Vrije Universiteit Brussel

Faculteit Wetenschappen
Vakgroep Computerwetenschappen

Aspectgerichte samenwerking tussen workflows in nomadische netwerken

Proefschrift ingediend met het oog op het behalen van de graad van Master in de Ingenieurswetenschappen: Computerwetenschappen

Jeroen Van den haute

Promotor: Prof. Dr. Viviane Jonckers
Begeleiders: Dr. Eline Philips
Dr. Niels Joncheere

Juni 2013

Abstract

Processen worden samengesteld uit verschillende taken. Deze taak kan het invokeren van een service (bvb. webservices, zoals Google Maps) zijn, of het contacteren van een persoon die dan een taak uitvoert. Tegenwoordig beschikt iedereen over een mobiel toestel waardoor er meer services beschikbaar zijn die hierop kunnen worden uitgevoerd. Wanneer er een proces wordt gedefinieerd moet er dus van uitgegaan worden dat voor het uitvoeren van een taak mogelijk communicatie met een mobiel apparaat nodig is. Het gebruik van nomadische netwerken is alomtegenwoordig. Een nomadisch netwerk bestaat uit een groep van mobiele toestellen die verbonden is met een vaste infrastructuur. Wanneer we kijken naar luchthavens of grote winkelcentra is er steeds een vaste infrastructuur aanwezig waarop processen kunnen worden gedefinieerd en uitgevoerd. Deze processen worden voorgesteld door workflows, die op voorhand gedefinieerd zijn en dynamisch moeten kunnen samenwerken. Dit geldt zowel voor workflows die op dezelfde vaste infrastructuur zijn gedefinieerd, als voor workflows die op verschillende vaste infrastructuren zijn gedefinieerd. Onder samenwerking verstaan we ondermeer het uitwisselen van data tussen de verschillende workflows, een workflow die na het beëindigen van een taak een andere workflow start en het synchroniseren van verschillende workflows. Omdat de workflows worden uitgevoerd in nomadische netwerken moeten we ook steeds rekening houden met de volatiliteit van netwerkconnecties, wat een effect kan hebben op de samenwerking tussen workflows. NOW is een nomadische workflow taal die ons toelaat om workflows te definiëren in nomadische netwerken. Het is mogelijk om workflows apart van elkaar te definiëren, maar deze kunnen niet samenwerken. Indien we toch een samenwerking willen moeten we beide workflows aanpassen en combineren in één grote workflow. Wij willen een samenwerking tussen workflows bekomen door gebruik te maken van aspectgeoriënteerd programmeren. Hierbij definiëren we in een aspect de samenwerking tussen workflows. Pointcuts laten ons toe om verschillende joinpoints in workflows te selecteren, terwijl we in de advices definiëren hoe workflows moeten samenwerken. In workflows kunnen we enkele specifieke joinpoints identificeren, zoals taken, data en patronen. Omdat we ook steeds rekening houden met disconnecties in het nomadische netwerk is een disconnectie ook een joinpoint. Aspectgeoriënteerd programmeren laat ons toe om de samenwerking tussen workflows te definiëren zonder dat we hiervoor de workflows zelf moeten aanpassen, wat ons een verhoogde mate van scheiding van concerns geeft. Omdat we ook niet op voorhand weten welke workflows er allemaal worden uitgevoerd in de omgeving is het onmogelijk om deze op voorhand aan te passen zodat er een samenwerking mogelijk is. Een aspect wordt toegepast op alle workflows in de omgeving, ook diegene die er nieuw bijkomen. We hebben gezien dat, door de code complexiteit van de implementaties van twee voorbeeldtoepassingen te vergelijken in NOW en in de aspectgeoriënteerde versie van NOW, er een veel uitgebreidere en eenvoudigere samenwerking tussen workflows mogelijk is wanneer we gebruik maken van aspecten.

Dankwoord

Mijn oprechte dank gaat uit naar Prof. Viviane Jonckers voor het ondersteunen van deze thesis. Hierbij wil ook mijn begeleiders, Eline Philips en Niels Joncheere, oprecht bedanken voor de goede begeleiding bij het maken van deze thesis. Met al mijn vragen kon ik steeds bij hen terecht en de vele meetings verliepen steeds in een constructieve en aangename sfeer. Hiernaast wil ik ook mijn familie, en in het bijzonder mijn ouders bedanken voor de steun en kansen die ze me gegeven hebben doorheen mijn studies. Vervolgens wil ik ook nog mijn tante Veerle en nonkel Wim bedanken voor het grondig nalezen van mijn thesis. Tenslotte wil ik ook nog Kevin, Gertjan, Gillis en Bernd bedanken voor het voorzien van de nodige ontspanning en cafébezoekjes wanneer er nood was aan wat extra ontspanning.

Lijst van figuren

Lijst van tabellen

Listings

Inhoudsopgave

Hoofdstuk 1

Inleiding

1.1 Motivatie

Wanneer we kijken naar het dagdagelijkse leven constateren we dat mensen meer en meer gebruik maken van allerlei toestellen om het leven te vereenvoudigen [Barkhuus and Polichar, 2011]. We maken meer en meer gebruik van mobiele apparaten, zoals bijvoorbeeld smartphones, tablets en laptops. Dankzij de connectiviteit van de verschillende apparaten met een mobiel netwerk, kan men deze apparaten gebruiken om met elkaar te communiceren [Beale, 2005]. Doorheen de dag worden verschillende taken uitgevoerd door zowel personen als apparaten. Wanneer we nu al deze taken plaatsen in een workflow [van der Aalst and van Hee, 2002], laat dit ons toe om deze op een gestructureerde manier te ordenen, en zijn we zeker dat ze in de juiste volgorde worden uitgevoerd. Wanneer door verschillende personen op hun mobiele apparaten taken worden uitgevoerd in dezelfde omgeving, is het mogelijk dat er gemeenschappelijke belangen zijn, en dat deze beter kunnen worden verwezenlijkt wanneer er een samenwerking is. Om dit te bereiken is er communicatie nodig tussen de verschillende workflows die hieraan deelnemen.

Wij zullen ons concentreren op *nomadische netwerken* [Mascolo et al., 2002] waar de workflow wordt uitgevoerd op een vaste infrastructuur. Dit geeft ons het voordeel dat we niet op voorhand specifiek moeten bepalen wie welke taken gaat uitvoeren. De personen die in de omgeving van de vaste infrastructuur komen voeren dan een taak uit. Mensen komen en gaan in deze omgeving en voeren de taken uit waartoe ze in staat zijn, en die moet worden uitgevoerd volgens de vaste infrastructuur. We kiezen voor een nomadisch netwerk omdat er dan een vaste, centrale infrastructuur aanwezig is die steeds werkt. Dit geeft ons de mogelijkheid om complexere toepassingen uit te voeren. Andere soorten van netwerken, zoals bijvoorbeeld peer-to-peer, zijn hiervoor veel minder geschikt en zullen enkel gebruikt worden om zeer eenvoudige toepassingen uit te voeren. Wanneer we verschillende workflows hebben die elk op hun eigen vaste infrastructuur wordt uitgevoerd, maar op dezelfde locatie, is het mogelijk dat er een samenwerking vereist is tussen de verschillende workflows, waarbij de taken van een workflow afhankelijk zijn van de taken van een andere. Om dit te verwezenlijken zal er een gegevensuitwisseling nodig zijn tussen

de verschillende workflows. Een mogelijk voorbeeld van de samenwerking tussen verschillende workflows is de samenwerking tussen verschillende hulpdiensten in een rampgebied, waar een grote nood aan is [Rao et al., 2007]. Iedere hulpdienst, zoals de brandweer en de politie, heeft zijn eigen workflow die elk wordt uitgevoerd op een aparte vaste infrastructuur, zoals bijvoorbeeld in de wagens van de brandweercommandant en de politiecommissaris. Wanneer deze dan op dezelfde locatie worden uitgevoerd is er een samenwerking vereist.

Deze workflows kunnen worden geïmplementeerd in NOW [Philips, 2013], een nomadische workflow taal die ons de mogelijkheid geeft om workflows uit te voeren in een nomadisch netwerk. Wanneer een taak van de workflow is uitgevoerd op een mobiel apparaat zal dit, samen met het resultaat, gecommuniceerd worden met de vaste infrastructuur. Doordat er in een nomadisch netwerk verschillende falingen kunnen voorkomen, voorziet NOW meerdere mogelijkheden om deze falingen af te handelen. NOW laat ons toe om op een eenvoudige manier workflows te definiëren, en voorziet de nodige abstracties om compensaties te specificeren voor falingen.

