to find an *optimized* coordination scenario, but the example is supposed to show that using the pattern languages for managing sequence and data dependencies complete coordination scenarios can be automatically generated. Therefore, we more or less randomly choose patterns for managing the dependencies, resulting in the following coordination model:

Sequence dependencies management We choose to manage the sequence dependencies in the travel agency example using three COORDINATORS. A first COORDINATOR, named the COORDINATING CUSTOMER SERVICE, controls two tasks: ’process customer’s request’ and ’send tickets to customer’. A second COORDINATOR, named the booking COORDINATOR, controls the three booking tasks. This pattern could, for instance, be useful because the three controllers for the three booking tasks all need to react to the same business event (i.e. the customer’s request is processed). Furthermore, the controllers all use the ’processed customer request’ data object in order to decide whether a booking should be made. The third COORDINATOR, referred to as the COORDINATING FINANCE SERVICE controls three tasks: ’notify customer’, ’arrange online payment’ and ’register payment’.

External Data Dependency management We decided to manage the External Data Dependency between the Customer Data Service and the ’send tickets to customer’ task using an ACTIVE SERVICE REQUESTER with a DIRECT DATA TRANSMISSION. This means that the COORDINATING

CUSTOMER SERVICE is responsible for requesting the Customer Data Service, and the Mail Service (i.e. the service that supports the 'send tickets to customer' task) receives the customer's address directly from the Customer Data Service.

Data Ouput Input Associations management The 'processed customer request' is required to execute the book hotel, book flight and book car task. All three Data Output Input Associations were managed using an INDIRECT DATA TRANSMISSION so that the service provider responsible for processing the customer request (e.g. the customer service) only needs to send the 'process customer request' once to the booking COORDINATOR.

Listing 6.13 shows how this coordination model was specified in XML.

```
<cm:coordinationModel xmlns:cm="http://servicecoordination.org/
      coordinationModel" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
      instance" xsi:schemaLocation="http://servicecoordination.org/
      coordinationModel coordinationModel.xsd">
  <cm:sequenceDependenciesManagement
     bpmnModel="travel_agency.bpmn">
     <cm:coordinator name="coordinating_customer_service">
       <!-- process customer request -->
       <cm:bpmnTaskId>sid-1EF2D490-69E2-466E-A425-40CCBADF84C0</
             cm:bpmnTaskId>
       <!-- send tickets to customer -->
       <cm:bpmnTaskId>sid-15FBD3B0-6C87-45FF-A8C6-76FB1B6F5F1D</
             cm:bpmnTaskId>
     </cm:coordinator>
     <cm:coordinator name="booking_coordinator">
       <!-- book hotel -->
       <cm:bpmnTaskId>sid-8E812A04-5457-42E3-A860-6357FCB374C6</
             cm:bpmnTaskId>
       <!-- book flight -->
       <cm:bpmnTaskId>sid-CE833A54-8CEA-4C17-89DF-822FDDB0D4CB</
             cm:bpmnTaskId>
       <!-- book car -->
       <cm:bpmnTaskId>sid-D63246E4-15DD-4555-B579-E9DECC8AC3F7</
             cm:bpmnTaskId>
     </cm:coordinator>
     <cm:coordinator name="coordinating_finance_service">
       <!-- notify customer -->
       <cm:bpmnTaskId>sid-AF13BF30-1281-43FC-93DC-1BC1E7E11766</
             cm:bpmnTaskId>
       <!-- arrange online payment -->
       <cm:bpmnTaskId>sid-E372E8A5-00D1-4723-8E47-0A85C8BE7AB0</
             cm:bpmnTaskId>
       <!-- register payment -->
       <cm:bpmnTaskId>sid-E7EC9268-5F89-4AA4-A0E2-0D662690F356</
             cm:bpmnTaskId>
     </cm:coordinator>
  </cm:sequenceDependenciesManagement>
  <cm:dataDependenciesManagement dataDependenciesModel="
      ddmodel_travel_agency.xml">
    <cm:dataDependencyManagement dataDependencyId="externalDD1"
        pattern="
        active_service_requester_with_direct_data_transmission" />
    <cm:dataOutputIsDataInputManagement outputAssociationId="sid
        -0027000C-BAC9-4F96-8B62-39C3847D2FAE" inputAssociationId="
        sid-414F33D7-55F0-4A47-8038-6FFCF52AF81A" pattern="
        indirect_data_transmission" id="DOIDIM1"/>
    <cm:dataOutputIsDataInputManagement outputAssociationId="sid
        -0027000C-BAC9-4F96-8B62-39C3847D2FAE" inputAssociationId="
        sid-76194C29-4F71-4534-8F62-E70B8CB5A272" pattern="
        indirect_data_transmission" id="DOIDIM2"/>
    <cm:dataOutputIsDataInputManagement outputAssociationId="sid
        -0027000C-BAC9-4F96-8B62-39C3847D2FAE" inputAssociationId="
        sid-25A06256-C6B6-4C5F-BB1A-3E240D6E1898" pattern="
        indirect_data_transmission" id="DOIDIM3"/>
  </cm:dataDependenciesManagement>
</cm:coordinationModel>
```

Listing 6.13: An XML-based coordination model for the travel agency example

6.4.3 The generated BPEL and WSDL files

Based on the input models our tool generated 12 BPEL processes and 12 WSDL descriptions. More specifically, one BPEL process and corresponding WSDL description was generated per business process task. For each COORDINATOR the tool generated also one BPEL process and one corresponding WSDL description. Finally, a BPEL process and WSDL description were generated for the BPEL-based event dispatcher.

In this section we only present visual representations of a limited set of generated BPEL processes[7]. In particular, we show how the generated BPEL processes deal with the different types of data dependencies that are present in the travel agency example.

The visual representations were generated using the Eclipse BPEL Designer[8] (Juric, 2006).

[7]All generated BPEL and WSDL files can be downloaded from: http://merode.econ.kuleuven.be/phd/monsieur/

[8]http://www.eclipse.org/bpel

BPEL process for COORDINATING CUSTOMER SERVICE

In Figure 6.5 a visual representation of the BPEL process for the COORDINATING CUSTOMER SERVICE is shown. One can clearly see that the COORDINATING CUSTOMER SERVICE controls two business process tasks: 'process customer request' and 'send tickets to customer'.

The controller for 'process customer request' manages the Decision Data Dependency by first sending a data request to the Finance Data Received. Subsequently this controller receives the information about unpaid invoices and then can decide if it needs to send the business request to the service supporting the 'process customer request' task.

The External Data Dependency between 'send tickets to customer' and the Customer Data Service is managed using an ACTIVE SERVICE REQUESTER and an INDIRECT DATA TRANSMISSION. As a consequence in the BPEL process shown in Figure 6.5 the controller for 'send tickets to customer' first sends a business request to the mail service and then sends a data request to the Customer Data Service.

BPEL process for the service supporting the 'process customer request' task

In Figure 6.6 a visual representation of the BPEL process for the service provider supporting the 'process customer request' task is shown.

The BPEL process shown in Figure 6.6 shows that the data output of 'process customer request' is sent to the booking COORDINATOR. The BPEL process of the booking COORDINATOR is discussed below.

BPEL process for the booking COORDINATOR

In Figure 6.7 a visual representation of the BPEL process for the booking COORDINATOR is shown. One can clearly see that the booking COORDINATOR controls the three booking tasks. When the booking COORDINATOR receives a business event notification describing that a customer request is processed, the COORDINATOR decides whether or not a specific booking needs to be made. This decision is made using the 'processed customer request' that is also received by the booking COORDINATOR. In parallel, the booking COORDINATOR also forwards the 'processed customer request' to the booking services.

Although the Eclipse BPEL designer does not succeed in representing all

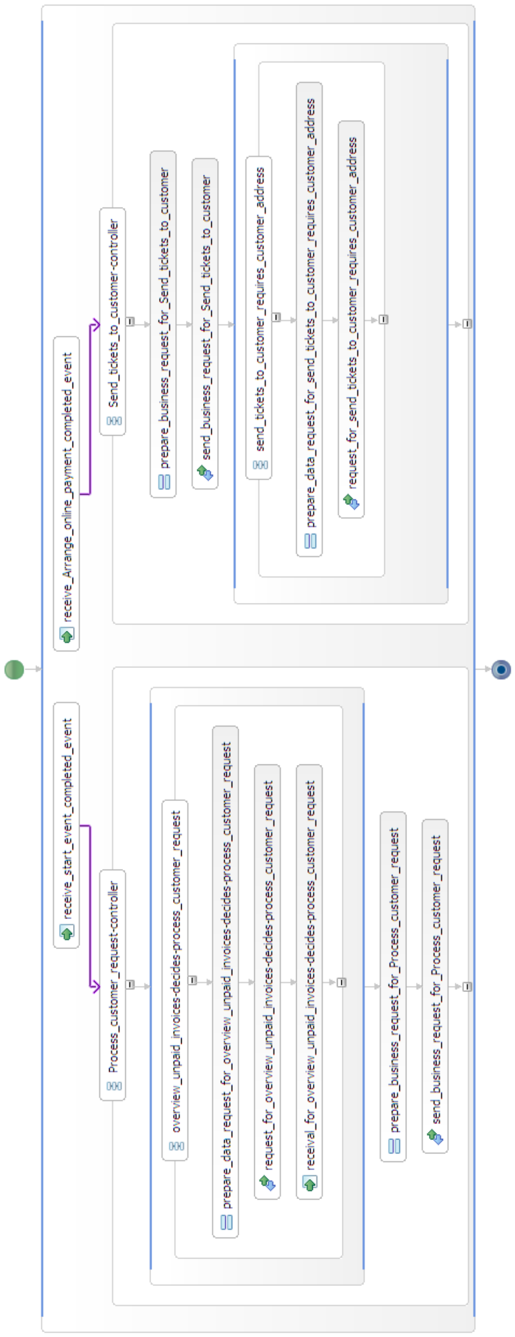

receive_start_event_completed_event
Process_customer_request-controller
overview_unpaid_invoices-decides-process_customer_request
prepare_data_request_for_overview_unpaid_invoices-decides-process_customer_request
request_for_overview_unpaid_invoices-decides-process_customer_request
receival_for_overview_unpaid_invoices-decides-process_customer_request
prepare_business_request_for_Process_customer_request
send_business_request_for_Process_customer_request
receive_Arrange_online_payment_completed_event
Send_tickets_to_customer-controller
prepare_business_request_for_Send_tickets_to_customer
send_business_request_for_Send_tickets_to_customer
send_tickets_to_customer_requires_customer_address
prepare_data_request_for_send_tickets_to_customer_requires_customer_address
request_for_send_tickets_to_customer_requires_customer_address

Figure 6.5: A visual representation of the BPEL (OASIS, 2007) process for the COORDINATING CUSTOMER SERVICE

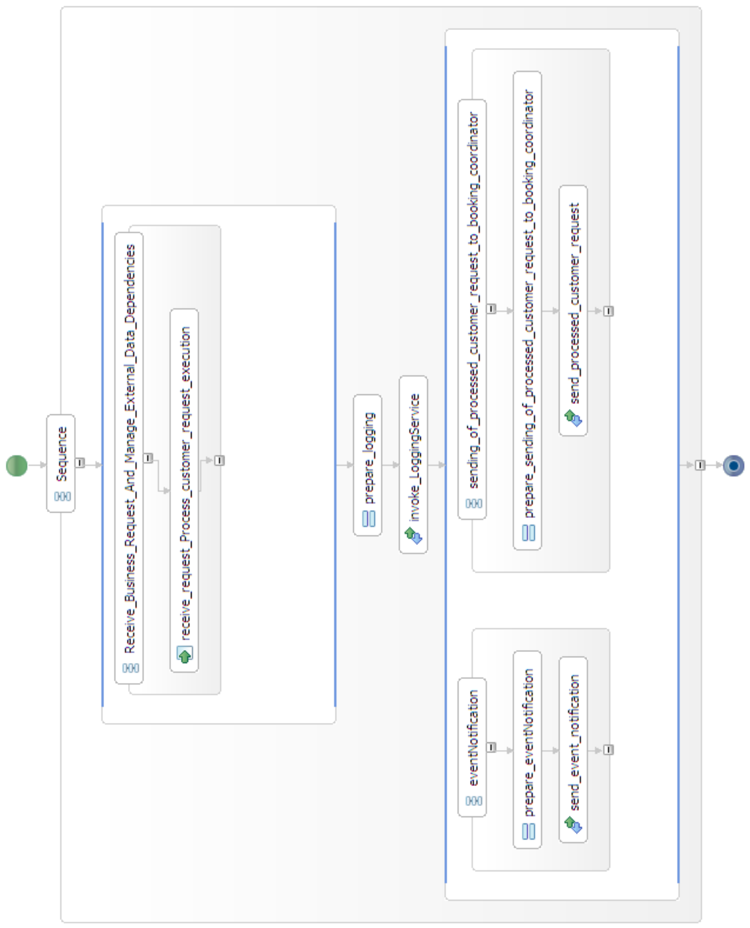

Sequence
Receive_Business_Request_And_Manage_External_Data_Dependencies
receive_request_Process_customer_request_execution
prepare_logging
invoke_LoggingService
sending_of_processed_customer_request_to_booking_coordinator
prepare_sending_of_processed_customer_request_to_booking_coordinator
send_processed_customer_request
eventNotification
prepare_eventNotification
send_event_notification

Figure 6.6: A visual representation of the BPEL (OASIS, 2007) process for the service provider supporting the 'process customer request' task

BPEL links in a very clear way, one should be able to see that a 'processed customer request' is sent to the booking service only if the 'processed customer request' is received. Similarly, the BPEL links are organized in a way that the BPEL *sequence* for the controller of a booking task is only started when both the 'processed customer request' is received and the event notification 'process customer request completed' is received.

BPEL process for the service supporting the 'book flight' task

In Figure 6.8 a visual representation of the BPEL process for the service provider supporting the 'book flight' task is shown.

The BPEL process represented in Figure 6.8 shows that the flight booking (i.e. the logging activity) is only made when both a business request and a 'processed customer request' is received (i.e. management of the Data Output Input Association).

6.4.4 Testing generated BPEL processes

In order to test the correctness of the generated BPEL processes we deployed several test examples, including the example described in this section, to a BPEL engine.