1.2 Probleemstelling

NOW geeft ons de mogelijkheid om verschillende workflows onafhankelijk van elkaar te definiëren, maar wanneer deze met elkaar moeten samenwerken brengt dit enkele problemen met zich mee. Eerst en vooral moeten we kunnen bepalen wanneer onafhankelijke workflows met elkaar moeten samenwerken. Zo is het mogelijk dat een workflow pas kan starten wanneer een andere workflow een bepaald punt bereikt heeft. Zo kan bijvoorbeeld de workflow van de politie pas gestart worden wanneer de brandweer een bepaalde activiteit heeft voltooid. Dit zijn twee onafhankelijke workflows die met elkaar moeten communiceren. Een tweede probleem stelt zich wanneer verschillende onafhankelijke workflows worden uitgevoerd, deze met elkaar moeten gesynchroniseerd worden. Een derde probleem is dat we altijd rekening moeten houden met mogelijke netwerk falingen. Dit komt doordat de workflows worden uitgevoerd in nomadische netwerken die gebruik maken van volatiele netwerkverbindingen, waardoor er een grote kans is op falingen van het netwerk. Het is mogelijk dat alles goed gaat en beide workflows hun taken perfect kunnen uitvoeren, maar het is ook mogelijk dat alles fout gaat: het netwerk kan volledig offline zijn, de persoon kan geen verbinding met het netwerk maken door problemen met zijn apparaat, ...

Het eerste probleem dat we kunnen identificeren is wanneer we twee verschillende onafhankelijke workflows hebben, het onmogelijk is om te bepalen dat een workflow mag starten wanneer een andere workflow een specifiek punt bereikt heeft. Naast het enkel starten van een andere workflow is het ook mogelijk dat hetgeen een workflow moet uitvoeren afhankelijk is van het resultaat van een andere workflow. Daarom moeten we ook de mogelijkheid hebben om data tussen verschillende workflows in dezelfde omgeving uit te wisselen. Momenteel kan een andere workflow in NOW enkel maar gestart worden door gebruik te maken van een expliciete trigger. Het nadeel hieraan is dat we deze trigger specifiek moeten definiëren in elke workflow die een andere workflow wil starten, alsook deze trigger alleen maar kan gebruikt worden om workflows

te starten die op dezelfde vaste infrastructuur worden uitgevoerd. Een mogelijke oplossing om deze trigger te verwijderen, is door gebruik te maken van het *inversion of control* [Pisa, 2009] principe, waarbij we enkel bij een workflow definiëren wanneer deze mag gestart worden, afhankelijk van de staat van andere onafhankelijke workflows, en waarbij we deze niet meer moeten wijzigen. Nog een reden om triggers te vermijden is het feit dat niet alle workflows op voorhand gekend zijn, waardoor we hieraan dan ook onmogelijk een trigger op voorhand kunnen toevoegen.

Het tweede probleem stelt zich wanneer de politie bijvoorbeeld moet wachten op het resultaat van het onderzoek van de brandweer, het niet mogelijk is om dit aan te geven in de workflow. We hebben het reeds gehad over het feit dat twee verschillende workflows data kunnen uitwisselen, maar voor de rest blijven deze workflows totaal onafhankelijk van elkaar. Daarom zullen we een synchronisatiepunt voorzien, waarbij we kunnen zeggen dat de workflow, volgend op het synchronisatiepunt, pas kan verdergaan wanneer een andere workflow een specifiek punt bereikt heeft of wanneer in een andere workflow specifieke data beschikbaar is. Wanneer deze data beschikbaar en uitgewisseld is met de andere workflow kan deze verdergaan en zijn uitvoering van taken verder zetten.

Een derde probleem is dat er verschillende disconnecties en reconnecties zijn waardoor taken van de workflow niet op tijd kunnen worden voltooid. Dit komt omdat de toestellen en de vaste infrastructuur functioneren in een omgeving waar volatiele verbindingen overheersen. Daarom voorzien we ook specifieke afhandeling van falingen. Deze hebben een invloed op de vorige twee problemen. Wanneer een bepaald punt in een workflow niet bereikt is binnen een bepaalde tijd, kunnen we een alternatieve actie definiëren die moet worden uitgevoerd. Wanneer bijvoorbeeld de brandweer na een bepaalde tijd de familie van de slachtoffers nog niet heeft ingelicht kan de politie dit doen in hun plaats.

1.3 Contributie

We zullen deze problemen oplossen door gebruik te maken van een aspectgeörienteerde oplossing [Kiczales et al., 1997]. Dit laat ons toe om op een eenvoudige manier *inversion of control* toe te passen, wat ons een verhoogde mate van ontkoppeling geeft in de code. Hiernaast verhoogt dit ook de modulariteit, als we iets willen wijzigen kan dit dan op één locatie in plaats van dit op verschillende plaatsen in de code te moeten wijzigen. Doordat onze oplossing gebruik maakt van een aspectgeörienteerde manier moeten we in de code de verschillende plaatsen kunnen aanduiden waar er iets extra moet worden uitgevoerd, namelijk de joinpoints. Omdat we AO willen toepassen op workflows moeten we kijken welke punten we hierin kunnen identificeren. We kunnen ondermeer kijken naar de activiteiten die in de workflow worden uitgevoerd. Omdat we niet altijd de andere workflows, en de activiteiten die deze bevat, op voorhand kennen, kunnen we ook een joinpoint nemen op de data die door een activiteit wordt toegevoegd aan de workflow. Wanneer we de verschillende joinpoints geïdentificeerd hebben kunnen we deze dan selecteren door pointcuts. We kunnen ook verschillende pointcuts combineren, waarbij we de in-

tersectie van meerdere pointcuts nemen. Hierbij moeten alle pointcuts voldaan zijn vooraleer het advice wordt uitgevoerd. Wanneer we de unie nemen van meerdere pointcuts, wordt het advice steeds uitgevoerd wanneer één van de pointcuts voldaan is. Wanneer we een pointcut hebben geselecteerd, kunnen we bepalen in het advice wat er moet gebeuren. Door de voorziening van aspecten, pointcuts en advices kunnen we het eerste probleem aanpakken. Omdat we een specifiek pointcut voorzien voor disconnecties kunnen we hiermee ook het derde probleem aanpakken.

Wanneer we de verschillende joinpoints geïdentificeerd hebben, kunnen we bepalen wat er moet gebeuren wanneer deze bereikt worden. Dit definiëren we in het advice. Er zijn verschillende types van advices, zoals ondermeer het *before, after* en *around* advice. Afhankelijk van de types van deze advices, zal er bepaald worden wat er moet gebeuren. Wanneer het een anonieme functie is, zal dit worden uitgevoerd op het juiste moment. Deze functie zal data kunnen gebruiken die momenteel nog niet beschikbaar is in de workflow, maar dit zal later wel het geval zijn wanneer het joinpoint bereikt is. Wanneer het argument een pointcut is, zal op deze positie een synchronisatiepunt worden toegevoegd, waarmee we het tweede probleem kunnen aanpakken. Wanneer het argument een deel van een workflow is, zal deze worden toegevoegd op de juiste plaats.

1.4 Overzicht van de thesis

In hoofdstuk 2 bespreken we uitgebreid de context van deze thesis. We kijken naar de structuur van workflows en uit welke verschillende patronen deze kan bestaan. Vervolgens overlopen we NOW, een nomadische workflow taal die ons toelaat om workflows te implementeren in nomadische netwerken. Hiernaast bespreken we ook nog aspectgeörienteerd programmeren, en welke technieken er reeds bestaan om dit toe te passen op workflows. Hoofdstuk 3 gaat over hoe we ons probleem aanpakken. We bediscussiëren aan welke vereisten de oplossing moet voldoen, en hoe aan deze vereisten wordt voldaan. We overlopen het joinpoint model en de pointcut taal, alsook het advice model en taal. Hierna bespreken we het aspect module model en de compositie van aspecten. In hoofdstuk 4 presenteren we de implementatie van onze oplossing. We overlopen het design van een aspect, de pointcuts en de advices. Vervolgens bespreken we ook welke wijzigingen er nodig zijn aan de implementatie van NOW. In hoofdstuk 5 voorzien we een validatie van onze oplossingen, waarbij we 2 scenario's bespreken, en de vergelijking maken tussen onze oplossing, en hoe we het zouden doen door enkel gebruik te maken van NOW. Tenslotte trekken we in hoofdstuk 6 een conclusie.

Hoofdstuk 2

Context

In het volgende hoofdstuk gaan we dieper in op workflows en bestaande workflow talen. We bespreken eerst het doel van een workflow en de verschillende perspectieven. Hierna overlopen we de terminologie van workflows, en bespreken we kort BPEL, een workflowtaal voor de orchestratie van webservices. Omdat de nadruk van deze thesis ligt op het laten samenwerken van workflows in nomadische netwerken, overlopen we NOW, een nomadische workflowtaal die ons toelaat om workflows te definiëren en uit te voeren in nomadische netwerken. Hiernaast gaan we ook wat dieper in op open workflow, een paradigma gebruikt om workflows te combineren in mobiele ad-hoc netwerken. Omdat we willen gebruik maken van een aspectgeoriënteerde oplossing, bespreken we het doel hiervan en hoe dit wordt verwezenlijkt. Tenslotte overlopen we enkele bestaande programma's die aspectgeoriënteerde technieken toepassen op workflowtalen, zoals AO4BPEL en Unify.

2.1 Workflows

Een workflow wordt gedefinieerd als de automatisering van business processen. Hierbij kunnen de nodige documenten, informatie of taken worden uitgewisseld tussen de deelnemers die verantwoordelijk zijn voor het uitvoeren van acties, en dit moet gebeuren volgens een reeks van procedurele regels [Workflow Management Coalition, 1999]. Workflowtalen trachten alle relevante workflow informatie te omvatten waarbij er een gecontroleerde uitvoering van de workflow moet worden mogelijk gemaakt door het workflow management systeem [Georgakopoulos et al., 1995]. De verschillende werkeenheden (activiteiten) waaruit een proces bestaat moeten dus worden gespecificeerd, alsook in welke volgorde ze moeten worden uitgevoerd. Dit wordt het *control flow* perspectief van de workflow genoemd [Russell et al., 2006]. Naast het *control flow* perspectief zijn er ook nog 4 andere perspectieven. Het *data* perspectief specificeert hoe data moet worden verwerkt door de workflow [Russell et al., 2004a]. Hiernaast bestaat ook nog het *resource* perspectief [Russell et al., 2004b], waarbij het werk van een workflow wordt toegekend aan specifieke middelen. Het vierde perspectief is het *exception handling* perspectief [Nick Russell and Hofstede, 2006], dat bepaalt wat er moet gebeuren wanneer er een exceptie voorkomt, en het vijfde en laatste perspectief is het *presentatie* perspectief [La Rosa et al., 2011a] [La Rosa

et al., 2011b], dat definieert hoe een workflow grafisch wordt weergegeven.

Het Workflow Pattern Initiative [van der Aalst et al., 2012] voorziet een beschrijving van verschillende workflow patronen die worden gebruikt in workflows, met respect voor reeds voorgestelde workflow patronen abstracties die voorkomen in bestaande workflowtalen. We bespreken nu de verschillende perspectieven voorgesteld door het Workflow Pattern Initiative.

- **Control flow perspectief**
  Door het grondig bestuderen van de modellering van business processen worden er enkele fundamentele vereisten ontdekt en deze kunnen beschreven worden op een imperatieve wijze. Na de eerste ontdekking waren er 20 patronen die het *control flow* perspectief van workflow systemen beschrijven [van der Aalst et al., 2003]. Na een grondige evalutatie van de bestaande patronen en vereisten, zijn er nog 23 nieuwe patronen ontdekt, waardoor het *control flow* perspectief momenteel 43 patronen bevat. Er zijn verschillende categorieën van patronen, zoals bijvoorbeeld de basis *control flow* categorie, die ondermeer het *sequence* patroon en het *parallelle split* patroon bevat. Een andere meer geavanceerde categorie zijn de branch en synchronisatie patronen. Deze categorie bevat ondermeer het *Multi-Choice* patroon en het *Structured Partial Join* patroon.

- **Data flow perspectief**
  In dit perspectief wordt gefocust op hoe data kan worden gerepresenteerd en gemanipuleerd in workflows [Russell et al., 2004a]. Er worden patronen voorzien die zorgen voor data zichtbaarheid, data interactie, data overdracht, en datagebaseerde routing. In totaal worden er 40 verschillende patronen voorzien.

- **Resource perspectief**
  Dit perspectief [Russell et al., 2004b] legt de nadruk op resources, die kunnen gedefinieerd worden als "entiteiten die een taak kunnen uitvoeren". Er wordt gedefinieerd hoe deze resources worden gerepresenteerd en gebruikt in workflows. Er zijn verschillende soorten van categoriën voorzien, zoals ondermeer *creatie-*, *push-* en *pull* patronen.

- **Exception handling perspectief**
  Dit perspectief [Nick Russell and Hofstede, 2006] achterhaalt de oorzaak van verschillende exceptions in workflows en hoe deze kunnen worden afgehandeld. Er zijn vijf verschillende types van exceptions mogelijk: een *work item* faling, een verstreken deadline, een resource die onbeschikbaar is, een externe trigger en tenslotte een constraint overtreding. Deze falingen worden opgevangen door drie mogelijke compensaties: niets doen, een compensatie uitvoeren, of teruggaan en opnieuw proberen.

- **Presentatie perspectief**
  Het laatste perspectief is het presentatie perspectief, dat zal bepalen hoe een workflow visueel wordt weergegeven. Er worden verschillende patronen voorzien die worden onderverdeeld in twee klassen: de abstracte syntax [La Rosa et al., 2011b] en concrete syntax [La Rosa et al., 2011a]. De patronen die bij de abstracte syntax horen zullen de com-

plexiteit van workflows vereenvoudigen, terwijl patronen die bij de concrete syntax horen de visuele representatie van workflows verbeteren.

Deze perspectieven worden gebruikt voor de evaluatie van workflowtalen.

2.1.1 Terminologie

We overlopen verschillende termen die worden gebruikt in workflows. Deze definities van termen zijn vastgelegd door de Workflow Managment Coalition [Workflow Management Coalition, 1999].

- **Business process** is gedefinieerd als *"een verzameling van één of meer gelinkte procedures of activiteiten die gezamenlijk een bedrijfsdoelstelling of beleidsdoel realiseren. Dit gebeurt normaal gezien binnen de context van een organisatorische structuur die verschillende functies en relaties definieert."*

- **Workflow** is *"de automatisering van business processes, gedeeltelijk of volledig, waarbij documenten, informatie of taken worden doorgegeven van de ene deelnemer naar de andere, waarbij deze een actie moeten uitvoeren volgens een reeks van procedurele regels."*

- **Workflow engine** is *"een software service die de run time uitvoering omgeving voorziet voor een proces instantie."*

- **Workflow management system** is *"een systeem dat de uitvoering van workflows definieert, creëert en beheert door gebruik te maken van software, uitgevoerd op één of meerdere workflow engines, die in staat zijn om de verschillende proces definities te interpreteren, te interageren met workflow deelnemers en, indien nodig, het invokeren van het gebruik van IT tools en applicaties."*

- **Proces** is gedefinieerd als *"een geformaliseerd overzicht van een business process, gerepresenteerd als een gecoördineerde verzameling (parallel en/of serieel) van proces activiteiten, die verbonden zijn met elkaar om een gemeenschappelijk doel te bereiken."*

- **Deelproces** is *"een proces dat wordt vastgesteld of opgeroepen van een ander (geïnitieerd) proces (of deelproces), en dat deel uitmaakt van het complete (geïnitieerde) proces."*

- **Activiteit**, ook gekend als een taak, is gedefinieerd als *"een beschrijving van een deel van het werk dat een logische stap vormt binnen het proces."*

- **Instantie** is *"de weergave van een enkele vaststelling van een proces, of activiteit binnen een proces, waartoe ook de geassocieerde data behoort. Elke instantie stelt een aparte thread van de uitvoering van het proces of activiteit voor, welke onafhankelijk kan gecontroleerd worden en zijn eigen interne staat heeft, alsook zijn uitwendig zichtbare identiteit."*

- **Workflow participant** is gedefinieerd als *"een bron die het werk uitvoert, gerepresenteerd door een instantie van een workflow activiteit."*

- **Proces definitie** is gespecificeerd als *"de representatie van een business process in een vorm die geautomatiseerde manipulatie voorziet, zoals modellering, of in werking treden door een workflow management systeem."*

2.1.2 (WS)BPEL

BPEL [Ouyang et al., 2007], dat staat voor *Business Process Execution Language*, is een workflowtaal die het mogelijk maakt om op een eenvoudige manier webservices te orchestreren. Het is zo ontwikkeld dat het zowel de definitie van abstracte processen ondersteunt alsook de definitie van uitvoerbare processen. *Abstracte processen* zijn processen die het gedrag van een klasse van diensten specificeert, en waarbij constraints worden vastgelegd op het ordenen van berichten die moeten ontvangen en verzonden worden door een service. Een *uitvoerbaar proces* definieert de volgorde van uitvoering van een verzameling activiteiten, de partners betrokken bij het proces, de berichten die tussen de partners worden verzonden, en de reactie op specifieke events en fouten.

Een BPEL proces bestaat uit een aantal activiteiten. Deze activiteiten worden onderverdeeld in twee categorieën: basis activiteiten en gestructureerde activiteiten. Onder *basis activiteiten* vallen atomaire acties, zoals *invoke*, *receive*, *reply*, *wait*, *assign*, *throw*, *exit*, *compensate* en *empty*. *Gestructureerde activiteiten* leggen gedrags- en uitvoeringconstraints op aan een reeks van activiteiten die hierin worden uitgevoerd. Voorbeelden daarvan zijn *sequence*, *flow*, *switch*, *pick*, *while* and *scope*. Deze gestructureerde activiteiten kunnen worden genest en gecombineerd op verschillende manieren, wat ons toelaat om complexe structuren te definiëren in BPEL processen.