It should be noted that today's BPEL engines do not always completely support the BPEL specification. For example, the BPEL engine by Sun, currently does not provide support for BPEL links (Jennings & Salter, 2008). However, we intensively use BPEL links in our generated BPEL processes (i.e. to link business event notifications to controllers), which makes it impossible to test our generated BPEL processes in the BPEL Engine by Sun. Another issue that we discovered was the combination of the *createInstance=yes* attribute and the correlation attribute *initiate=join* to indicate that a new BPEL instance needs to be created only if the message received cannot be correlated to an existing BPEL instance. This combination is also frequently used throughout our generated code (e.g. for correlating several business event notifications to the same business process in a COORDINATOR). However, although this is also legal BPEL 2.0 code, it is, for instance, not supported by the Apache ODE Engine.

Generated BPEL processes were successfully tested using the OW2 Orchestra BPEL engine[9]. All BPEL processes were always deployed to the same BPEL engine running on a single machine. In a real-life business setting

[9]http://orchestra.ow2.org/

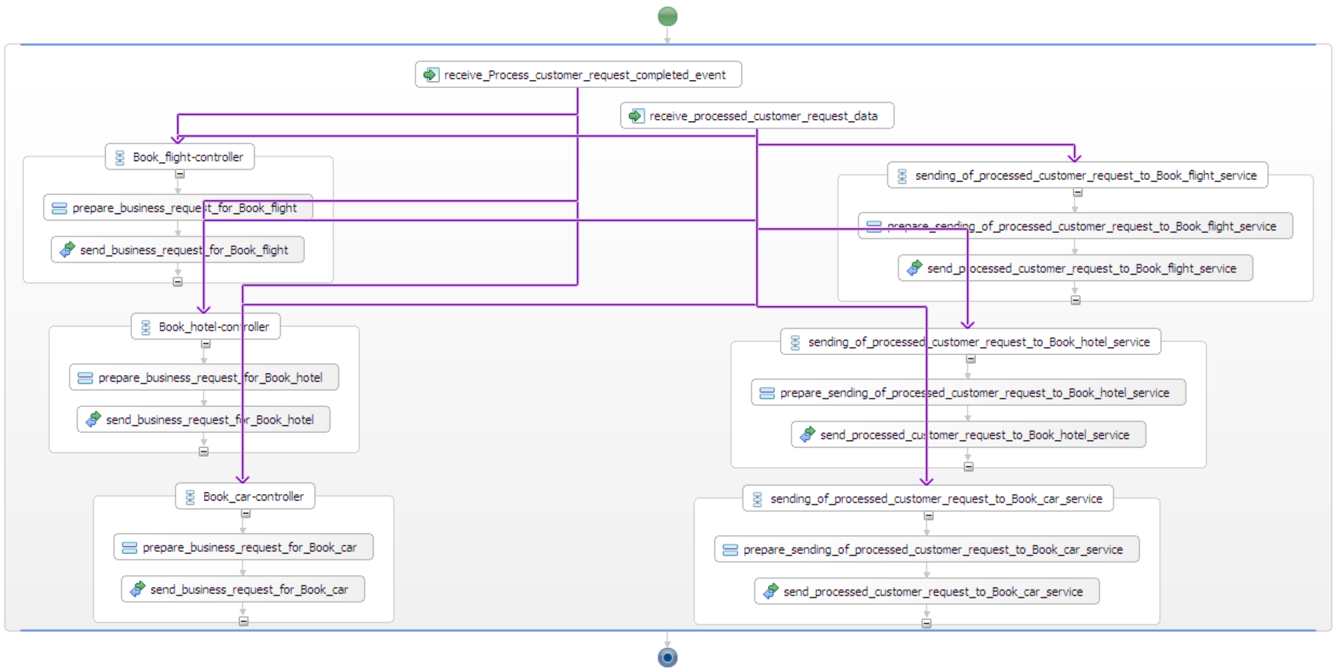

receive_Process_customer_request_completed_event
receive_processed_customer_request_data
Book_flight-controller
prepare_business_request_for_Book_flight
send_business_request_for_Book_flight
Book_hotel-controller
prepare_business_request_for_Book_hotel
send_business_request_for_Book_hotel
Book_car-controller
prepare_business_request_for_Book_car
send_business_request_for_Book_car
sending_of_processed_customer_request_to_Book_flight_service
prepare_sending_of_processed_customer_request_to_Book_flight_service
send_processed_customer_request_to_Book_flight_service
sending_of_processed_customer_request_to_Book_hotel_service
prepare_sending_of_processed_customer_request_to_Book_hotel_service
send_processed_customer_request_to_Book_hotel_service
sending_of_processed_customer_request_to_Book_car_service
prepare_sending_of_processed_customer_request_to_Book_car_service
send_processed_customer_request_to_Book_car_service

Figure 6.7: A visual representation of the BPEL (OASIS, 2007) process for the booking COORDINATOR

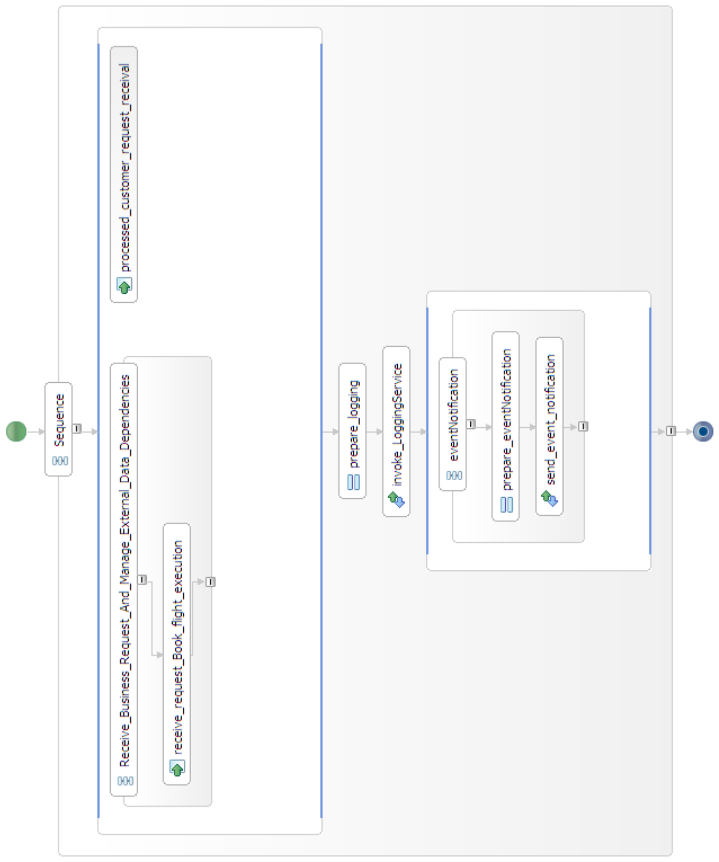

Sequence
Receive_Business_Request_And_Manage_External_Data_Dependencies
receive_request_Book_flight_execution
processed_customer_request_receival
prepare_logging
invoke_LoggingService
eventNotification
prepare_eventNotification
send_event_notification

Figure 6.8: A visual representation of the BPEL (OASIS, 2007) process for the service provider supporting the 'book flight' task

typically services are running on different machines, sometimes in different BPEL engines and perhaps even in different locations or companies. However, by simply changing the port locations in the generated WSDL files the BPEL processes can be easily deployed to different engines in different locations, if desired.

To facilitate the test process, we mostly generated additional BPEL activities that invoke a Web service that simply logged all received messages. As such we could easily log when, for instance, a business process task was supposed to start in a service provider.

6.5 Conclusion

In this chapter we have shown that it is possible to automatically generated coordination scenarios from business process specifications by letting developers pick specific patterns for dependencies management. We developed a tool for this generation, which was described in Section 6.3. The input for this tool consists of three elements. A first input element is a BPMN (OMG, 2010a) model (serialized in XML) specifying the business process that needs to be implemented. This BPMN model typically specifies all sequence dependencies and those data dependencies that can be defined using BPMN (see Section 2.5.1 in Chapter 2 and Subsection 6.2.1 in this chapter for more details on representing data dependencies in BPMN). Other data dependencies are defined in a separate XML file, named the data dependencies model, which forms the second input element. The third input element is another XML file, which is referred to as the coordination model and specifies the choice of patterns for dependencies management that will be used for generating the desired coordination scenario. Based on these three elements the tool generates a complete coordination scenario in the form of several BPEL (OASIS, 2007) processes.

In Section 6.4 the tool for generating coordination scenarios was demonstrated using a travel agency example.

We must all take time to do enough thinking to formulate our own conclusions

– **Thomas John Watson, Sr.** (1874–1956), Former president of International Business Machines (IBM)

7 Conclusions

In this chapter we summarize the thesis and its contributions by evaluating the research objectives (see Section 7.1). We also discuss limitations and issues for further research (see Section 7.2).

7.1 Research objectives evaluation

In general, the solution described in this thesis consists of three main parts: a pattern language for managing sequence dependencies, a pattern language for managing data dependencies and a tool for pattern-based coordination. In this section we describe why this solution answers the research questions that we posed in Chapter 1 (see Section 1.2). We show this by evaluating the three research objectives that we formulated in Chapter 1 (see Section 1.3) and which were directly derived from the research questions. Figure 7.1 gives a high-level view on the relationship between the solution presented in this dissertation and the research goals, research questions and research objectives.

The **first research objective** described that our solution should entail a set of *fundamental building blocks* for *coordination* logic. As explained in our research methodology (see Section 1.3), we followed the coordination definition by Malone and Crowston (1994), who define coordination as *managing dependencies* between activities. In Chapter 2 we motivated why we further decomposed the service coordination problem into managing both *sequence* and *data dependencies*. In Chapter 3 and 4 we proposed a pattern language for managing sequence dependencies and data dependencies, respectively. The

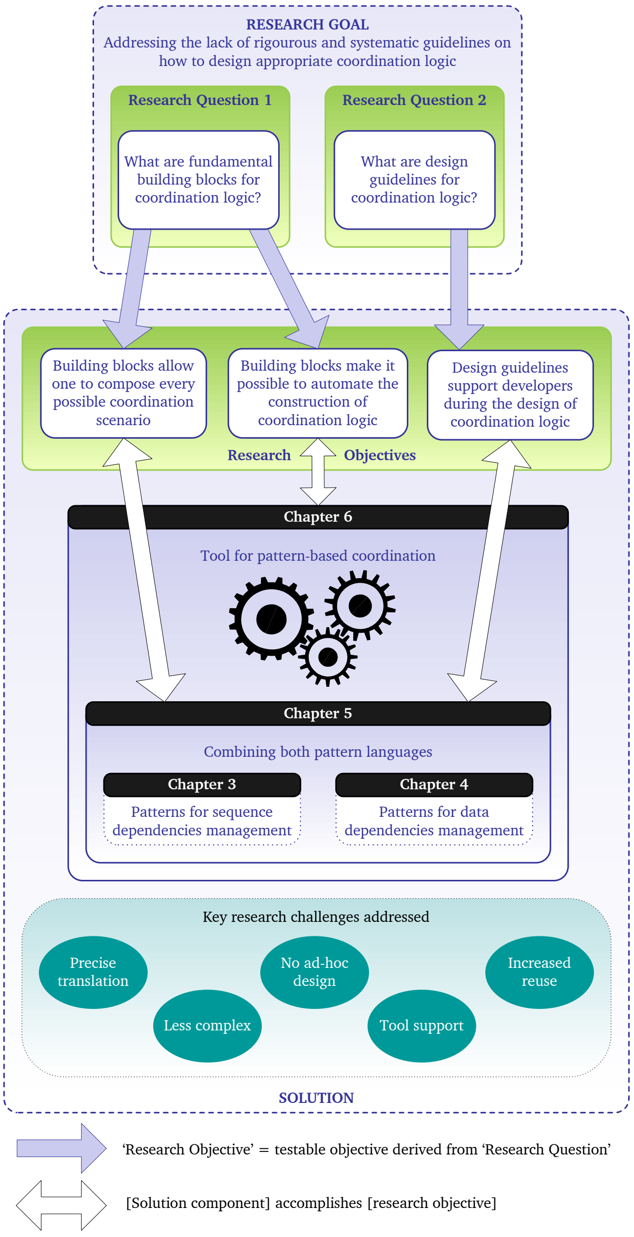

RESEARCH GOAL
Addressing the lack of rigourous and systematic guidelines on how to design appropriate coordination logic
Research Question 1
What are fundamental building blocks for coordination logic?
Research Question 2
What are design guidelines for coordination logic?
Building blocks allow one to compose every possible coordination scenario
Building blocks make it possible to automate the construction of coordination logic
Design guidelines support developers during the design of coordination logic
Research Objectives
Chapter 6
Tool for pattern-based coordination
Chapter 5
Combining both pattern languages
Chapter 3
Patterns for sequence dependencies management
Chapter 4
Patterns for data dependencies management
Key research challenges addressed
Precise translation
No ad-hoc design
Increased reuse
Less complex
Tool support
SOLUTION
'Research Objective' = testable objective derived from 'Research Question'
[Solution component] accomplishes [research objective]

Figure 7.1: The relationship between the solution and our research goal, research questions and research objectives

patterns in both pattern languages form the building blocks for coordination logic.

Based on the pattern-based building blocks we aimed to semi-automate the construction of a coordination scenario by letting developers pick specific patterns for dependencies management and automatically generate a complete coordination scenario from a business process specification. We developed a tool for achieving this aim, which was presented in Chapter 6. The input for this tool consists of three elements. A first input element is a BPMN (OMG, 2010a) model (serialized in XML) specifying the business process that needs to be implemented. This BPMN model typically specifies all sequence dependencies and those data dependencies that can be defined using BPMN (see Section 2.5.1 in Chapter 2 and Chapter 6 for more details). Other data dependencies are defined in a separate XML file, which forms the second input element. The third input element is another XML file, which specifies the choice of patterns for dependencies management that will be used for generating the desired coordination scenario.