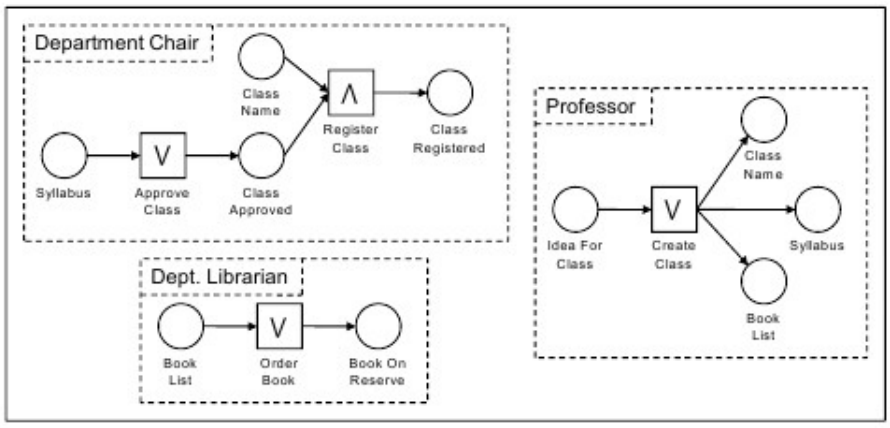
Naast een activiteit bevat een BPEL proces ook nog een aantal variabelen en partner links. De variabelen representeren de globale data van het proces. Elke variabele heeft een type dat wordt gedefinieerd door gebruik te maken van XML Schema [van der Vlist, 2002]. Een aantal partner links representeren de verschillende partners waarmee de processen samenwerken. Deze partner links corresponderen dus met een aantal webservices. Elke partner link heeft een type dat is gedefinieerd in de corresponderende specificatie van de webservice zijn WSDL interface.

2.2 Samenwerking tussen workflows

Momenteel is de huidige workflowtechnologie gebaseerd op servers, die de statisch gedefinieerde workflow bijhouden en het workflow management systeem of engine aanbieden. Voorbeelden van deze workflowtalen zijn BPEL [Ouyang et al., 2007] en YAWL [van der Aalst and ter Hofstede, 2005]. In de huidige tijd maken personen meer en meer gebruik van mobiele apparaten. Personen komen bij elkaar in de buurt en gaan na een tijd weer uit elkaar, waardoor er een nood is aan de dynamische generatie van workflows. De term *open workflow* [Thomas et al., 2009] wordt gebruikt om de specificatie aan te duiden van verschillende workflows, zijn constructie en hoe deze kan worden uitgevoerd door gebruik te maken van een mobiele ad-hoc draadloos netwerk als onderliggende infrastructuur. Het systeem dat deze workflow uitvoert evolueert in de loop ter

tijd, en bestaat uit een aantal deelnemers (personen die een mobiel apparaat bij zich dragen) die zich verplaatsen doorheen de ruimte en met elkaar en de omgeving kunnen communiceren.

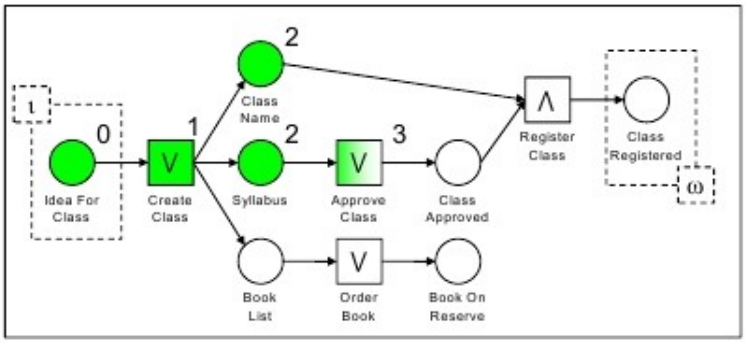
Voor het laten samenwerken van verschillende workflows moeten de verschillende deelnemers elk een mobiel apparaat bij zich dragen waardoor er een draadloos mobiel ad-hoc netwerk kan worden gevormd. Wanneer één van de deelnemers identificeert dat er een bepaald doel moet worden bereikt, wordt er automatisch een samengestelde workflow gecreëerd. Dit wordt verwezenlijkt door het combineren van workflows die gekend zijn in de omgeving. Wanneer de samengestelde workflow is gecreëerd worden de taken verdeeld onder de verschillende deelnemers die in staat zijn om bepaalde taken uit te voeren.



Figuur 2.1: Workflow fragmenten in een omgeving.

Voor het combineren van de verschillende workflows tot één grote workflow wordt er gebruikt gemaakt van een algoritme. Hierbij worden workflows voorgesteld als graffen, waarbij de taken worden voorgesteld door nodes en data afhankelijkheden tussen taken door bogen. In figuur 2.1 zien we de representatie van workflows door graffen. Eén workflow gaat over hoe een professor een nieuw vak creëert, de tweede workflow toont hoe een vak wordt goedgekeurd door het bestuur van het departement, en de derde workflow gaat over het bestellen van boeken in de bibliotheek. Deze workflows komen voor in dezelfde omgeving.

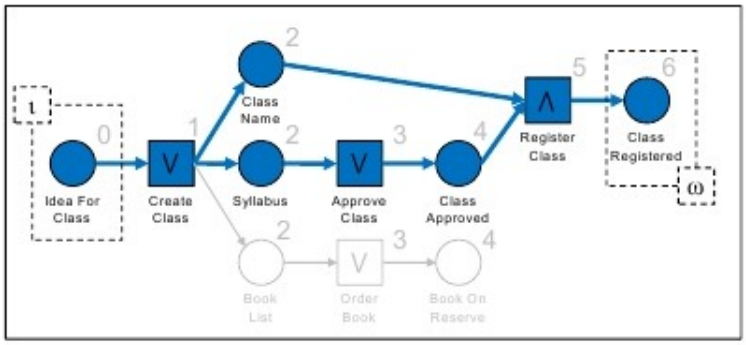
We kunnen nu deze graffen combineren in één grote graf, door de verschillende graffen die dezelfde taken hebben samen te voegen. In figuur 2.1 worden enkele taken geïdentificeerd die in meerdere workflows worden gebruikt, namelijk *Class name*, *Syllabus* en *Book list*. De start van deze workflow is *Idea for class* en het einddoel *Class registered*. Het is echter niet zeker dat de nieuwe samengestelde workflow geldig is, deze kan loops bevatten, ontbrekende input, ongeldige output, ... Daarom wordt er één uitvoerbare workflow in deze samengestelde workflow geïdentificeerd. Dit wordt gedaan door eerst alle nodes een afstand te geven vanaf het beginpunt van de graf. In figuur 2.2 start de node bij *Idea for class*, die als waarde 0 krijgt. Hierna worden



Figuur 2.2: Exploreren van de supergraf.

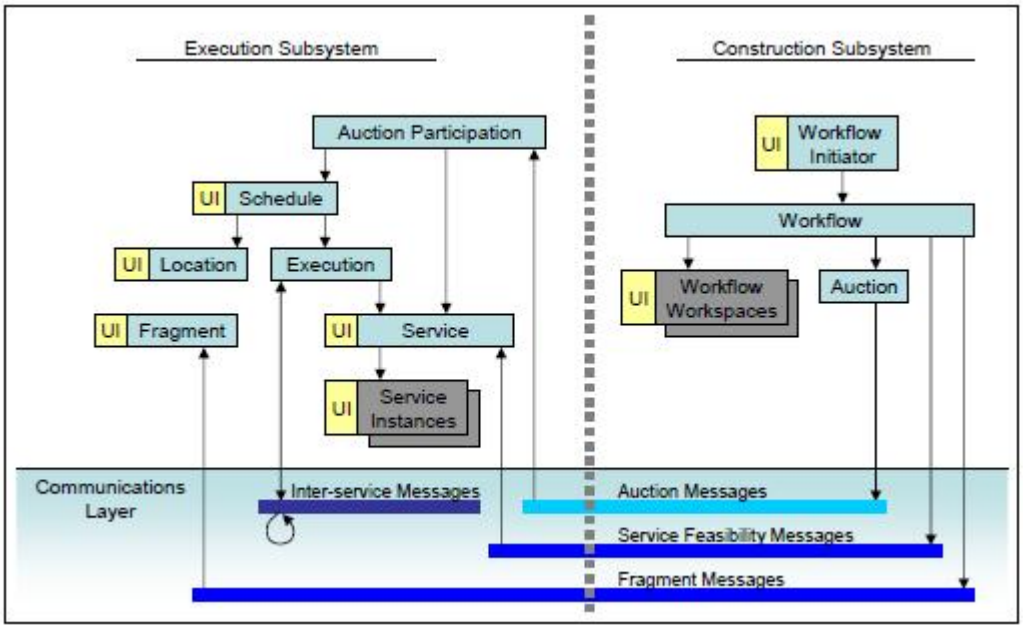
de bogen gevolgd naar de volgende nodes, die als waarde 1 krijgen. Zo wordt verder gegaan tot alle nodes een afstand vanaf het beginpunt hebben gekregen. Wanneer er een disjunctie voorkomt nemen we de laagste afstand, terwijl bij een conjunctie de maximale afstand wordt genomen.

Na het bepalen van de afstanden tussen de nodes wordt de samengestelde workflow terug volledig afgelopen maar nu in de omgekeerde richting, van het doel naar de start, waarbij alle belangrijke nodes worden geïdentificeerd. Dit zijn de nodes die moeten worden uitgevoerd voor het bereiken van het doel. Het resultaat hiervan kan gezien worden in figuur 2.3. De nodes in het blauw stellen een geldige workflow voor die moet worden uitgevoerd om het doel te bereiken.



Figuur 2.3: Resultaat samengestelde workflow.

We gaan ervan uit dat er steeds één deelnemer is die de constructie van de open workflow initieert, namelijk de coördinator. Deze gaat eerst de samengestelde workflow construeren, waarna kan worden begonnen met de allocatie en uitvoering ervan. De coördinator is dan ook verantwoordelijk voor de communicatie met de gebruikers zodat alle taken van de workflow worden gealloceerd door hen. Hiervoor moet deze dan ook eerst verschillende metadata berekenen voor alle taken die de workflow bevat, zoals bijvoorbeeld de topologische ordening. Hierna stuurt de coördinator taaksollicitaties naar de werkers die de tijd, locatie en service van de taak vergelij-



Figuur 2.4: Systeem Architectuur.

ken met hun eigen mogelijkheden en capaciteiten. Tot deze werkers behoren alle personen die willen deelnemen aan de uitvoering van de workflow. Wanneer een werker een taaksollicitatie ontvangt die past in zijn planning en waarvoor deze de nodige capaciteiten bevat, zal hij een bod uitbrengen op deze taak. Dit bod bevat informatie over hoe goed de uitvoering van de taak past in het schema van de werker, alsook in welke mate de werker in staat is om deze taak goed uit te voeren. Bij dit bod hoort ook een deadline die aangeeft wanneer de coördinator ten laatste aan de werker moet laten weten of zijn bod wordt aanvaardt.

De coördinator krijgt dus van verschillende werkers biedingen binnen op taken. Hij zal steeds voorwaardelijk het beste bod selecteren voor de uitvoering van een taak. Wanneer er een nieuw bod binnenkomt van een werker wordt dit vergeleken met het voorlopig beste bod. Een finale beslissing wordt gemaakt wanneer de deadline voor de uitvoering van de taak bijna wordt overschreden, of de deadline van de werker die het beste bod aanbiedt. De coördinator zal zo lang mogelijk wachten met het alloceren van een taak zodat hij de best mogelijke uitvoerder kan vinden. Wanneer een taak gealloceerd is wordt aan de werker de nodige data bezorgd zodat deze de taak kan uitvoeren, waarna hij het resultaat terugstuurt naar de coördinator.

In figuur 2.4 hebben we een overzicht van een architectuur die ons toelaat om een open workflow te construeren en uit te voeren. Omdat een gebruiker zowel als coördinator als als werker kan acteren is het systeem opgesplitst in twee subsystemen. Eén subsysteem is verantwoordelijk voor de constructie van de workflow. Hierbij wordt ondermeer het probleem dat moet worden opgelost geïdentificeerd, een uitvoeringsplan geformuleerd en alle taken toegekend aan werkers. Het tweede subsysteem, verantwoordelijk voor de uitvoering, zal de taken accepteren waarvan

men in staat is om ze uit te voeren en deze dan ook effectief uit te voeren.

Het open workflow paradigma besproken in deze sectie geeft ons de mogelijkheid om dynamisch workflows te combineren. Dit wordt verwezenlijkt door gebruik te maken van een algoritme wat ons toelaat om een geldige samengestelde workflow te construeren. Het samenstellen van meerdere workflows wordt uitgevoerd door een coördinator, en de taken worden uitgevoerd door werkers. Dit paradigma laat ons toe om workflows dynamisch te laten samenwerken, maar omdat er gebruik wordt gemaakt van een peer-to-peer netwerk ontbreekt er een centrale eenheid die alle belangrijke data bijhoudt. Een voorbeeld hiervan is een overzicht van alle passagiers op een luchthaven. Het is onmogelijk om deze data bij te houden op één mobiel toestel, daarom maken we gebruik van nomadische netwerken.

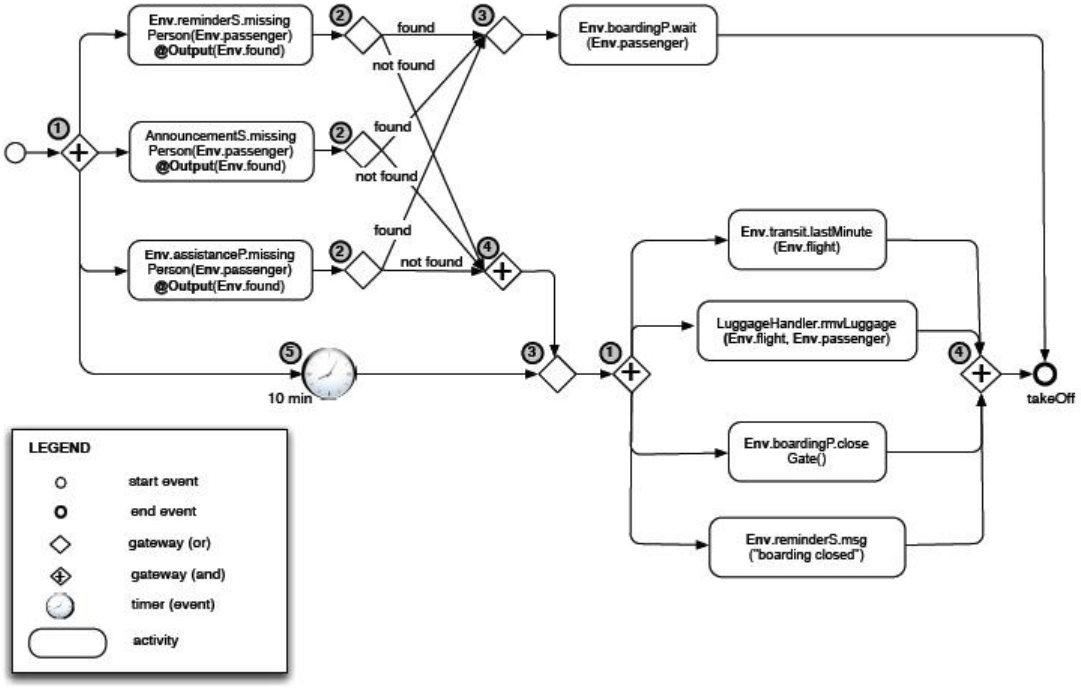
2.3 Workflows voor nomadische netwerken: NOW

Wanneer we kijken naar onze omgeving, merken we op dat we omringd zijn door allerlei mobiele apparaten. Dit gaat van smartphones tot laptops en tablets. Aangezien we deze ter beschikking hebben, kunnen we deze apparaten gebruiken voor het uitvoeren van taken. We kunnen verschillende soorten netwerken onderscheiden waarin dit soort apparaten verbonden is. In mobiele ad-hoc netwerken zullen verschillende personen bij elkaar komen en hun apparaten zullen verbinding met elkaar maken. Aangezien deze verbonden zijn door een draadloze verbinding, is het zeker dat er disconnecties zullen plaatsvinden, en hiermee moet rekening gehouden worden. In tegenstelling tot mobiele ad-hoc netwerken maken nomadische netwerken gebruik van een vaste infrastructuur. De mobiele apparaten zullen dan verbinding maken met deze infrastructuur in plaats van met andere mobiele apparaten. Aangezien ook hier gebruik gemaakt wordt van een draadloze verbinding zullen er disconnecties plaatsvinden.

In mobiele ad-hoc netwerken kunnen we workflows gaan definiëren. Wanneer we kijken naar bestaande workflowtalen voor mobiele netwerken, zoals Cian [Sen et al., 2008], een workflowtaal voor mobiele ad-hoc netwerken geïmplementeerd in Java, merken we op dat er hier geen rekening wordt gehouden met de grote kans op disconnectiviteit. Het is niet mogelijk om te bepalen wat er moet gebeuren wanneer er een disconnectie plaatsvindt. Wanneer we een workflow willen definiëren in een nomadisch netwerk, kunnen we gebruik maken van NOW [Philips et al., 2013], een workflowtaal voor nomadische netwerken geïmplementeerd bovenop AmbientTalk [Van Cutsem et al., 2007]. Hierin zullen services, diegenen die bepaalde activiteiten gaan uitvoeren, dynamisch worden ontdekt. Wanneer een service niet beschikbaar is, zal dit de volledige workflow niet stoppen. De workflowtaal laat ons toe om te beschrijven wat er moet gebeuren wanneer er een faling plaatsvindt. Zo kunnen we zeggen dat wanneer er een bepaalde service gedisconnecteerd is, er een activiteit in de workflow moet worden vervangen. In de volgende secties zullen we NOW verder in detail bespreken.

2.3.1 Workflow voorbeeld

Laten we eerst eens kijken naar een voorbeeld van een workflow gedefinieerd in NOW. In afbeelding 2.5 zien we een workflow die kan worden uitgevoerd in een luchthaven wanneer er een persoon niet komt opdagen voor een vlucht. Als dit gebeurt wordt er een bericht naar deze persoon gestuurd, hij wordt omgeroepen in de luchthaven, en het personeel wordt verwittigd om uit te kijken naar deze persoon. Wanneer deze persoon binnen een bepaalde tijd gevonden wordt, zal de vlucht wachten tot deze gearriveerd is. Als de gemiste persoon niet op tijd wordt teruggevonden, zal men de reisorganisatie laten weten dat er nog een lastminute beschikbaar is, de vermiste persoon zijn bagage verwijderen, de boarding afsluiten en laten weten aan alle diensten dat de boarding is afgesloten.