Based on these three elements the tool generates a complete coordination scenario. In other words, thanks to the pattern languages the tool offers a **precise translation** between business process design and implementation. This coordination scenario consists of a set of BPEL (OASIS, 2007) processes. Although the tool is limited to generating coordination scenarios specified in BPEL, this does not have to be a downside for two important reasons. First, BPEL is the most widely used standard for specifying Web service orchestrations. Second, although BPEL originally only was developed for specifying Web service orchestrations, in the upcoming standard *Service Component Architecture* (Open SOA Collaboration, 2007) BPEL plays the role of the de facto standard language to combine several components into a composite application, regardless of the underlying technology used in the different components (e.g. Java Message Service (JMS), SOAP Web service, Enterprise Java Beans (EJBs)) (Edwards, 2007).

The successful implementation of the tool described above proves the value of the pattern languages presented in this dissertation. It shows that the patterns can be used as building blocks for coordination logic and hence form the basis for a **development tool that supports service composers** by facilitating or even automating the coordination challenges associated with service composition.

The **second research objective** focused on the different ways in which the building blocks or the patterns can be combined. More specifically, we aimed to construct every potential coordination scenario using the patterns. In Subsection 3.4.2 in Chapter 3 we first showed that every interaction in a service composition can be composed by the patterns described for managing sequence dependencies. Subsequently, we demonstrated that the pattern language is also compatible with coordination styles such as centralized or decentralized coordination. Together with our detailed discussion on orchestration and choreography in Chapter 2 (see Section 2.2), that gives service composers more insight into designing coordination logic and hence potentially reduces the complexity of message-based service compositions. In Section 4.5 of Chapter 4 we showed that *all* potential coordination scenarios for managing a data dependency can be composed by combining the patterns that were presented in that chapter. For this completeness confirmation of our pattern language, we declaratively specified what a coordination scenario should accomplish and in which message exchanges a service requester, service provider and data provider can be involved. For example, it is easy to understand that in every coordination scenario the service provider must receive data from another entity. All conditions that should be met by a coordination scenario were specified in Prolog, so that an execution of the Prolog program (i.e. a query that calculates or derives all solutions) lists all possible coordination scenarios. As such it turned out that the list of all possible coordination scenarios exactly matches the set of coordination scenarios that can be composed by combining our patterns.

The fact that the pattern languages presented in this dissertation are complete (i.e. every possible coordination scenario to manage sequence and data dependencies can be composed from the patterns), has important consequences for the way coordination logic is constructed in the future. In the past researchers argued that an over-emphasis on service interactions is at the expense of other aspects like business goals (Ko, Lee, & Lee, 2009; Koubarakis & Plexousakis, 1999; Andersson, Bider, Johannesson, & Perjons, 2005). However, since all potential coordination scenarios can be composed from the patterns, it makes no sense anymore to spend expensive time designing coordination scenarios at the level of service interactions and message exchanges. In other words, the design and construction of coordination logic becomes **less complex**. The use of the pattern language raises the abstraction level and in line with the principles of the Model-Driven Architecture (Kleppe, Warmer, & Bast, 2003) contributes to an efficient (e.g. automatically generated coordination scenarios) and effective (e.g. less errors and more consistency) development of service composition. Furthermore, developers do not need to repeatedly implement the same implementation patterns, resulting in an **increased reuse**.

In the **third research objective** we expected that our solution (i.e. the two pattern languages) guides developers to choose and combine the building blocks in a way that an optimized coordination scenario is composed.

Therefore, for each pattern we identified the consequences (e.g. with respect to loose coupling or data confidentiality) so that service composers can select patterns on the basis of these consequences and their requirements. Based on these consequences we have presented concrete guidelines on how to combine the patterns to compose coordination scenarios (see Subsection 3.3.1 in Chapter 3 and Subsection 4.3.2 in Chapter 4). Furthermore, we have applied the patterns and guidelines for managing sequence dependencies to the basic control flow patterns (Van der Aalst et al., 2003) (see Subsection 3.3.2). The patterns and guidelines for data dependencies management were applied in both a fictive and real-life business case to construct an optimized coordination scenario (see Subsections 4.3.3 and 4.4.2 in Chapter 4). Thanks to the design guidelines it can be **more easily avoided that service composition and coordination are performed in an ad-hoc fashion**.

7.2 Limitations and issues for future research

Throughout this thesis we mentioned several assumptions that we made in the course of our research. For example, when we discussed the different types of dependencies that can occur in service compositions, we emphasized the way we defined a data dependency. More specifically, it is important to know that we defined a data dependency between two services as a situation in which one service needs data from second service, *without* the need for blocking the data in the first service (see Section 2.1 in Chapter 2). In case data needs to be blocked and cannot be read and/or changed by other services, coordination scenarios would become even more complex. However, this is considered to be another research problem that needs to be addressed by future research.

Another assumption was described in the context of the patterns for managing sequence dependencies (see Section 3.2.1). In particular we made the assumption that business events are generated and published by service providers and notifications of these events are disseminated by so called business event dispatchers. However, possibly service providers not always publish all relevant business events. In such cases, additional patterns (e.g. event listeners, polling patterns, etc.) could be useful to request business event information from service providers.

In the rest of this section we discuss several other assumptions, limitations and issues for future research that were not mentioned earlier in this thesis.

7.2.1 Sequence versus action dependencies

When using the pattern language for managing sequence dependencies (see Chapter 3) business process tasks are triggered by sending so called business or service requests to a service provider that supports the execution of that task. These requests are sent when the controller has received all necessary business event notifications. For instance, in case of a SYNCHRONIZATION (Van der Aalst et al., 2003) this means that the controller needs to receive several business event notifications that indicate that all tasks before the parallel gateway are completed successfully. Once all relevant event notifications are received, the controller triggers the business task by sending a business request to the service provider. Hence, the sequence dependencies are considered as some kind of *trigger* or *action* dependencies, which means that tasks need to be triggered *directly* after a specified set of events occurred.

However, in reality, business process tasks do not always need to be triggered when previous business process task are completed. For instance, widely-used business process management suites (Møller, Maack, & Tan, 2008) (e.g. Oracle BPM Suite (Jellema & Dikmans, 2010), IBM WebSphere Process Server (Iyengar et al., 2007), JBoss jBPM (Cumberlidge, 2007)) often provide task inboxes to end-users. Such inboxes list all tasks that an end-user or employee is responsible for. Typically, tasks enter these inboxes when previous business process tasks are completed. Then it is up to the end-user to decide when he or she should start the task execution. Moreover, additional preconditions for starting the task execution might be present and need to be checked by the controller. That would mean that the event notifications are necessary but not sufficient for triggering tasks. In line with this idea is the use of *event-condition-action* rules. This kind of rules originated in active database systems (Paton & Díaz, 1999) but has also its place in workflow management systems (Lu & Sadiq, 2007; Kappel, Rausch-Schott, & Retschitzegger, 1998).

Hence, more research is needed to implement the event-condition-action principle in the coordination patterns and to make a better distinction between sequence and trigger or action dependencies. Nevertheless, the patterns presented in this thesis provide a fundamental basis for this extension. First of all the idea of event-condition-action rules can be relatively easy implemented by assuming that checking preconditions is modeled as a separated business task (as shown in the SEQUENCE (Van der Aalst et al., 2003) example in Figure 7.2). However, this is probably not the most appropriate solution because this increases the complexity of the business process model and business analysts prefer to hide such execution details (Dreiling, Rosemann, Van der Aalst, & Sadiq, 2008). Second, assuming that there exists

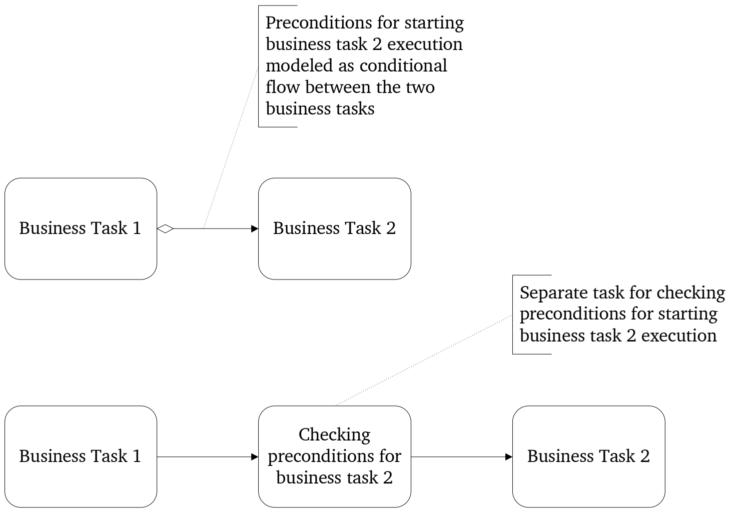

Preconditions for starting business task 2 execution modeled as conditional flow between the two business tasks
Business Task 1
Business Task 2
Separate task for checking preconditions for starting business task 2 execution
Business Task 1
Checking preconditions for business task 2
Business Task 2

Figure 7.2: Modeling preconditions in a SEQUENCE (Van der Aalst et al., 2003)

a way of checking preconditions by consuming a certain service, checking preconditions can be modeled as an additional data dependency. Indeed, the controller needs that data (i.e. the result of checking preconditions) in order to decide when and if it should start the task execution by sending a business request to a service provider. In that way, the event-condition-action principle is already (partially) implemented, because data requirements for deciding on a task execution are included in our demonstration (see Subsection 5.1 in Chapter 4).

Ultimately, checking if all relevant events have occurred and all (other) preconditions are met for triggering a task execution can be completely separated from the entity that triggers tasks, resulting in a STATELESS PROCESS ENACTMENT (Haesen et al., 2007). While the controllers as defined in Chapter 3 hold some process state in the form of received business event notifications, in STATELESS PROCESS ENACTMENT the state of a process is derived from the state of the business objects and the triggering and completion of activities is derived from the state of business objects (Haesen, Snoeck, Lemahieu, & Poelmans, 2009). Future research should study how STATELESS PROCESS ENACTMENT can be integrated with the coordination patterns presented in this thesis.

7.2.2 Extending the execution model

As every pattern the patterns for managing sequence dependencies provide a solution to a problem (i.e. managing sequence dependencies) in a certain *context*. In Subsection 3.2.1 of Chapter 3 we have described the context in which our patterns can be applied. In this context we stated that the execution of business process task can be started by sending a *business* or *service request* to a service provider supporting that task. Throughout this thesis we assumed that once a controller sends a request to a service provider, the latter service successfully starts and eventually completes the execution of the business process task. Hence, there needs to be a contract or agreement with the service provider that specifies what a controller can expect from the service provider (e.g. specifying preconditions, postconditions, invariants and nonfunctional requirements (Jones, 2005)). In practice, this means that a controller needs to be able to deal with errors or unexpected behavior that can occur during the task execution. Furthermore, a business process can specify compensation tasks that need to be triggered in case of concrete failures. It should also mentioned that a failure is not always defined in an unique way. What could be successful for a service provider, could perhaps be not sufficient for a service consumer (e.g. a service consumer possibly desires a higher level of quality). This all creates new coordination challenges for coordination patterns. In fact, this all comes down to managing additional dependencies. For example, as discussed in the chapter on related work Bhiri et al. (2005) have described several *transactional* dependencies, e.g. compensation, cancellation, etc. dependencies (see Section 2.1 in Chapter 2). Similarly, BPMN (OMG, 2010a) offers modeling constructs for exception handling, transactions, and compensation. Applied to the patterns presented in this thesis this means that future patterns need to deal with several new types of business events (e.g. an event indicating a task execution failure) and several types of service requests (e.g. a request to compensate a task).

In Figure 7.3 a transaction is represented using BPMN (OMG, 2010a). The transaction, which is based on the introductory example presented in Chapter 3 (see Section 3.1), consists of three parallel tasks: book hotel, book car rental and book flight. If one or more bookings fail, all other bookings need to be compensated using the corresponding compensating tasks.

Since the compensation controller of the hotel booking service and the compensation controller of the flight booking service both need event information about the errors during the car rental booking, it is useful to create one coordinator that is the compensation controller of both the hotel booking service and flight booking service. This means that the hotel booking service and flight booking service are controlled by a COORDINATOR. In a similar

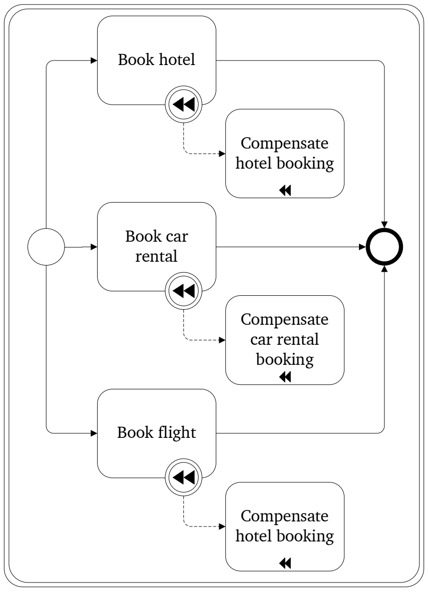


Figure 7.3: BPMN (OMG, 2010a) representation of transaction consisting of hotel, car rental and flight booking tasks and corresponding compensating tasks

way, one can argue that it is useful to have one compensation coordinator for the car rental booking service and flight booking service. Hence, in order to minimize the complexity of the coordination scenario (i.e. minimizing the event communication) it is preferable to combine all compensation controllers. Figure 7.5 shows such a coordination scenario using an INDEPENDENT COORDINATOR, while Figure 7.4 shows a more complex coordination scenario using three INDEPENDENT CONTROLLERS. The dashed arrows represent event information transfers, while solid arrows denote service requests. In Figure 7.7 (simple and optimized) and Figure 7.6 (complex) BPMN (OMG, 2010a) representations of these scenarios are shown.