Figuur 2.5: Voorbeeld Workflow Luchthaven.

We merken meteen enkele verschillende patronen op die worden voorzien in NOW. De workflow start met een *parallelle split*(1) zodat meerdere activiteiten in parallel kunnen worden uitgevoerd. Er wordt een herinnering verstuurd, de vermiste persoon wordt omgeroepen, en extra assistentie wordt verwittigd. Elk van deze activiteit is verbonden met een *exclusive choice*(2), die bepaalt wat er gebeurt als de persoon al dan niet gevonden is. Een ander patroon dat wordt gebruikt is het *cancelling discriminator* patroon(3). Van zodra deze een signaal krijgt dat de persoon gevonden is, wordt de boarding verwittigd dat ze moeten wachten tot de persoon aanwezig is. Het *synchronisatie* patroon(4) wordt gebruikt om zeker te zijn dat, wanneer alle activiteiten de persoon niet gevonden hebben, de nodige taken worden uitgevoerd. Hierna wordt dit patroon nog eens

gebruikt, om er zeker van te zijn dat de vlucht pas kan opstijgen wanneer alle vorige 4 taken zijn afgerond. Tenslotte hebben we nog een *persistent trigger* patroon(5), die ervoor zorgt dat na 10 minuten een signaal wordt gestuurd naar een *cancelling discriminator* patroon.

2.3.2 Services

De verschillende services kunnen worden uitgevoerd op mobiele apparaten. We moeten nu kunnen aanduiden welke services er op een mobiel apparaat specifiek kunnen worden uitgevoerd. Dit kan in AmbientTalk bereikt worden door een object te gaan taggen als een service, en exporteren als een bepaalde service. Dit object zal dan de verschillende methoden bevatten die deze service kan uitvoeren. Zo zal het voorbeeld in listing 2.1 op lijn 3 een service definiëren. Vervolgens definiëren we het object die de verschillende methoden bevat die een service kan uitvoeren. Deze methoden representeren de activiteiten gedefinieerd in een workflow. De argumenten van de activiteit in de workflow komen overeen met deze van de methode, en de output is het resultaat van de methode. Zo hebben we op lijn 12 de methode `removeLuggage`, die als parameters de id van de vlucht en de passagier meekrijgt, en als resultaat de locatie van zijn bagage teruggeeft. Op lijn 20 zien we dat de functie `applicationService` wordt opgeroepen, die de service publiciteert in het netwerk onder een service naam. Hier zal dit de service `LuggageHandler` zijn, die altijd een subtype moet zijn van service, wat wordt verwezenlijkt op lijn 1.

```
1  deftype LuggageHandler <: Service;
2
3  def service := isolate: {
4      def companyName := "Aviapartner";
5      /* ... other fields */
6
7      def init(cn) {
8          self.companyName := cn;
9          /* assignment of other fields */
10     };
11
12     def removeLuggage(flight, name) {
13     /* Remove luggage from flight,
14     and return the location where it is stored */
15     };
16
17    /* ... other methods */
18 };
19
20 applicationService(LuggageHandler, service);
```

Listing 2.1: Service bagage afhandeling.

Het is mogelijk dat de vorige service uitgevoerd en geëxporteerd wordt vanaf een mobiel apparaat. De volledige workflow wordt uitgevoerd op een vaste infrastructuur. Aangezien het object gedefinieerd in listing 2.1 wordt geëxporteerd als een `LuggageHandler` service, moeten we een verwijzing hebben naar deze service in de workflow, wat gebeurt in listing 2.2 op lijn 1.

```
1 createServiceType(`LuggageHandler);
2 LuggageHandler.removeLuggage(Env.flight, Env.name)@Output(Env.location);
```

Listing 2.2: Definiëren activiteit in NOW.

Hierna kan in de workflow de service dan een taak uitvoeren, zoals gedefinieerd op lijn 2, waarbij de bagage verwijderd wordt door `removeLuggage`. We merken op dat deze service twee parameters heeft, `Env.flight` en `Env.name`. Hierbij is `Env` een object dat wordt doorgegeven doorheen de volledige workflow en alle data bevat die behoort tot de workflow, wat we de omgeving noemen. `Env.flight` geeft de data van de workflow terug met naam `flight`. Op lijn 2 zien we dat de service wordt afgesloten met `@Output(Env.found)`. Dit betekent dat de service een resultaat moet teruggeven, die in de omgeving wordt bijgehouden onder de naam `found`.

2.3.3 Patronen

Naast enkel de definitie van activiteiten hebben we ook patronen nodig zodat we de activiteiten in de juiste volgorde kunnen uitvoeren. NOW [Philips et al., 2013] voorziet 19 verschillende patronen die gedefinieerd zijn door van der Aalst [Russell et al., 2006]. In het voorbeeld van de luchthaven kunnen we volgende patronen identificeren: parallelsplit, synchronization, exclusive choice, cancelling discriminator en persistent trigger. In listing 2.3 hebben we een implementatie van de workflow van het voorbeeld over de luchthaven.

```
1 def sync1 := Synchronization(Env.tower.takeOff(Env.flight), restore);
2
3 def cancDiscr1 :=
4     CancellingDiscriminator(
5         ParallelSplit( Sequence( Env.transit.lastMinute(Env.flight),
6                                  Connection(sync1) ),
7                        Sequence( LuggageHandler.rmvLuggage(Env.flight,
8                                  Env.pass), Connection(sync1) ),
9                        Sequence( Env.boardingP.closeGate(),
10                                 Connection(sync1) ),
11                       Sequence( Env.reminderS.msg("boarding closed"),
12                                 Connection(sync1) ) ) );
13
14 def sync2 := Synchronization(Connection(cancDiscr1), restore);
15
16 def cancDiscr2 := CancellingDiscriminator(
17                       Sequence( Env.boardingP.wait(Env.pass),
18                                 Env.tower.takeOff(Env.flight) ) );
19
20 def wf := ParallelSplit(
21      Sequence( Env.reminderS.missingPerson(Env.pass)@Output(Env.found),
22                ExclusiveChoice( {|found| found}, Connection(cancDiscr2),
23                Connection(sync2) ) ),
```

```
24      Sequence( Env.announcementS.missingPerson(Env.pass)@Output(Env.found),
25                ExclusiveChoice( {|found| found}, Connection(cancDiscr2),
26                Connection(sync2) ) ),
27      Sequence( Env.assistanceP.missingPerson(Env.pass)@Output(Env.found),
28                ExclusiveChoice( {|found| found}, Connection(cancDiscr2),
29                Connection(sync2) ) ),
30      PersistentTrigger( Connection(cancDiscr1), after(minutes(10)) ) );
```

Listing 2.3: Workflow Voorbeeld Luchthaven.

Op lijn 1 en 14 hebben we een synchronisatie, waarbij beiden gebruik maken van de *restore merge* strategie. Hierbij wordt de omgeving van de workflow hersteld naar het punt waar de parallelle split gestart is. De synchronisatie op lijn 1 wordt gebruikt om er zeker van te zijn dat de vlucht pas opstijgt wanneer vier activiteiten voltooid zijn. Deze zijn gedefinieerd op lijn 5 tot 12. De tweede synchronize op lijn 14 is om zeker te zijn dat er pas actie wordt ondernomen wanneer alle drie de activiteiten (lijn 21 tot 29) hebben aangegeven dat de persoon niet gevonden is. We hebben ook twee `CancellingDiscriminator` patronen op lijn 4 en 16. Wanneer de persoon gevonden is wordt er een signaal gestuurd naar deze op lijn 16, waarna het vervolg van de workflow wordt uitgevoerd. De tweede `CancellingDiscriminator` zal de workflow vervolgen wanneer ofwel 10 minuten verstreken zijn sinds de start van de workflow (lijn 30) ofwel de synchronisatie op lijn 14 voltooid is. We maken ook nog gebruik van het `ExclusiveChoice` patroon waarbij het vervolg van de workflow afhankelijk is van het feit of de passagier gevonden is. Deze bevindt zich op lijn 22, 25 en 28. Hiernaast hebben we ook nog op lijn 5 en 20 een `ParallelSplit`, en een aantal `Sequence`.

2.3.4 Falingen

Aangezien onze workflows worden uitgevoerd in nomadische netwerken, is er een grote kans op disconnecties tussen de services en de vaste infrastructuur. Daarom voorziet NOW verschillende falingen en compensaties die kunnen worden uitgevoerd. Een mogelijke faling bestaat uit het feit dat er een disconnectie kan plaatsvinden met een service, waardoor een activiteit niet kan worden uitgevoerd. Een andere mogelijke faling is dat de verantwoordelijke service nog niet ontdekt is. Een derde mogelijke faling die kan plaatsvinden is wanneer een activiteit niet binnen een bepaalde tijd wordt uitgevoerd en er dus een timeout is. De vierde en laatste faling waarmee we rekening houden is wanneer er een exceptie zich voordoet veroorzaakt door een service. Het is ook mogelijk om deze falingen te combineren, of een compensatie pas uit te voeren wanneer een faling zich meerdere malen voordoet.