7.2.3 Time dimensions in data dependencies management

An important limitation of our pattern language for managing data dependencies is the absence of a time dimension. The patterns can be applied both before and after a service request is sent to a service provider. However, this time-related aspect potentially influences other forces (e.g. data can be out-of-date). In our opinion, including this time dimension would have

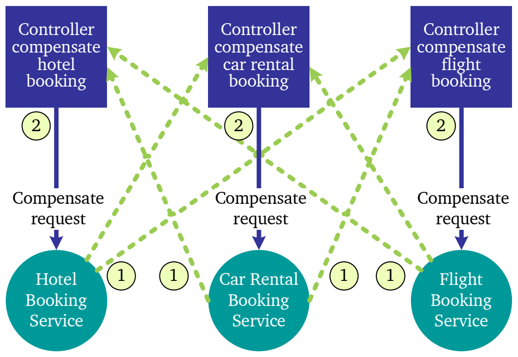

Controller compensate hotel booking
Controller compensate car rental booking
Controller compensate flight booking
2
2
2
Compensate request
Compensate request
Compensate request
1
1
1
1
Hotel Booking Service
Car Rental Booking Service
Flight Booking Service

Figure 7.4: Simplified representation of complex transaction coordination

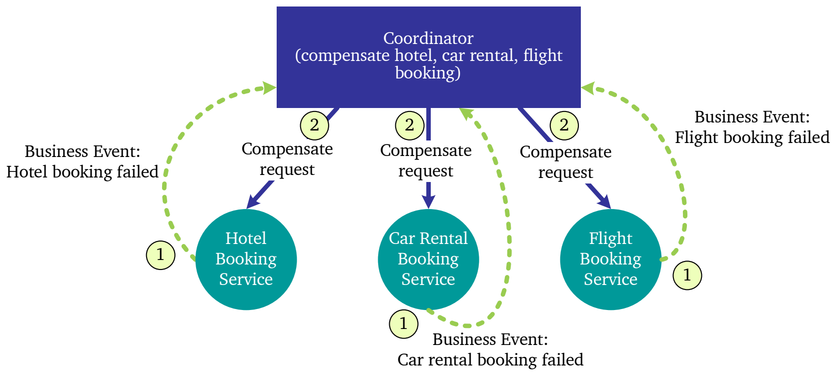

Coordinator (compensate hotel, car rental, flight booking)
Business Event: Hotel booking failed
Business Event: Flight booking failed
Business Event: Car rental booking failed
2
2
2
Compensate request
Compensate request
Compensate request
1
1
1
Hotel Booking Service
Car Rental Booking Service
Flight Booking Service

Figure 7.5: Simplified representation of simple (optimized) transaction coordination

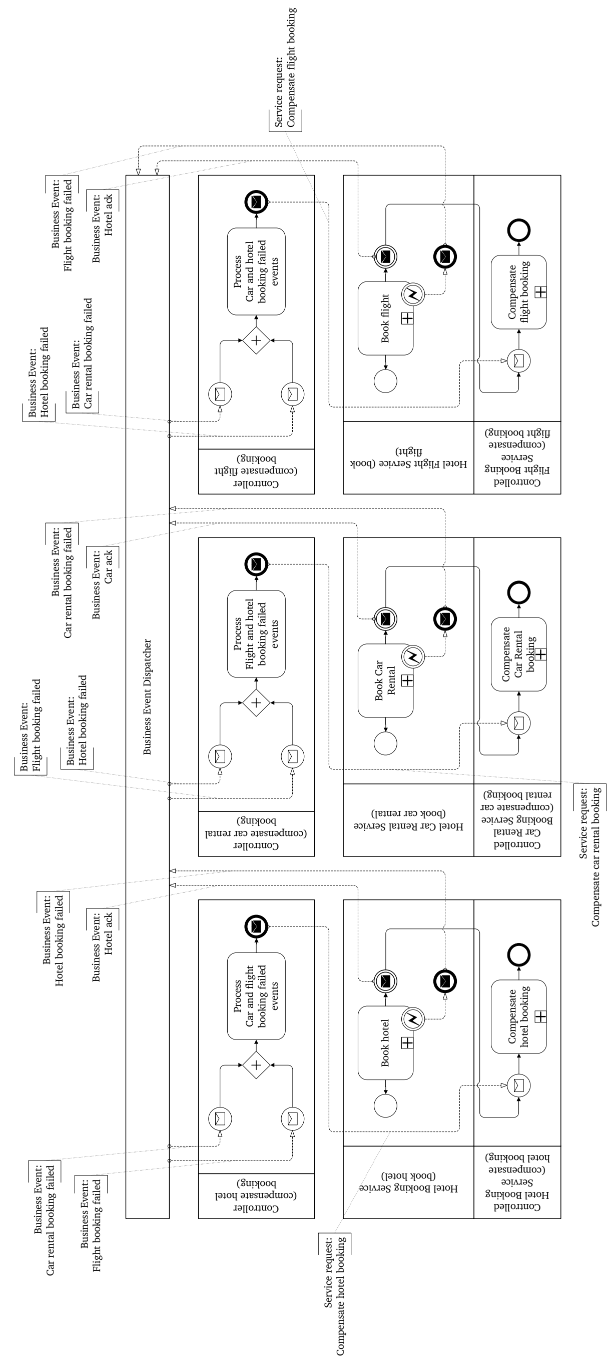

Business Event Dispatcher
Business Event: Flight booking failed
Business Event: Hotel ack
Business Event: Hotel booking failed
Business Event: Car rental booking failed
Service request: Compensate flight booking
Controller (compensate flight booking)
Process Car and hotel booking failed events
Hotel Flight Service (book flight)
Book flight
Controlled Flight Booking Service (compensate flight booking)
Compensate flight booking
Business Event: Car rental booking failed
Business Event: Car ack
Business Event: Flight booking failed
Business Event: Hotel booking failed
Controller (compensate car rental booking)
Process Flight and hotel booking failed events
Hotel Car Rental Service (book car rental)
Book Car Rental
Controlled Car Rental Booking Service (compensate car rental booking)
Compensate Car Rental booking
Service request: Compensate car rental booking
Business Event: Hotel booking failed
Business Event: Hotel ack
Business Event: Car rental booking failed
Business Event: Flight booking failed
Controller (compensate hotel booking)
Process Car and flight booking failed events
Hotel Booking Service (book hotel)
Book hotel
Controlled Hotel Booking Service (compensate hotel booking)
Compensate hotel booking
Service request: Compensate hotel booking

Figure 7.6: BPMN (OMG, 2010a) representation of complex transaction coordination

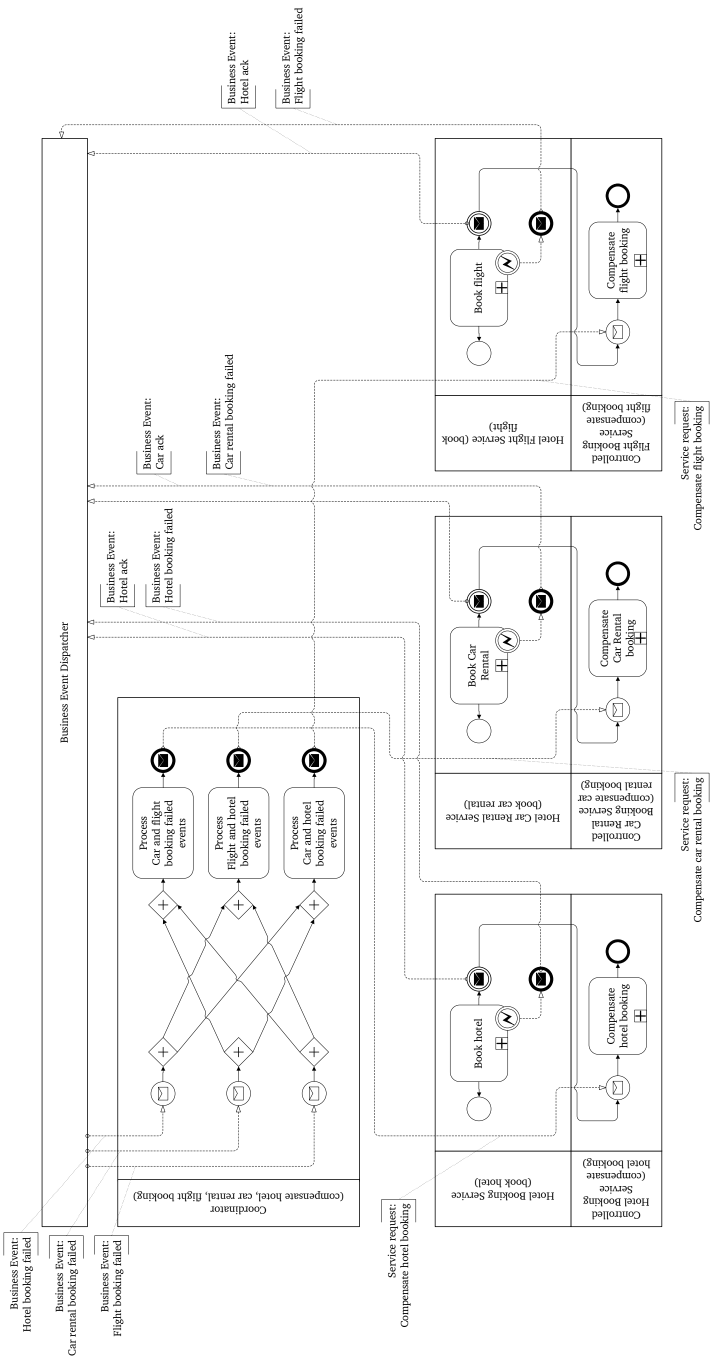

Business Event: Hotel ack
Business Event: Flight booking failed
Business Event: Car ack
Business Event: Car rental booking failed
Business Event: Hotel ack
Business Event: Hotel booking failed
Business Event Dispatcher
Coordinator (compensate hotel, car rental, flight booking)
Process Car and flight booking failed events
Process Flight and hotel booking failed events
Process Car and hotel booking failed events
Hotel Booking Service (book hotel)
Controlled Hotel Booking Service (compensate hotel booking)
Book hotel
Compensate hotel booking
Hotel Car Rental Service (book car rental)
Controlled Car Rental Booking Service (compensate car rental booking)
Book Car Rental
Compensate Car Rental booking
Hotel Flight Service (book flight)
Controlled Flight Booking Service (compensate flight booking)
Book flight
Compensate flight booking
Service request: Compensate flight booking
Service request: Compensate car rental booking
Service request: Compensate hotel booking
Business Event: Hotel booking failed
Business Event: Car rental booking failed
Business Event: Flight booking failed

Figure 7.7: BPMN (OMG, 2010a) representation of simple (optimized) transaction coordination

overloaded the pattern language drastically and the overall message that this dissertation wants to bring would have moved to the background. Therefore, more research is needed to investigate time-related issues. Note that the demonstration described in this thesis (see Chapter 6) assumes that data requests and provisioning always occurs after a service request is sent.

7.2.4 Additional validation

As described in Chapter 1 a typical challenge that comes with the development of service-based systems is to tackle the complexity of message-based implementations. Service composition too often requires time-consuming hand coding and low-level programming (see Section 1.1.2). Since our tool for pattern-based coordination generates a complete coordination scenario based on a business process specification and a selection of patterns, we claim that developers need to spend less time designing service interactions. In other words, by using our tool and a pattern-based coordination approach we claim a less complex development of the message-based implementation (see Section 7.1). However, in order to completely prove this claim it is necessary to share our tool with service engineers so that they could confirm this strength of our approach. Future research could also include specific complexity measures for BPEL processes (Cardoso, 2007).

Sharing this tool with service engineers is also necessary to further assess the value of the design guidelines that are included in our pattern languages. Although, our patterns were applied to a real-life case (e.g. see Section 4.4.2 in Chapter 4) more research is needed to quantitatively evaluate that the use of the patterns contributes to a more efficient and effective development of coordination scenarios.

7.2.5 Joined forces

Both the pattern language for managing sequence dependencies and the pattern language for managing data dependencies, include a list of forces that together with pattern consequences can be used as a basis for concrete design guidelines. Since a typical coordination scenario manages both sequence and data dependencies one needs to follow both sets of design guidelines. However, although it is not discussed in this thesis, some forces present in both pattern languages are related to each other (e.g. coupling, access restrictions and flexibility/robustness change). Hence, potentially it is necessary to better integrate both sets of design guidelines that were based on the forces and pattern consequences. Furthermore, some choices in the application of one pattern language possibly limit the potential choices for the application of the other pattern language. Therefore, future research could focus on the pattern forces and identify joined forces.

7.2.6 Composite data requests

In the real-life case in which our pattern language for managing data dependencies was applied (see Section 4.4.2 in Chapter 4) the final solution for the management of the data dependencies was constructed by combining several coordination scenarios, each taking care of a particular set of data. Each coordination scenario always consists of one data request, one data transmission and one data provider. However, sometimes a service provider's data request can only be fulfilled by combining data from several data providers. Such scenarios require more complex coordinations scenarios in which, for example, a service needs to be responsible for splitting up data requests or combining data. In the future, we intend to extend the patterns described in Chapter 4 for composing such coordination scenarios.

7.2.7 Patterns for self-adaptive service coordination

As explained in both the introduction of this thesis (see Sections 1.1 and 1.2 in Chapter 1) and in Section 7.1 of this Chapter, the coordination patterns presented in this thesis help to make service-based systems more flexible. This is mainly because the patterns provide guidelines that help service composers to effectively and efficiently design an optimized coordination scenario. Moreover, the service-based system can more easily and rapidly adapt to any change in a business process, because implementing coordination logic can be automated (see Chapter 6).

However, future software systems will have to operate in a constantly evolving environment, requiring *self-adaptive* service-based systems. Such systems automatically and autonomously adapt their behavior at runtime to respond to evolving requirements, changes in its context, as well as failures of component services (Di Nitto et al., 2008).

In the past several advances are made to achieve run-time adaptability in service-based systems. We first briefly review some research results and then discuss how the patterns presented in this thesis can potentially form the basis for future research on self-adaptive service-based systems.

Current approaches toward adaptive service-based systems

Ardagna, Comuzzi, Mussi, Pernici, and Plebani (2007) developed PAWS (Processes with Adaptive Web Services), a framework for flexible and adaptive execution of managed service-based processes. The main contribution of PAWS is twofold. PAWS should select the best available services for executing the process and define the most appropriate quality-of-service (QoS) levels for delivering them. Second, PAWS should guarantee service provisioning, even in case of failures, through recovery actions and self-adaptation if the context changes. To meet these goals, PAWS provides methods and a toolset to support design-time specification of all information required for automatic runtime adaptation of processes according to dynamically changing user preferences and context. In general, PAWS provides flexibility in terms of optimization, mediation, and self healing functionalities.

In their research Ardagna and Pernici (2007) proposed an advanced approach to the QoS constrained Web service selection problem. In their approach the Web service selection problem is formalized as a mixed integer linear programming problem, loops peeling is adopted in the optimization, and constraints posed by stateful Web services are considered. Moreover, negotiation techniques are exploited to identify a feasible solution of the problem, if one does not exist.

METEOR-S is a framework for (semi-)automated configuration of service compositions by addressing the following two issues (Verma, Gomadam, Sheth, Miller, & Wu, 2005). The first issue is about dynamically selecting optimal partner Web services for a service composition based on process constraints. The second aspect aims at facilitating interaction with the optimal partner Web services in the presence of data and protocol heterogeneities, as well as, supporting run-time reconfiguration in presence of Web service invocation errors.

Williams, Battle, and Cuadrado (2006) have developed a Web services protocol mediation framework that allows interaction between two services despite the difference of the protocols they rely on. Their approach is centered on the identification of common domain specific protocol independent communicative acts; the description of abstract protocols which constrain the sequencing of communicative acts; and the description of concrete protocols that describe the mechanisms by which the client of a web service interface can utter and perceive communicative acts.

Denaro, Pezzé, Tosi, and Schilling (2006) solved similar issues. They proposed an approach that enables clients to automatically adapt their behavior to alternative Web services that provide compatible functionality through different interaction protocols. It uses an infrastructure that traces the successful interactions of the Web services, automatically synthesize models that approximate the interaction protocols, and steer client-side adaptations at runtime.

The SCENE platform provides linguistic and infrastructural mechanisms to support self-configuration of a service composition (Colombo, Di Nitto, & Mauri, 2006). The idea behind SCENE is that the problem of dynamic reconfiguration of service compositions has to be tackled at two levels: on the one side, the runtime platform should be flexible enough to support the selection of alternative services, the negotiation of their service level agreements, and the partial replanning of a composition. On the other side, the language used to develop the composition should support the designer in defining the constraints and conditions that regulate selection, negotiation, and replanning actions at runtime. The SCENE platform addresses the above issues by offering a language for composition design that extends the standard BPEL language with rules used to guide the execution of binding and re-binding self-reconfiguration operations.

Cavallaro and Di Nitto (2008) have addressed the problem of invoking services having an interface or protocol different from those originally expected by the service requester. They have identified a number of possible mismatches between services and some basic mapping functions that can be used to solve simple mismatches. Such mapping functions can be combined in a script to solve complex mismatches. Scripts can be executed by a mediator that receives an operation request, parses it, and eventually performs the needed adaptations. This approach is implemented as an extension of the SCENE framework. Originally SCENE was based on the hypothesis that all concrete services would show an identical interface or protocol. These researchers overcome this limitation by introducing an adapter in the framework.

Towards adaptive service coordination

Based on the literature discussed above, we conclude that current approaches for the adaptability of service-based systems mainly focus on the service infrastructure layer or specific aspects of service composition (Di Nitto et al., 2008). Infrastructural solutions towards adaptability often provide support for the dynamic run-time selection of candidate component services (Ardagna et al., 2007; Ardagna & Pernici, 2007; Verma et al., 2005). Approaches covering adaptability at the service composition and coordination layer mostly facilitate run time service (re)binding (Colombo et al., 2006; Verma et al., 2005) or the mediation of interface differences or protocol mismatches between invoked services (Brogi & Popescu, 2006; Cavallaro & Di Nitto, 2008; Denaro et al., 2006; Williams et al., 2006). These approaches can be considered as necessary but not sufficient for the construction of a truly dynamic and adaptive service-based system. Approaches covering higher-level aspects of the service composition and coordination layer and the business process management layer are still missing, which hampers the development of an integrated approach to adaptation that spans the concerns in all functional layers (Di Nitto et al., 2008).

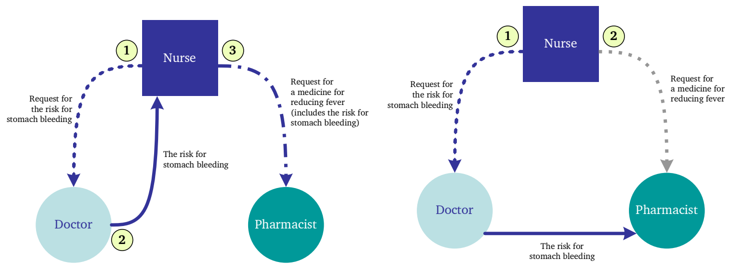
Another key point of difference is that quite some work can also be characterized as taking the perspective of an individual service. Examples of such research focus on resolving incompatibilities between collaborating services at different levels: syntactical (data part of messages), behavioral (sequences of messages) or semantic (resolving differences in meaning of data). So, this type of research is mainly concerned with the question of how a service can adapt itself to changes in the environment in which it is running, including changes in the interfaces of its collaborating services.

Therefore, in the future more research is needed from the perspective of the service system, tackling the question how service coordination can be adapted at runtime. We can derive two important research questions from these conclusions:

1. **How should coordination be organized such that an easy transition can be made from an as-is situation to a to-be situation, based on identified changes in the context?**
   Assuming that a service-based system is more adaptable if less component services need to be adapted, it can be of major importance that the initial coordination scenario is constructed such that potential context changes can be dealt with in the most efficient way. Therefore, identifying the relationships between existing coordination patterns (e.g. the patterns presented in this thesis) and considering several what-if situations, forms a first step in the path to self-adaptive service coordination.

   Example Consider the switch from a central data flow to a decentral data flow as shown in Figure 7.8[1] (i.e. switch from the scenario shown in Figure 7.8(a) to the scenario shown in Figure 7.8(b)). Such a switch requires the modification of the interface of the pharmacist. In the first scenario (Figure 7.8(a)), the pharmacist can accept one message with combined information on the required medicine including risk information. In the second scenario, the

[1]A complete description of this example can be found in Section 4.1 in Chapter 4


1
Nurse
3
Request for the risk for stomach bleeding
Request for a medicine for reducing fever (includes the risk for stomach bleeding)
The risk for stomach bleeding
Doctor
2
Pharmacist
1
Nurse
2
Request for the risk for stomach bleeding
Request for a medicine for reducing fever
Doctor
Pharmacist
The risk for stomach bleeding

(a) Central data flow (b) Decentral data flow

Figure 7.8: Two possible data flows for the hospital example

pharmacist needs to be capable to receive this information in two separate steps: one step for the risk information, and one step for the required medicine. A switch from the second scenario (Figure 7.8(b)) to the first scenario (Figure 7.8(a)) can however be realized without modifying the pharmacist's interface as it suffices to send two consecutive messages to realize the third step in the first scenario (Figure 7.8(a)).

2. **How can a new coordination be made effective at run-time?** Whereas at design time a developer can freely choose between alternative coordination scenarios (i.e. using the patterns and pattern combinations as presented in this thesis), the service infrastructure environment will pose restrictions on feasible run-time transitions from one scenario to another. Therefore, in order to address the second research question (how can a new coordination be made effective at run-time), the second aspect that needs to be investigated are the conditions that should be met to enable run-time adaptation. We therefore need to assess the feasibility of the patterns presented in this thesis and the patterns that are the result of answers to the previous research question in relation to available adaptation technology in the service infrastructure and service composition layer. The available technology may pose further restrictions on feasible adaptations of service coordination, leading to further refinement of the patterns.

   Example Suppose for example that the service infrastructure layer enables the rebinding of services, provided they have the same interface definitions. In such a case, a coordination can be adapted by redirecting for example a message from one party to another, provided the second one offers the same services as the first one.

In the above example, switching from the scenario shown in Figure 7.8(a) to the scenario in Figure 7.8(b) requires for example rebinding such that the message with risk information goes from the doctor to the pharmacists (as in Figure 7.8(b)) instead of to the nurse (as in Figure 7.8(a)).

A
Prolog program for completeness confirmation

```
% defining participants in a coordination scenario
participant(service_requester).
participant(service_provider).
participant(data_provider).

% defining two types of messages
message(data_request).
message(data).

% defining a message exchange between two participants
message_exchange(Participant1,Participant2,Message) :-
   participant(Participant1),
   participant(Participant2),
   Participant1\=Participant2,
   message(Message).

% C1.1: The service provider can only send data requests or receive
     data
possible_message_exchange(service_provider,Y,data_request) :-
   message_exchange(service_provider,Y,data_request).
possible_message_exchange(X,service_provider,data) :-
   message_exchange(X,service_provider,data).

% C1.2: The data provider can only receive data requests or send
     data
possible_message_exchange(data_provider,Y,data) :-
   message_exchange(data_provider,Y,data).
possible_message_exchange(X,data_provider,data_request) :-
   message_exchange(X,data_provider,data_request).

coordination_scenario(CoordinationScenario) :-
```

```
  % C1: A coordination scenario needs to be a proper interaction
      scenario
  findall(X, interaction_scenario(X),ListOfInteractionScenarios),
  member(CoordinationScenario,ListOfInteractionScenarios),
  % C2: the service provider must receive data from another entity
  member((_,service_provider,data),CoordinationScenario),
  % C3: the data provider must send data to an entity
  member((data_provider,_,data),CoordinationScenario),
  % C4: The resulting data flow must be complete
  complete_data_flow(CoordinationScenario),
  % requests or data can only be sent once per participant
  not(multiple_requests_or_data_sent(CoordinationScenario)),
  % requests or data can only be received once per participant
  not(multiple_requests_or_data_received(CoordinationScenario)).

interaction_scenario(MessageExchanges) :-
  setof((Participant1,Participant2,Message),
      possible_message_exchange(Participant1,Participant2,Message),L
      ),
  sublist(MessageExchanges,L).

% C4: The data flow is complete
% Specification as the negation of not C4.1 or not C4.2
complete_data_flow(Coordination_messages) :-
  not(incomplete_data_flow(Coordination_messages)).

% C4.1: The service requester must forward any data request to the
    data provider.
% Specification of a coordination in which C4.1 is not true
incomplete_data_flow(Coordination_messages) :-
  member((service_provider,service_requester,data_request),
      Coordination_messages),
  not(member((service_requester,data_provider,data_request),
      Coordination_messages)).

% C4.2: An entity can only send data if this entity is the data
    provider or has received data from another entity.
% Specification of a coordination in which C4.2 is not true
incomplete_data_flow(Coordination_messages) :-
  member((Participant1,Participant2,data),Coordination_messages),
  not(member((_,Participant1,data),Coordination_messages)),
  Participant1\=data_provider.

multiple_requests_or_data_sent(Coordination_messages) :-
  member((Participant1,Participant2a,Message),Coordination_messages)
      ,
  member((Participant1,Participant2b,Message),Coordination_messages)
      ,
  Participant2a\=Participant2b.

multiple_requests_or_data_received(Coordination_messages) :-
  member((Participant1a,Participant2,Message),Coordination_messages)
      ,
  member((Participant1b ,Participant2 ,Message) ,Coordination_messages)
      ,
  Participant1a\=Participant1b.

sublist([], _).

sublist([A|B],[C|D]) :-
  (
    A = C,
    sublist(B,D)
    ;
    sublist([A|B],D)
    ).
```

Listing A.1: Prolog program completeness confirmation

B
XML Schemas

B.1 XML Schema for a data dependencies model

```
<schema targetNamespace="http://servicecoordination.org/
     dataDependencies"
   elementFormDefault="qualified" xmlns="http://www.w3.org/2001/
       XMLSchema" xmlns:dd="http://servicecoordination.org/
       dataDependencies">

    <element name="dataDependenciesModel" type="
         dd:tDataDependenciesModel"></element>

    <complexType name="tDataDependenciesModel">
      <all>
        <element name="dataProviders" type="dd:tDataProviders"/>
        <element name="dataDependencies" type="dd:tDataDependencies"
             />
        <element name="dataAssociations" type="dd:tDataAssociations"
             />
      </all>
      <attribute name="bpmnLocation" type="string" use="required"></
           attribute>
    </complexType>

    <complexType name="tDataDependencies">
        <sequence>
            <element name="decisionDataDependency" type="
                 dd:tDecisionDataDependency" minOccurs="0" maxOccurs=
                 "unbounded"/>
            <element name="externalDataDependency" type="
                 dd:tExternalDataDependency" minOccurs="0" maxOccurs=
                 "unbounded"/>
        </sequence>
```

```
  </complexType>

  <complexType name="tDataAssociations">
    <sequence>
      <element name="decisionDataAssociation" type="
          dd:tDecisionDataAssociation" minOccurs="0" maxOccurs="
          unbounded"/>
      <element name="dataOutputToDataInputAssociation" type="
          dd:tDataOutputToDataInputAssociation" minOccurs="0"
          maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <complexType name="tExternalDataDependency">
    <complexContent>
      <extension base="dd:tDataDependency">
        <attribute name="dataInputAssociationId" type="NCName" use
            ="optional"></attribute>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tDecisionDataDependency">
    <complexContent>
      <extension base="dd:tDataDependency"></extension>
    </complexContent>
  </complexType>

  <complexType name="tDataDependency">
    <attribute name="id" type="ID" use="required"/>
      <attribute name="name" type="string" use="required"/>
      <attribute name="dataProviderId" type="NCName" use="required
          "/>
    <attribute name="bpmnTaskId" type="NCName" use="required"/>
  </complexType>

  <complexType name="tDataProviders">
    <sequence>
      <element name="dataProvider" type="dd:tDataProvider"
          minOccurs="1" maxOccurs="unbounded"></element>
    </sequence>
  </complexType>

  <complexType name="tDataProvider">
    <attribute name="id" type="ID" use="required"/>
    <attribute name="name" type="string" use="required"/>
    <attribute name="wsdlLocation" type="string" use="required"/>
    <attribute name="requestOperationName" type="string" use="
        required"/>
    <attribute name="requestRoleName" type="string" use="required"
        />
    <attribute name="receiveOperationName" type="string" use="
        required"/>
        <attribute name="partnerLinkTypeName" type="string" use="
            required"/>
        <attribute name="receiveRoleName" type="string" use="
            required"/>
    </complexType>


  <complexType name="tDecisionDataAssociation">
        <attribute name="id" type="ID" use="required"/>
        <attribute name="sourceRef" type="NCName" use="required"/>
      <attribute name="targetRef" type="NCName" use="required"/>
    </complexType>


    <complexType name="tDataOutputToDataInputAssociation">
      <attribute name="id" type="ID" use="required"></attribute>
        <attribute name="dataOutputAssociationId" type="NCName" use=
            "required"></attribute>
        <attribute name="dataInputAssociationId" type="NCName" use="
            required"></attribute>
    </complexType>
</schema>
```