Naast het vaststellen van deze falingen voorziet NOW ook verschillende compensaties die kunnen worden uitgevoerd wanneer er zo een faling plaatsvindt. In NOW zullen we een faling en zijn afhandeling definiëren als een patroon die rond een component zit. Deze component kan een activiteit zijn die moet worden uitgevoerd door een service, of een patroon zoals een parallelle split die zelf activiteiten en/of patronen zal bevatten. Wanneer er bij één van deze patronen die een component bevat zich een faling voordoet, kan er een compensatie worden uitgevoerd. Zo

```
1 Sequence( Env.luggageS.getInfo(Env.flight)@Output(Env.trailer, Env.belt),
2           Failure( Env.trailer.getDuration(Env.belt)@Output(Env.time),
3                    [ Description(Timeout(seconds(20)), Retry(3, Skip())),
4                      Description(Disconnection(), Skip()) ]),
5                      Env.gui.update(Env.flight, Env.belt, Env.time) );
```

Listing 2.4: Falingen en compensaties op de luchthaven.

hebben we de mogelijkheid om deze component een aantal keer te laten herstarten, en wanneer dit steeds blijft falen kunnen we een andere actie uitvoeren. Deze actie kan opnieuw een compensatie zijn die moet worden uitgevoerd. Een andere mogelijke compensatie is dat we de activiteit die de faling veroorzaakt gewoon overslagen. Wanneer de faling veroorzaakt wordt door een service die offline is, kunnen we deze een aantal keer proberen te ontdekken, en wanneer dit steeds faalt een andere actie uitvoeren. Vervolgens hebben we ook nog de compensatie *retry* die de service nogmaals zal proberen te invokeren. Hiernaast kunnen we ook nog een compensatie die een tijdje wacht en hierna een andere actie uitvoert. Tenslotte kunnen we de component die de faling veroorzaakt vervangen door een andere component, ofwel een alternatief uitvoeren waarbij naast de component zelf ook de rest van de workflow die hierop volgt wordt vervangen door een andere component.

Wanneer we kijken naar het voorbeeld over falingen en compensaties op de luchthaven in listing 2.4, zien we op lijn 2 dat er een failure component zich rond de *getDuration* activiteit bevindt. Deze bevat twee falingen en compensaties die zullen worden uitgevoerd wanneer de faling plaatsvindt. Zo hebben we op lijn 3 een timeout, die wanneer de activiteit niet binnen de 20 seconden zijn resultaat teruggeeft drie keer opnieuw zal geprobeerd worden. Wanneer hierna de activiteit nog steeds geen resultaat heeft, zal deze worden overgeslagen. Op lijn 4 hebben we een tweede faling die kan voorkomen. Wanneer de verantwoordelijke service gedisconnecteerd is wordt deze activiteit overgeslagen. Op deze manier kunnen we falingen opvangen door verschillende compensaties uit te voeren.

2.3.5 Groepspatronen

Naast de definitie van gewone workflows is er in NOW ook nood aan groep orchestratie in een mobiele omgeving [Philips et al., 2012]. Hieronder verstaan we het organiseren van een verzameling van diensten die een logische groep vormen waarbij al de leden die tot deze groep behoren een specifiek proces moeten uitvoeren. Een voorbeeld hiervan kunnen we terugvinden op een festival, waar alle fans één grote groep vormen. Een band kan dan via het nomadische netwerk hun fans betrekken in hun optreden, door bijvoorbeeld te vragen naar de fans hun favoriete liedjes die zeker moeten gespeeld worden.

In listing 2.5 definiëren we op lijn 1 een groep die een beschrijving heeft en op lijn 2 de naam van de variabele die bij deze groep hoort. Deze groep zal zich dus richten tot alle personen die

```
1  Group( description,
2         `fan,
3         Sequence( Env.fan.show("vote?")@Output(Env.interested),
4                   Filter( {|env| env.find(`interested) == "yes"} ),
5                   Env.fan.show(Env.discography)@Output(Env.selection),
6                   SynchronisedTask(
7                       Env.headliner.show(Env.selection)@Output(Env.playlist),
8                       at(time(20,30,0)) ),
9                   CancellingBarrier( at(time(22,20,0)) ),
10                  Env.fan.show(Env.playlist) ) );
```

Listing 2.5: Voorbeeld groep orchestratie.

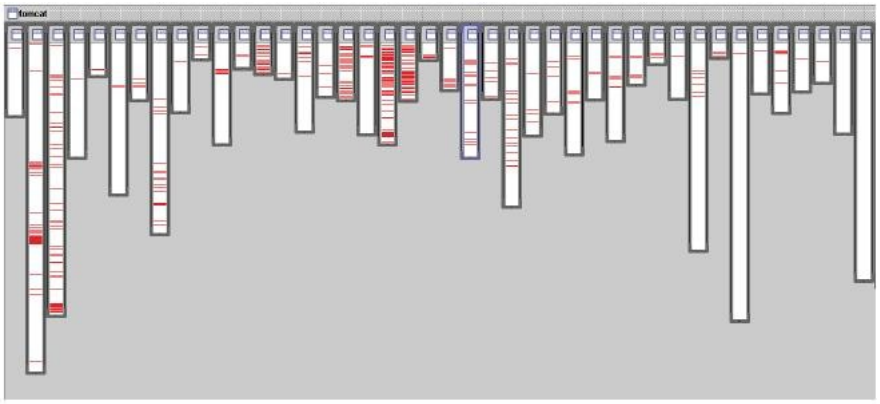
een fan zijn. Vanaf lijn 3 hebben we dan de workflow die door elke fan kan worden uitgevoerd. Zo zullen we eerst vragen of de fan wel geïnteresseerd is om te stemmen. Vervolgens filteren we op lijn 4 de groep op alle geïnteresseerden, waardoor de rest van de workflow alleen bij deze zal worden uitgevoerd. Hierna geven we aan alle geïnteresseerde fans een overzicht (lijn 5) van alle liedjes, waarna deze een selectie kan maken. Op lijn 6 hebben we dan een synchronisatie, waarbij de band de selectie van alle fans te zien krijgt, en hierna de playlist zal bepalen. Dit zal worden uitgevoerd wanneer het tijdstip op lijn 7 is bereikt. Hierna hebben we op lijn 9 een *CancellingBarrier* die zal wachten tot het bepaalde tijdstip, en hierna zal op lijn 10 de door de fans gekozen playlist worden getoond aan hen.

2.4 Aspectgeoriënteerd programmeren

Aspectgeoriënteerd programmeren is een techniek die wordt gebruikt voor het verbeteren van de *separation of concerns* [Dijkstra, 1982]. Hierbij gaan we een programma proberen opsplitsen in verschillende delen, waarbij elk deel een apart *concern* adresseert. Een *concern* wordt beschreven als een bepaald doel of concept waar de programmeur steeds rekening moet mee houden, en die een deel van het programma beschrijft. Dit brengt een aantal voordelen met zich mee. Eerst en vooral zorgt dit voor een vermindering van de complexiteit. Hiernaast vermindert het de impact wanneer er wijzigingen moet worden aangebracht. Het vereenvoudigt de integratie en moedigt het hergebruik van modules aan [Kiczales and Mezini, 2005].

We proberen dit te bereiken door gebruik te maken van verschillende modules, waarbij we code met dezelfde functionaliteit verzamelen in dezelfde module. Desondanks zijn er steeds functionaliteiten die niet kunnen verzameld worden in één module, maar verspreid zijn over verschillende modules. Dit wordt *crosscutting concerns* genoemd. Enkele symptomen hiervan zijn *scattering* en *tangling*. Bij *scattering* zal de code die tot één concern behoort, verspreid zijn over verschillende plaatsen, terwijl bij *tangling* code op één plaats verschillende concerns adresseert. Dit kan leiden tot redundante code, die moeilijk te begrijpen en te wijzigen is. Voorbeelden hiervan zijn logging en caching, waarbij functie oproepen wijd verspreid zijn over de code. De

volgende afbeelding is een voorbeeld van logging in een tomcat server. We zien de verschillende modules die de server implementeert. Alle rode strepen die in de modules voorkomen zijn logfuncties die worden uitgevoerd. We zien dat deze wijd verspreid zijn over alle modules, en niet kunnen verzameld worden in één module.



Figuur 2.6: Logging in Tomcat, afbeelding uit [Hilsdale et al., 2001].

Crosscutting concerns zijn onafscheidelijk verbonden met complexe systemen. Ze hebben een duidelijk doelstelling die zegt wat er moet gebeuren, en verschillende interactiepunten, die zeggen waar en wanneer het moet gebeuren. Aspectgeoriënteerd programmeren geeft de mogelijkheid om dit te bereiken. In een aspect wordt de toepasbaarheid en functionaliteit gedefinieerd. De toepasbaarheid bepaalt wanneer er iets moet worden uitgevoerd, en de functionaliteit zegt wat er dan moet worden uitgevoerd.