Listing B.1: XML Schema for a data dependencies model

B.2 XML Schema for a coordination model

```
<schema
  targetNamespace="http://servicecoordination.org/coordinationModel"
  elementFormDefault="qualified" xmlns="http://www.w3.org/2001/
      XMLSchema"
  xmlns:cm="http://servicecoordination.org/coordinationModel"
      xmlns:dd="http://servicecoordination.org/dataDependencies">

  <element name="coordinationModel" type="cm:tCoordinationModel"></
      element>

  <complexType name="tCoordinationModel">
    <sequence>
      <element name="sequenceDependenciesManagement"
        type="cm:tSequenceDependenciesManagement">
      </element>
      <element name="dataDependenciesManagement"
        type="cm:tDataDependenciesManagement">
      </element>
    </sequence>
  </complexType>

  <complexType name="tSequenceDependenciesManagement">
    <sequence minOccurs="1" maxOccurs="1">
      <element name="coordinator" type="cm:tCoordinator" minOccurs="
          1" maxOccurs="unbounded"></element>
    </sequence>
    <attribute name="bpmnModel" type="string" use="required"></
         attribute>
  </complexType>

  <complexType name="tCoordinator">
    <sequence minOccurs="1" maxOccurs="1">
      <element name="bpmnTaskId" type="NCName" minOccurs="1"
        maxOccurs="unbounded">
      </element>
    </sequence>
    <attribute name="name" type="string" use="required"></attribute>
  </complexType>


  <complexType name="tDataDependenciesManagement">
        <sequence>
          <element name="dataDependencyManagement" type="
               cm:tDataDependencyManagement" minOccurs="0" maxOccurs=
               "unbounded"/>
          <element name="dataOutputIsDataInputManagement" type="
               cm:tDataOutputIsInputManagement" minOccurs="0"
               maxOccurs="unbounded"/>
        </sequence>
    <attribute name="dataDependenciesModel" type="string" use="
         required"/>
  </complexType>

  <complexType name="tDataDependencyManagement">
  <attribute name="id" type="ID"/>
  <attribute name="dataDependencyId" type="NCName"/>
  <attribute name="pattern" type="cm:dataDependencyManagementPattern
       "/>
</complexType>

  <simpleType name="dataDependencyManagementPattern">
    <restriction base="string">
      <enumeration
        value="active_service_provider_with_
␣␣␣␣␣␣␣␣direct_request_and_direct_data_transmission" />
      <enumeration
        value="active_service_provider_with_
␣␣␣␣␣␣␣␣direct_request_and_indirect_data_transmission" />
      <enumeration
        value="active_service_provider_with_
␣␣␣␣␣␣␣␣indirect_request_and_direct_data_transmission" />
      <enumeration
        value="active_service_provider_with_
␣␣␣␣␣␣␣␣indirect_request_and_indirect_data_transmission" />
      <enumeration
        value="active_service_requester_with_
␣␣␣␣␣␣␣␣direct_data_transmission" />
      <enumeration
```

```
        value="active_service_requester_with_
␣␣␣␣␣␣␣␣indirect_data_transmission" />
    </restriction>
  </simpleType>



  <complexType name="tCoordinator">
    <sequence minOccurs="1" maxOccurs="1">
      <element name="bpmnTaskId" type="NCName" minOccurs="1"
        maxOccurs="unbounded">
      </element>
    </sequence>
    <attribute name="name" type="string" use="required"></attribute>
  </complexType>

  <complexType name="tDataOutputIsInputManagement">
      <attribute name="id" type="ID" use="required"/>
        <attribute name="outputAssociationId" type="NCName" use="
            required"/>
        <attribute name="inputAssociationId" type="NCName" use="
            required"/>
      <attribute name="pattern" type="
          cm:dataOutputIsInputManagementPattern" use="required"/>
    </complexType>

    <simpleType name="dataOutputIsInputManagementPattern">
    <restriction base="string">
      <enumeration
        value="direct_data_transmission" />
      <enumeration
        value="indirect_data_transmission" />
    </restriction>
  </simpleType>
</schema>
```

Listing B.2: XML Schema for a coordination model

List of Figures

1.1 DSRM Process Model 7

1.2 The relationship between our research goal, research questions and research objectives 8

2.1 Basic concepts in service compositions 20

2.2 Two ways of coordinating the pharmacist and doctor 27

2.3 Two possible data flows for hospital example 28

2.4 An alternative data flow for the scenarios represented in figures 2.3(a) and 2.3(b) 31

2.5 Two ways of modeling that the output data of a certain task is the input for another task in BPMN 34

2.6 Modeling external data as input data in BPMN 35

2.7 Modeling that data objects are required to make a decision in BPMN 36

2.8 An undefined way of modeling that data objects are required to make a decision in BPMN 36

3.1 A travel agency business process 39

3.2 Independent coordinator managing all sequence dependencies 41

3.3 Customer Service managing all sequence dependencies 42

3.4 Decentralized coordination (1) 43

3.5 Decentralized coordination (2) 44

3.6 An overview of the pattern language 48

3.7 A legend for the simplified pattern representation 49

3.8 A BPMN representation of a CONTROLLED SERVICE PROVIDER . 50

3.9 A simplified representation of a CONTROLLED SERVICE 50

3.10 BPMN representation of the Finance Service as a CONTROLLED SERVICE PROVIDER 52

3.11 A simplified representation of the Finance Service as a CONTROLLED SERVICE PROVIDER 52

3.12 A BPMN representation of a CONTROLLED SERVICE PROVIDER controlled by an INDEPENDENT CONTROLLER 54

3.13 A simplified representation of a CONTROLLED SERVICE controlled by an INDEPENDENT CONTROLLER 54

3.14 A BPMN representation of a CONTROLLED SERVICE PROVIDER controlled by a CONTROLLING SERVICE PROVIDER 56

3.15 A simplified representation of a CONTROLLED SERVICE controlled by a CONTROLLING SERVICE PROVIDER 57

3.16 A BPMN representation of the Finance Service controlling the Online Payment Service 58

3.17 A simplified representation of a the Finance Service controlling the Online Payment Service 59

3.18 A BPMN representation of a CONTROLLED SERVICE PROVIDER controlled by a COORDINATOR 60

3.19 A simplified representation of a CONTROLLED SERVICE controlled by a COORDINATOR 60

3.20 A BPMN representation of a CONTROLLED SERVICE PROVIDER controlled by a COORDINATOR 62

3.21 A simplified representation of a CONTROLLED SERVICE controlled by a COORDINATOR 62

3.22 A BPMN representation of a SELF-CONTROLLED SERVICE PROVIDER 63

3.23 A simplified representation of a SELF-CONTROLLED SERVICE . . 63

3.24 A BPMN representation of a SELF-CONTROLLED SERVICE PROVIDER 65

3.25 A simplified representation of a SELF-CONTROLLED SERVICE . . 65

3.26 PARALLEL SPLIT pattern represented in BPMN 70

3.27 Coordinated parallel split using a SELF-CONTROLLED (BEFORE-SPLIT) SERVICE and CONTROLLED (AFTER-SPLIT) SERVICES controlled by a COORDINATING (BEFORE-SPLIT) SERVICE 71

3.28 SYNCHRONIZATION pattern represented in BPMN 73

4.1 A business process for taking care of patients with high fever . 80

4.2 Three questions that need to be answered by a specific coordination scenario that manages a data dependency 83
4.3 Relationships between the three patterns 87
4.4 DATA FLOW INITIATION 88
4.5 DIRECT REQUEST versus INDIRECT REQUEST 92
4.6 DIRECT DATA TRANSMISSION versus INDIRECT DATA TRANSMISSION 95
4.7 ACTIVE SERVICE PROVIDER with DIRECT REQUEST and DIRECT DATA TRANSMISSION 98
4.8 ACTIVE SERVICE PROVIDER with DIRECT REQUEST and INDIRECT DATA TRANSMISSION 99
4.9 ACTIVE SERVICE PROVIDER with INDIRECT REQUEST and INDIRECT DATA TRANSMISSION 99
4.10 ACTIVE SERVICE PROVIDER with INDIRECT REQUEST and DIRECT DATA TRANSMISSION 100
4.11 ACTIVE SERVICE REQUESTER with INDIRECT DATA TRANSMISSION100
4.12 ACTIVE SERVICE REQUESTER with DIRECT DATA TRANSMISSION 101
4.13 ACTIVE DATA PROVIDER with DIRECT DATA TRANSMISSION . . . 101
4.14 Active Data Provider with INDIRECT DATA TRANSMISSION . . . 102
4.15 Decision tree DATA FLOW INITIATION 103
4.16 Decision tree DATA REQUEST 104
4.17 Decision tree DATA TRANSMISSION 105
4.18 ACTIVE PHARMACIST with INDIRECT REQUEST and DIRECT DATA TRANSMISSION 106
4.19 Interaction procedure 114
4.20 Two coordination scenarios visualized as implementation patterns 115

5.1 Managing an Output Input Association using an ACTIVE DATA PROVIDER (i.e. Service Provider actively sends data) with a DIRECT DATA TRANSMISSION 124
5.2 Managing an Output Input Association using an ACTIVE DATA PROVIDER (i.e. Service Provider actively sends data) with an INDIRECT DATA TRANSMISSION 125

5.3 Managing a Decision Data Dependency 125
5.4 Managing a Decision Data Association 126
5.5 The use of the COORDINATOR pattern combined with two Decision Data Associations that are related to the same task data output 127
5.6 The use of the COORDINATOR pattern combined with two Output Input Associations that are related to the same task data output 128
5.7 The use of the COORDINATOR pattern combined with an Output Input Association and a Decision Data Association that are related to the same task data output 129

6.1 Modeling Decision Data Associations in BPMN 134
6.2 Input and output for pattern-based coordination 140
6.3 A UML class diagram showing the core of our tool 141
6.4 A BPMN model for the extended travel agency example 151
6.5 A visual representation of the BPEL process for the COORDINATING CUSTOMER SERVICE 158
6.6 A visual representation of the BPEL process for the service provider supporting the 'process customer request' task 159
6.7 A visual representation of the BPEL process for the booking COORDINATOR 161
6.8 A visual representation of the BPEL process for the service provider supporting the 'book flight' task 162

7.1 The relationship between the solution and our research goal, research questions and research objectives 166
7.2 Modeling preconditions in a SEQUENCE 171
7.3 BPMN representation of transaction consisting of hotel, car rental and flight booking tasks and corresponding compensating tasks 173
7.4 Simplified representation of complex transaction coordination 174
7.5 Simplified representation of simple (optimized) transaction coordination 174

7.6 BPMN representation of complex transaction coordination . . 175

7.7 BPMN representation of simple (optimized) transaction coordination 176

7.8 Two possible data flows for the hospital example 182

List of Tables

3.1 Combining the patterns to coordinate a SEQUENCE 69

3.2 Combining requests and event notifications in service compositions using the patterns 75

4.1 Summary of the consequences of DATA FLOW INITIATION . . . 91

4.2 Summary of the consequences of DIRECT-INDIRECT REQUEST . 94

4.3 Summary of the consequences of DIRECT-INDIRECT DATA TRANSMISSION 97

5.1 Four types of data dependencies in service compositions . . . 123

List of Code Listings

4.1 The coordination_scenario predicate 117

4.2 Restricting possible message exchanges for a Service Provider and a Data Provider 117

4.3 The complete_data_flow predicate 118

4.4 Querying the Prolog program 118

6.1 The complex type for data dependencies 135

6.2 The complex type for a data provider 136

6.3 The complex type for a Decision Data Association 136

6.4 The complex type for sequence dependencies management . . 137

6.5 The complex type for a COORDINATOR 137

6.6 The complex type for data dependencies management 137

6.7 The complex type for Data Dependency management 138

6.8 The simple type that defines the different coordination scenarios for managing an External Data Dependency 138

6.9 The complex type for Data Output Input Association management 139

6.10 The simple type that defines the different coordination scenarios for managing a Data Output Input Association 139

6.11 The use of XQuery in Java to find flow elements that target a certain BPMN *task* element 142

6.12 An XML-based data dependencies model for the travel agency example 153

6.13 An XML-based coordination model for the travel agency example155

A.1 Prolog program completeness confirmation 185

B.1 XML Schema for a data dependencies model 189

B.2 XML Schema for a coordination model 191

Bibliography

Alexander, C. (1979). *The timeless way of building*. New York, USA: Oxford University Press.

Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004). *Web services: concepts, architectures and applications*. New York, NY, USA: Springer-Verlag Berlin Heidelberg.

Andersson, B., Bider, I., Johannesson, P., & Perjons, E. (2005). Towards a formal definition of goal-oriented business process patterns. *Business Process Management Journal*, *11*(6), 650–662.

Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., & Plebani, P. (2007). PAWS: A Framework for Executing Adaptive Web-Service Processes. *IEEE Software*, *24*(6), 39–46.

Ardagna, D., & Pernici, B. (2007). Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, *33*(6), 369–384.

Avgeriou, P., & Zdun, U. (2005). Architectural patterns revisited a pattern language. In *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPloP 2005)* (pp. 1–39).

Balasooriya, J., Padhye, M., Prasad, S. K., & Navathe, S. B. (2005). Bondflow: A system for distributed coordination of workflows over web services. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005) - Workshop 1* (p. 121.1). Washington, DC, USA: IEEE Computer Society.

Bandara, W., Indulska, M., Sadiq, S., Chong, S., Rosemann, M., & Green, P. (2007). Major issues in Business Process Management: an Expert Perspective. In *Proceedings of the 15th European Conference on Information Systems (ECIS 2007)*.

Barker, A., Weissman, J. B., & Hemert, J. I. (2009). The circulate architecture: avoiding workflow bottlenecks caused by centralised orchestration. *Cluster Computing*, *12*(2), 221–235.

Barker, A., Weissman, J. B., & Van Hemert, J. (2008a). Eliminating the middleman: peer-to-peer dataflow. In *Proceedings of the 17th international symposium on High performance distributed computing (HPDC 2008)* (pp. 55–64). New York, NY, USA: ACM.

Barker, A., Weissman, J. B., & Van Hemert, J. (2008b). Orchestrating data-centric workflows. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2008)* (pp. 210–217). Washington, DC, USA: IEEE Computer Society.

Barros, A., & Börger, E. (2005). A compositional framework for service interaction patterns and interaction flows. In K.-K. Lau & R. Banach

(Eds.), *Formal methods and software engineering* (Vol. 3785, p. 5-35). Springer-Verlag Berlin Heidelberg.

Barros, A., Dumas, M., & Hofstede, A. H. ter. (2005). Service Interaction Patterns. In W. M. P. Van der Aalst, B. Benatallah, F. Casati, & F. Curbera (Eds.), *Business Process Management* (Vol. 3649, p. 302-318). Springer-Verlag Berlin Heidelberg.

Barros, A., Dumas, M., & Oaks, P. (2006). Standards for web service choreography and orchestration: Status and perspectives. In C. Bussler & A. Haller (Eds.), *Business process management workshops* (Vol. 3812, pp. 61–74). Springer-Verlag Berlin Heidelberg.

Beek, M. ter, Bucchiarone, A., & Gnesi, S. (2006). *A survey on service composition approaches: From industrial standards to formal methods* (Tech. Rep. No. 2006-TR-15). Pisa, Italy: Consiglio Nazionale delle Ricerche.

Bellini, A., Prado, A. F. d., & Zaina, L. A. M. (2010). Top-down approach for web services development. In *Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services (ICIW 2010)* (pp. 426–431). Washington, DC, USA: IEEE Computer Society.

Benatallah, B., Dijkman, R., Dumas, M., & Maamar, Z. (2005). Service Composition: Concepts, Techniques, Tools and Trends. In *Service-oriented software system engineering: Challenges and practices* (pp. 48–66). IGI Publishing.

Benatallah, B., Sheng, Q., & Dumas, M. (2003). The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, *7*(1), 40–48.

Bhiri, S., Perrin, O., & Godart, C. (2005). Ensuring required failure atomicity of composite web services. In *Proceedings of the 14th international conference on World Wide Web (WWW 2005)* (pp. 138–147). New York, NY, USA: ACM.

Bhiri, S., Perrin, O., & Godart, C. (2006). Extending workflow patterns with transactional dependencies to define reliable composite web services. In *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT-ICIW 2006)* (p. 145). Washington, DC, USA: IEEE Computer Society.

Binder, W., Constantinescu, I., & Faltings, B. (2006). Decentralized orchestration of composite web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2006)* (pp. 869–876). Washington, DC, USA: IEEE Computer Society.

Brahe, S. (2007). Bpm on top of soa: Experiences from the financial industry. In G. Alonso, P. Dadam, & M. Rosemann (Eds.), *Business Process Management* (Vol. 4714, pp. 96–111). Springer-Verlag Berlin Heidelberg.

Brogi, A., & Popescu, R. (2006). Automated generation of bpel adapters. In A. Dan & W. Lamersdorf (Eds.), *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC 2006)* (Vol. 4294, pp. 27–39). Springer-Verlag Berlin Heidelberg.

Buschmann, F., Henney, K., & Schmidt, D. (2007). *Pattern-oriented software architecture: On patterns and pattern languages*. The Atrium, Southern Gate, Chichester, West Sussex, England: John Wiley & Sons Ltd.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture: A system of patterns*. The Atrium, Southern Gate, Chichester, West Sussex, England: John Wiley & Sons Ltd.

Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., & Zavattaro, G. (2005). Towards a formal framework for choreography. In *Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE 2005)* (pp. 107–112). Washington, DC, USA: IEEE Computer Society.

Cardoso, J. (2007). Complexity analysis of BPEL Web processes. *Software Process: Improvement and Practice, 12*(1), 35–49.

Cavallaro, L., & Di Nitto, E. (2008). An approach to adapt service requests to actual service interfaces. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems (SEAMS 2008)* (pp. 129–136). New York, NY, USA: ACM.

Chang, Y.-C., Mazzoleni, P., Mihaila, G. A., & Cohn, D. (2008). Solving the service composition puzzle. In *Proceedings of the 2008 IEEE International Conference on Services Computing (SCC 2008)* (pp. 387–394). Washington, DC, USA: IEEE Computer Society.

Charfi, A., & Mezini, M. (2007). AO4BPEL: An Aspect-oriented Extension to BPEL. *World Wide Web, 10*(3), 309–344.

Cherbakov, L., Galambos, G., Harishankar, R., Kalyana, S., & Rackham, G. (2005). Impact of service orientation at the business level. *IBM Systems Journal, 44*(4), 653–668.

Clocksin, W., & Mellish, C. (2003). *Programming in PROLOG (Fifth Edition)*. New York, NY, USA: Springer-Verlag Berlin Heidelberg.

Colombo, M., Di Nitto, E., & Mauri, M. (2006). SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In A. Dan & W. Lamersdorf (Eds.), *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC 2006)* (Vol. 4294, pp. 191–202). Springer-Verlag Berlin Heidelberg.

Cumberlidge, M. (2007). *Business Process Management with JBoss jBPM*. Birmingham, United Kingdom: Packt Publishing.

Decker, G. (2009). *Design and analysis of process choreographies*. Unpublished doctoral dissertation, Business Process Technology Group, Hasso

Plattner Institute, University of Potsdam, Germany.

Decker, G., & Barros, A. (2008). Interaction modeling using bpmn. In *Proceedings of the 2007 international conference on Business Process Management (BPM 2007)* (pp. 208–219). Springer-Verlag Berlin Heidelberg.

Decker, G., Kopp, O., Leymann, F., & Weske, M. (2007). BPEL4Chor: Extending BPEL for Modeling Choreographies. In *Proceedings of IEEE 2007 International Conference on Web Services (ICWS 2007)* (p. 296 -303). Washington, DC, USA: IEEE Computer Society.

Decker, G., Kopp, O., Leymann, F., & Weske, M. (2009). Interacting services: From specification to execution. *Data & Knowledge Engineering, 68*(10), 946–972.

Decker, G., Overdick, H., & Weske, M. (2008). Oryx - an open modeling platform for the bpm community. In M. Dumas, M. Reichert, & M.-C. Shan (Eds.), *Business process management* (Vol. 5240, pp. 382–385). Springer-Verlag Berlin Heidelberg.

Decker, G., Overdick, H., & Zaha, J. (2006). On the Suitability of WS-CDL for Choreography Modeling. In *Proceedings of Methoden, Konzepte und Technologien für die Entwicklung von dienstebasierten Informationssystemen (EMISA 2006).* Citeseer.

Decker, G., & Puhlmann, F. (2007). Extending bpmn for modeling complex choreographies. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems (OTM 2007)* (pp. 24–40). Springer-Verlag Berlin Heidelberg.

Denaro, G., Pezzé, M., Tosi, D., & Schilling, D. (2006). Towards self-adaptive service-oriented architectures. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications (TAV-WEB 2006)* (pp. 10–16). New York, NY, USA: ACM.

DeRemer, F., & Kron, H. (1975). Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on reliable software* (pp. 114–121). New York, NY, USA: ACM.

Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., & Pohl, K. (2008). A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering, 15*(3-4), 313–341.

Dreiling, A., Rosemann, M., Van der Aalst, W. M. P., & Sadiq, W. (2008). From conceptual process models to running systems: A holistic approach for the configuration of enterprise system processes. *Decision Support Systems, 45*(2), 189–207.

Dustdar, S., & Schreiner, W. (2005). A survey on web services composition. *International Journal of Web and Grid Services, 1*(1), 1–30.

Edwards, M. (2007, March 15th). *Relationship between SCA and BPEL.* SCA BPEL White Paper. Available from `http://www.osoa.org/display/Main/SCA+BPEL+White+Paper`

Emig, C., Momm, C., Weisser, J., & Abeck, S. (2005). Programming in the Large based on the Business Process Modeling Notation. *Lecture Notes in Informatics (LNI)*, *68*, 627–631.

Erl, T. (2005). *Service-oriented architecture: Concepts, technology, and design*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Erl, T. (2007). *SOA Principles of Service Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Erl, T. (2009). *SOA Design Patterns*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, *35*(2), 114–131.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., et al. (1999, June). *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Available from `http://www.w3.org/Protocols/rfc2616/rfc2616.txt`

Fjellheim, T., Milliner, S., Dumas, M., & Vayssière, J. (2007). A process-based methodology for designing event-based mobile composite applications. *Data & Knowledge Engineering*, *61*(1), 6–22.

Gabriel, R. P. (2002). *Writer's Workshops and the Work of Making Things*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA.

Goethals, F. (2008). Important Issues for Evaluating Inter-Organizational Data Integration Configurations. *Electronic Journal Information Systems Evaluation*, *11*(3), 185–196.

Haesen, R. (2009). *Designing Information System Services in Information-Intensive Organisations*. Unpublished doctoral dissertation, Faculty of Business and Economics, Katholieke Universiteit Leuven.

Haesen, R., De Rore, L., Goedertier, S., Snoeck, M., Lemahieu, W., & Poelmans, S. (2007). Stateless process enactment. In *Proceedings of the 14th Conference on Pattern Languages of Programs (PLoP 2007)* (pp. 1–5). New York, NY, USA: ACM.

Haesen, R., De Rore, L., Snoeck, M., Lemahieu, W., & Poelmans, S. (2006). Active-passive hybrid data collection. In *Proceedings of the 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006)* (pp. 565–577).

Haesen, R., Snoeck, M., Lemahieu, W., & Poelmans, S. (2009). Existence dependency-based domain modeling for improving stateless process enactment. In *Proceedings of the 2009 Congress on Services - I (SERVICES 2009)* (pp. 515–521). Washington, DC, USA: IEEE Computer Society.

Hagel III, J., & Singer, M. (1999). Unbundling the Corporation. *Harvard*

Business Review, *77*(2), 133–141.

Henkel, M., Zdravkovic, J., & Johannesson, P. (2004). Service-based processes: design for business and technology. In *Proceedings of the 2nd international conference on Service oriented computing (ICSOC 2004)* (pp. 21–29). New York, NY, USA: ACM.

Hens, P., Snoeck, M., Poels, G., & De Backer, M. (2009). *The use of the concept of event in enterprise ontologies and requirements engineering literature* (Tech. Rep. No. KBI 0909). Leuven, Belgium: Faculty of Business and Economics, Katholieke Universiteit Leuven. Available from `http://www.econ.kuleuven.be/fetew/int_reports.aspx?group_id=17`

Hentrich, C., & Zdun, U. (2006). Patterns for process-oriented integration in service-oriented architectures. In *Proceedings of the 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006).*

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, *28*(1), 75–105.

Hewlett-Packard. (2003, July 16th). *Web Services Events (WS-Events) (version 2.0).*

Hohpe, G., & Woolf, B. (2003). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Iyengar, A., Jessani, V., & Chilanti, M. (2007). *WebSphere Business Integration Primer: Process Server, BPEL, SCA, and SOA*. IBM Press.

Janssen, M., & Feenstra, R. (2008). Socio-technical design of service compositions: a coordination view. In *Proceedings of the 2nd International Conference on Theory and Practice of Electronic Governance (ICEGOV 2008)* (pp. 323–330). New York, NY, USA: ACM.

Jellema, L., & Dikmans, L. (2010). *Oracle SOA Suite 11g Handbook*. New York, NY, USA: McGraw-Hill, Inc.

Jennings, F., & Salter, D. (2008). *Building SOA-Based Composite Applications Using NetBeans IDE 6*. Birmingham, United Kingdom: Packt Publishing.

Jones, S. (2005). Toward an acceptable definition of service. *IEEE Software*, *22*(3), 87–93.

Juric, M. B. (2006). *Business Process Execution Language for Web Services BPEL and BPEL4WS (2nd Edition)*. Birmingham, United Kingdom: Packt Publishing, Limited.

Juric, M. B. (2010). Wsdl and bpel extensions for event driven architecture. *Information and Software Technology*, *52*(10), 1023–1043.

Kappel, G., Rausch-Schott, S., & Retschitzegger, W. (1998). Coordination in workflow management systems - a rule-based approach. In *Coordination Technology for Collaborative Applications - Organizations, Processes, and Agents [ASIAN 1996 Workshop]* (pp. 99–120). London, UK: Springer-Verlag.

Kleppe, A. G., Warmer, J., & Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Ko, R. K. L., Lee, S. S. G., & Lee, E. W. (2009). Business process management (bpm) standards: a survey. *Business Process Management Journal*, *15*(5), 744–791.

Kohlborn, T., Korthaus, A., Chan, T., & Rosemann, M. (2009). Identification and Analysis of Business and Software Services – A Consolidated Approach. *IEEE Transactions on Services Computing*, *2*(1), 50–64.

Koubarakis, M., & Plexousakis, D. (1999). Business process modelling and design - a formal model and methodology. *BT Technology Journal*, *17*(4), 23–35.

Leymann, F. (2006). Workflow-based coordination and cooperation in a service world. In R. Meersman & Z. Tari (Eds.), *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE* (Vol. 4275, p. 2-16). Springer-Verlag Berlin Heidelberg.

Lin, F.-r., & Chang, H.-c. (2005). The development and evaluation of exception handling mechanisms for order fulfillment process based on bpel4ws. In *Proceedings of the 7th international conference on Electronic commerce (ICEC 2005)* (pp. 478–484). New York, NY, USA: ACM.

Liu, D., Law, K. H., & Wiederhold, G. (2002a). Analysis of integration models for service composition. In *Proceedings of the 3rd international workshop on Software and performance (WOSP 2002)* (pp. 158–165). New York, NY, USA: ACM.

Liu, D., Law, K. H., & Wiederhold, G. (2002b). Data-flow distribution in FICAS service composition infrastructure. In *Proceedings of the 15th International Conference on Parallel and Distributed Computing Systems (PDCS 2002)*. Louisville, Kentucky USA: ISCA.

Lu, R., & Sadiq, S. (2007). A survey of comparative business process modeling approaches. In *Proceedings of the 10th international conference on Business information systems (BIS 2007)* (pp. 82–94). Berlin, Heidelberg: Springer-Verlag.

Malinova, A., & Gocheva-Ilieva, S. (2008). Using the Business Process Execution Language for Managing Scientific Processes. *International Journal Information Technologies and Knowledge*, *2*(3), 257–261.

Malone, T., & Crowston, K. (1994). The interdisciplinary study of coordination. *ACM Computing Surveys (CSUR)*, *26*(1), 119.

Mcafee, A., Sjoman, A., & Dessain, V. (2004). *Zara: IT for fast fashion* (Case 9-604-081). Harvard Business School.

Metzger, A., & Pohl, K. (2009). Towards the Next Generation of Service-Based Systems: The S-Cube Research Framework. In P. van Eck, J. Gordijn, & R. Wieringa (Eds.), *Advanced Information Systems Engineering* (Vol.

5565, pp. 11–16). Springer-Verlag Berlin Heidelberg.

Mitra, S., Kumar, R., & Basu, S. (2008). Optimum decentralized choreography for web services composition. In *Proceedings of the 2008 IEEE International Conference on Services Computing (SCC 2008)* (pp. 395–402). Washington, DC, USA: IEEE Computer Society.

Møller, C., Maack, C., & Tan, R. (2008). What is business process management: A two stage literature review of an emerging field. In L. Xu, A. Tjoa, & S. Chaudhry (Eds.), *Research and Practical Issues of Enterprise Information Systems II Volume 1* (Vol. 254, pp. 19–31). Springer Boston.

Monsieur, G. (2008). Gestructureerd bedrijfsprocessen implementeren (structured business process implementation). *IT Professional*, *3*(43), 26–27.

Monsieur, G., De Rore, L., Snoeck, M., & Lemahieu, W. (2008). Handling transactional business services. In *Proceedings of the 15th Conference on Pattern Languages of Programs (PLoP 2008)*. New York, NY, USA: ACM.

Monsieur, G., Snoeck, M., & Lemahieu, W. (2007). Coordinated Web Services Orchestration. In *Proceedings of IEEE 2007 International Conference on Web Services (ICWS 2007)* (pp. 775–783). Washington, DC, USA: IEEE Computer Society.

Monsieur, G., Snoeck, M., & Lemahieu, W. (2009). A pattern language for service input data provisioning. In *Proceedings of the 16th Conference on Pattern Languages of Programs (PLoP 2009)*. New York, NY, USA: ACM.

Monsieur, G., Snoeck, M., & Lemahieu, W. (2010a). Managing data dependencies in service compositions [Submitted for review]. *IEEE Transactions on Software Engineering*.

Monsieur, G., Snoeck, M., & Lemahieu, W. (2010b). Managing sequence dependencies in service compositions. In *Proceedings of the 15th European Conference on Pattern Languages of Programs (EuroPLoP 2010)*.

Ng, T. H., Cheung, S. C., Chan, W. K., & Yu, Y. T. (2006). Work experience versus refactoring to design patterns: a controlled experiment. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)* (pp. 12–22). New York, NY, USA: ACM.

Niblett, P., & Graham, S. (2005). Events and service-oriented architecture: the OASIS Web services notification specifications. *IBM Systems Journal*, *44*(4), 869–886.

Nitzsche, J., Lessen, T. van, Karastoyanova, D., & Leymann, F. (2007). BPEL^{light}. In G. Alonso, P. Dadam, & M. Rosemann (Eds.), *Business process management* (Vol. 4714, pp. 214–229). Springer-Verlag Berlin Heidelberg.

OASIS. (2006a, October 12th). *Reference Model for Service Oriented Architecture 1.0*. OASIS Standard. Available from `http://docs.oasis-open.org/soa-rm/v1.0/`

OASIS. (2006b, October 1st). *Web Services Brokered Notification 1.3 (WS-BrokeredNotification).* OASIS Standard.

OASIS. (2007, April 11th). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0.* OASIS Standard. Available from `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`

OMG. (2010a, June). *Business Process Model and Notation (BPMN) Version 2.0.* OMG Document (dtc/2010-06-05). Available from `http://www.omg.org/spec/BPMN/2.0`

OMG. (2010b, May). *OMG Unified Modeling Language^{TM}(OMG UML), Superstructure Version 2.3.* OMG Document (formal/2010-05-05). Available from `http://www.omg.org/spec/UML/2.3/Superstructure`

Open SOA Collaboration. (2007, March 15th). *SCA Service Component Architecture - Assembly Model Specification.* Final Version 1.0 Specification. Available from `http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf?version=1`

Ouyang, C., Dumas, M., Hofstede, A. H. M. ter, & Van der Aalst, W. M. P. (2006). From bpmn process models to bpel web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2006)* (pp. 285–292). Washington, DC, USA: IEEE Computer Society.

Paikens, A., & Arnicans, G. (2008). Use of Design Patterns in PHP-Based Web Application Frameworks. *Scientific Papers University of Latvia, Computer Science and Information Technologies*, *733*, 53-71.

Papazoglou, M. (2003). Service-oriented computing: Concepts, characteristics and directions. *International Conference on Web Information Systems Engineering (WISE 2003)*.

Papazoglou, M. (2005). Extending the service-oriented architecture. *Business Integration Journal*, *7*(1), 18–21.

Papazoglou, M. (2007). *Web services: Principles and technology*. Harlow, Essex, England: Pearson Education Limited.

Papazoglou, M., Delis, A., Bouguettaya, A., & Haghjoo, M. (1997). Class library support for workflow environments and applications. *IEEE Transactions on Computers*, *46*(6), 673–686.

Papazoglou, M., Traverso, P., Dustdar, S., & Leymann, F. (2007). Service-Oriented Computing: State of the Art and Research Challenges. *Computer,* 38–45.

Papazoglou, M., & Van den Heuvel, W.-J. (2003). *Service-Oriented Computing: State-of-the-Art and Open Research Issues* (Tech. Rep. No. TI/RS/2003/123). Enschede, The Netherlands: Telematica Instituut (Novay). Available from `https://doc.novay.nl/dsweb/Get/Document-40060`

Papazoglou, M., & Van den Heuvel, W.-J. (2007a). Business process development life cycle methodology. *Communications of the ACM*, *50*(10), 79–85.

Papazoglou, M., & Van den Heuvel, W.-J. (2007b). Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal - The International Journal on Very Large Data Bases*, *16*(3), 415.

Parker, D. (2010). *Microsoft Visio 2010 Business Process Diagramming and Validation*. Birmingham, United Kingdom: Packt Publishing, Limited.

Paton, N. W., & Díaz, O. (1999). Active database systems. *ACM Computing Surveys*, *31*(1), 63–103.

Pedraza, G., & Estublier, J. (2009). Distributed Orchestration Versus Choreography: The FOCAS Approach. In *Proceedings of the International Conference on Software Process (ICSP 2009)* (pp. 75–86). Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg.

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45–77.

Peltz, C. (2003). Web services orchestration and choreography. *Computer*, *36*(10), 46–52.

Pessoa, R. M., Silva, E., Sinderen, M. v., Quartel, D. A. C., & Pires, L. F. (2008). Enterprise interoperability with soa: a survey of service composition approaches. In *Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops (EDOCW)* (pp. 238–251). Washington, DC, USA: IEEE Computer Society.

Poppendieck, M., & Poppendieck, T. (2006). *Implementing lean software development: From concept to cash (the addison-wesley signature series)*. Addison-Wesley Professional.

Prechelt, L., Unger, B., Tichy, W., Brössler, P., & Votta, L. (2001). A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 1134–1144.

Recker, J. (2008). BPMN Modeling - Who, Where, How and Why. *BPTrends*, *5*(3).

Recker, J., & Mendling, J. (2006). On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In T. Latour & M. Petit (Eds.), *Proceedings of the 18th International Conference on Advanced Information Systems Engineering. Proceedings of Workshops and Doctoral Consortiums* (pp. 521–532).

RosettaNet. (n.d.). *RosettaNet Partner Interface Processes©(PIPs©)*. RosettaNet Standard. Retrieved September 12th, 2010, from `http://www.rosettanet.org/dnn_rose/Standards/RosettaNetStandards/PIPs/tabid/475/Default.aspx`

Shaw, M., & Garlan, D. (1996). *Software architecture: Perspectives on an emerging discipline*. Upper Saddle River, NJ, USA: Prentice Hall.

Singh, M., Chopra, A., Desai, N., & Mallya, A. (2004). Protocols for processes:

programming in the large for open systems. *ACM SIGPLAN Notices, 39*(12), 73–83.

Snoeck, M., Lemahieu, W., Goethals, F., Dedene, G., & Vandenbulcke, J. (2004). Events as atomic contracts for component integration. *Data & Knowledge Engineering, 51*(1), 81–107.

Tabatabaei, S., Kadir, W., & Ibrahim, S. (2008). Web Service Composition Approaches to Support Dynamic E-Business Systems. *Communications of the IBIMA, 2*, 115–121.

Vaishnavi, V. K., & Kuechler, W., Jr. (2007). *Design science research methods and patterns: Innovating information and communication technology*. Boston, MA, USA: Auerbach Publications.

Van der Aalst, W., Dumas, M., Hofstede, A., Russell, N., Verbeek, H., & Wohed, P. (2005). Life After BPEL? In M. Bravetti, L. Kloul, & G. Zavattaro (Eds.), *Formal techniques for computer systems and business processes* (Vol. 3670, pp. 35–50). Springer-Verlag Berlin Heidelberg.

Van der Aalst, W., Ter Hofstede, A., Kiepuszewski, B., & Barros, A. (2003). Workflow patterns. *Distributed and parallel databases, 14*(1), 5–51.

Verma, K., Gomadam, K., Sheth, A. P., Miller, J. A., & Wu, Z. (2005, June). *The METEOR-S approach for configuring and executing dynamic web processes* (Tech. Rep.). LSDIS Lab, University of Georgia, Athens, Georgia.

W3C. (2000, May 8th). *Simple Object Access Protocol (SOAP) 1.1.* W3C Note. Available from `http://www.w3.org/TR/soap/`

W3C. (2001, March 15th). *Web Services Description Language (WSDL) Version 1.1.* W3C Note. Available from `http://www.w3.org/TR/wsdl`

W3C. (2004, October 24th). *XML Schema Part 0: Primer Second Edition.* W3C Recommendation. Available from `http://www.w3.org/TR/xmlschema-0/`

W3C. (2005, November 9th). *Web Services Choreography Description Language (WS-CDL) Version 1.0.* W3C Candidate Recommendation. Available from `http://www.w3.org/TR/ws-cdl-10/`

W3C. (2006, March 15th). *Web Services Eventing (WS-Eventing).* W3C Member Submission. Available from `http://www.w3.org/Submission/WS-Eventing/`

W3C. (2007a, June 26th). *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language.* W3C Recommendation. Available from `http://www.w3.org/TR/wsdl20/`

W3C. (2007b, January 23th). *XQuery 1.0: An XML Query Language.* W3C Recommendation. Available from `http://www.w3.org/TR/xquery/`

Weber, R., Schuler, C., Neukomm, P., Schuldt, H., & Schek, H.-J. (2003). Web service composition with O'GRAPE and OSIRIS. In *Proceedings of the 29th international conference on Very large data bases (VLDB 2003)* (pp. 1081–1084). VLDB Endowment.

Weigand, H., & Van den Heuvel, W.-J. (2002). Cross-organizational workflow integration using contracts. *Decision Support Systems*, *33*(3), 247–265.

White, S. (2005, March). Using BPMN to Model a BPEL Process. *BPTrends*.

Wielemaker, J. (2003). An overview of the SWI-Prolog programming environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments* (pp. 1–16). Leuven, Belgium: Department of Computer Science, K.U.Leuven.

Williams, S., Battle, S., & Cuadrado, J. (2006). Protocol mediation for adaptation in semantic web services. In Y. Sure & J. Domingue (Eds.), *The semantic web: Research and applications* (Vol. 4011, p. 635-649). Springer-Verlag Berlin Heidelberg.

Wohed, P., Van der Aalst, W., Dumas, M., Ter Hofstede, A., & Russell, N. (2005). Pattern-Based Analysis of the Control-Flow Perspective of UML Activity Diagrams. In L. Delcambre, C. Kop, H. Mayr, J. Mylopoulos, & O. Pastor (Eds.), *Conceptual Modeling ER 2005* (Vol. 3716, p. 63-78). Springer-Verlag Berlin Heidelberg.

Wohed, P., Van der Aalst, W., Dumas, M., Ter Hofstede, A., & Russell, N. (2006). On the Suitability of BPMN for Business Process Modelling. In S. Dustdar, J. Fiadeiro, & A. Sheth (Eds.), *Business Process Management* (Vol. 4102, p. 161-176). Springer-Verlag Berlin Heidelberg.

Yang, J. (2003). Web service componentization. *Communications of the ACM*, *46*(10), 35–40.

Yang, J., Papazoglou, M., & Van den Heuvel, W.-J. (2002). Tackling the Challenges of Service Composition in E-Marketplaces. In *Proceedings of the 12th International Workshop on Research Issues in Data Engineering: Engineering E-Commerce/E-Business Systems (RIDE 2002)* (p. 125). Washington, DC, USA: IEEE Computer Society.

Zdun, U., Hentrich, C., & Van der Aalst, W. M. (2006). A survey of patterns for service-oriented architectures. *International Journal of Internet Protocol Technology*, *1*(3), 132–143.

Zirpins, C., & Lamersdorf, W. (2004). Service Co-operation Patterns and their Customised Coordination. In *Proceedings of the Second European Workshop on Object Orientation and Web Service (EOOWS 2004)*.

Zirpins, C., Lamersdorf, W., & Baier, T. (2004). Flexible coordination of service interaction patterns. In *Proceedings of the 2nd international conference on Service oriented computing (ICSOC 2004)* (pp. 49–56). New York, NY, USA: ACM.

Doctoral dissertations from the faculty of business and economics

See http://www.kuleuven.ac.be/doctoraatsverdediging/archief.htm

Biography

Geert Monsieur was born on the 9th of April 1983 in Maaseik, Belgium. In July 2005 he received the degree of Master in Computer Science cum laude from the Katholieke Universiteit Leuven (K.U.Leuven).

In October 2005, Geert started working as a PhD student at the Leuven Institute for Research on Information Systems (LIRIS) of the K.U.Leuven, under supervision of prof. dr. Monique Snoeck, prof. dr. Wilfried Lemahieu and prof. dr. Guido Dedene. In December 2010 he obtained a PhD in Applied Economics from the Faculty of Business and Economics at K.U.Leuven. His main research interests include service-based systems, patterns and pattern languages, business process management and model-driven software development. As a PhD student he was involved in several research projects of companies. In 2009 he was a guest lecturer at the IESEG School of Management in Lille, France.