2.4.1 Joinpoint en pointcut model

Om te bepalen waar en wanneer er iets moet worden uitgevoerd, wordt er gebruik gemaakt van een jointpointmodel. Een joinpoint is een interessant punt in het programma waar *concerns* kunnen worden vastgesteld. Er zijn een groot aantal punten te vinden in een programma, zoals ondermeer wanneer er een bericht wordt verzonden, een methode wordt uitgevoerd, of een exceptie wordt geworpen. Joinpoints kunnen zowel statisch als dynamisch zijn. *Statische joinpoints* zullen specifieke punten in het programma zijn, terwijl *dynamische joinpoints* runtime events zijn. AspectJ [Kiczales et al., 2001] is een aspectgeoriënteerde programmeertaal die toelaat om aspecten te definiëren in Java. In de vorige listing hebben we een voorbeeld van een Java klasse, `Point`, die twee variabelen bevat (lijn 2), een constructor (lijn 4), en `setter` (lijn 6 en 7) en `getter` (lijn 9 en 10) methoden. Wanneer de velden van `Point` worden gewijzigd door een `setX` of `setY` wordt dit ook geprint.

Verschillende joinpoints die we kunnen identificeren in Java zijn methode oproepen, methode uitvoeringen, constructor oproepen, constructor uitvoeringen en velden opvragen en/of wijzigen. Om deze joinpoints te selecteren maken we gebruik van pointcuts. Pointcuts worden meestal

```
1  class Point {
2      private int x, y;
3
4      Point(int x, int y) { this.x = x; this.y = y; }
5
6      void setX(int x) { this.x = x;
7                         System.out.println("Field in point is changed");
8                       }
9      void setY(int y) { this.y = y;
10                        System.out.println("Field in point is changed");
11                          }
12
13     int getX() { return x; }
14     int getY() { return y; }
15 }
```

Listing 2.6: Point klasse in Java.

uitgedrukt door gebruik te maken van declaratieve pointcut talen. In AspectJ worden pointcuts voorgesteld door een aantal primitieven, zoals bijvoorbeeld *call* en *execution*. Wanneer we naar een voorbeeld van een pointcut kijken in listing 2.7, zien we op lijn 1 een pointcut die alle setters in de klasse Point selecteert. Wanneer we in een pointcut gebruik maken van * geldt dit als een *wildcard*, wat betekent dat de waarde op deze plaats niet van belang is. Bij de pointcut op lijn 1 is het return type dus niet van belang.

```
1 pointcut setters(): (call(* setX(int)) || call(* setY(int))) && within(Point);
```

Listing 2.7: Pointcut die alle set-methoden van Point selecteert.

2.4.2 Advice model en taal

Wanneer we geselecteerd hebben waar en wanneer er iets moet gebeuren in het programma, wordt het gedrag gedefinieerd dat moet worden toegevoegd aan het joinpoint. Dit wordt besproken in het advice model. Een advice type bepaalt specifiek waar het gedrag juist moet worden toegevoegd ten opzichte van het joinpoint. Bekende types van advices zijn *before*, *after* en *around*. Bij het *before* advice type zal het gedrag onmiddellijk voor het joinpoint worden toegevoegd, terwijl bij *after* dit onmiddellijk na het joinpoint gebeurt. Het *around* advice type voegt het gedrag toe rond het joinpoint. Elk advice wordt dus geassocieerd met een pointcut die het joinpoint selecteert waar het advice moet worden uitgevoerd. Wat er dan juist moet worden uitgevoerd, is gedefinieerd in de body van het advice. Deze body wordt meestal gedefinieerd door gebruik te maken van de taal waarin aspectgeoriënteerd programmeren wordt toegepast. Dit voorziet echter niet altijd de nodige functionaliteit waardoor de taal kan worden uitgebreid met enkele *constructs*. Een voorbeeld hiervan is *thisJoinPoint* dat kan worden gebruikt om zowel statische als dynamische informatie over het joinpoint te weten te komen, zoals bijvoorbeeld zijn argumenten. In listing 2.8 zien we een voorbeeld van een *after* advice. Juist nadat het pointcut `setters` bereikt is, wordt de body van het advice uitgevoerd, wat bepaald wordt door `after`. In dit voorbeeld wordt enkel geprint dat een veld in een punt gewijzigd is.

```
1 after(): setters() {
2       System.out.println("Field in point is changed");
3   }
```

Listing 2.8: Definitie van een advice in Java.

2.4.3 Aspect module model

De pointcuts en advices worden gedefinieerd in een aspect. Een aspect implementeert een cross-cutting concern door zowel de pointcuts als de advices te groeperen. In listing 2.9 zien we de definitie van een aspect die het pointcut en advice bevat die we al eerder besproken hebben, zie 2.4.1 en 2.4.2. Op lijn 3 zien we de definitie van het pointcut, gevolgd door het advice op lijn 5. Nadat we dit aspect gedefinieerd hebben kunnen we in listing 2.6 `system.out.println` verwijderen (lijn 7 en 9). Dit wordt nu verwezenlijkt door het aspect.

```
1 aspect tracePointSetter {
2
3 pointcut getters(): (call(* setX(int)) || call(* setY(int))) && within(Point);
4
5 after(): setters() {
6       System.out.println("Field in point is changed");
7   }
8 }
```

Listing 2.9: Aspect.

2.4.4 Aspectinstantiatie model

Het is mogelijk dat er in een aspect een staat wordt gedefinieerd die mee bepaalt wat een advice moet uitvoeren. Een voorbeeld hiervan is dat wanneer we een *around* advice hebben, dit advice maar een aantal keer keer mag worden uitgevoerd. Om dit te controleren houden we een teller bij in het aspect. Het advice wordt dus uitgevoerd afhankelijk van de waarde van deze teller. In dit geval moeten we eens kijken naar de manier waarop het aspect geïnstantieerd is.

Er zijn drie verschillende instantie strategieën. De strategie die altijd als standaard wordt gebruikt is de *singleton strategie*. Hier zal er slechts één aspect worden geïnitialiseerd in het volledige programma. In AspectJ kan men steeds toegang krijgen tot dit aspect door `aspectOf()`. Een tweede strategie is de *per-object strategie*. Hierbij moeten we het object specificeren waarvoor er een aspect nodig is. Dit kunnen we doen door `perthis(pointcut)` of `pertarget(pointcut)`. Wanneer het aangegeven pointcut bereikt is, wordt er een aspectinstantie aangemaakt. Per object is er één aspectinstantie. Het advice wordt alleen uitgevoerd wanneer er een aspectinstantie is voor het uitvoerende object. De derde mogelijke strategie is de *per-control-flow* strategie. Hierbij wordt een aspect geïnstantieerd doorheen de control flow van het pointcut. Dit pointcut wordt meegegeven aan `percflow(pointcut)` of `percflowbelow(pointcut)`. Wanneer dit pointcut bereikt is, of er onder, wordt er een instantie aange-

maakt. Ook hier wordt er alleen een advice uitgevoerd wanneer er een instantie bestaat.

2.4.5 Aspect compositie model

Wanneer we meerdere aspecten gedefinieerd hebben in hetzelfde programma, is het mogelijk dat er interactie problemen voorkomen tussen de verschillende aspecten [Tian et al., 2009]. Dit wordt veroorzaakt door verschillende advices die worden uitgevoerd bij hetzelfde joinpoint. De uitvoering van een aspect kan hierdoor de uitvoering van een ander aspect belemmeren. Een andere mogelijkheid is dat de volgorde waarin de advices worden uitgevoerd van belang is. Om dit te vermijden wordt er een voorrang voorzien tussen de verschillende aspecten, die bepaalt in welke volgorde meerdere aspecten worden uitgevoerd. In AspectJ kan de juiste volgorde eenvoudig bepaald worden door gebruikt te maken van de taal construct: `declare precedence Aspect1, Aspect2` waarbij `Aspect1` zal worden uitgevoerd voor `Aspect2`.

2.4.6 Aspect Weaving Model

Nadat we een aspect gedefinieerd hebben, moet dit nog gecombineerd worden met het basisprogramma zodat het volledige aspect kan worden uitgevoerd. Dit wordt het *weaving process* genoemd. Er zijn twee verschillende aanpakken: het statisch *weaven* en dynamisch *weaven*. Bij *statisch weaven* wordt het aspect en het basis programma gecombineerd voordat het programma wordt uitgevoerd. Dit kan zowel in de source code als in de byte code gebeuren. Het nadeel aan deze aanpak is dat, wanneer de code wordt uitgevoerd, de aspecten niet meer kunnen worden gewijzigd of verwijderd, en er ook geen nieuwe aspecten meer kunnen worden toegevoegd. Bij *dynamisch weaven* [Popovici et al., 2002] worden de aspecten gecombineerd met het basisprogramma wanneer dit reeds wordt uitgevoerd. Dit geeft de mogelijkheid om aspecten te wijzigen, verwijderen of toe te voegen wanneer de code reeds wordt uitgevoerd.

2.5 Aspectgeoriënteerd programmeren voor workflows

2.5.1 AO4BPEL

Er bestaan reeds verschillende talen die toelaten om aspecten te definiëren in een workflow taal. Eén daarvan is AO4BPEL [Charfi and Mezini, 2004], een uitbreiding op BPEL. Wanneer we kijken naar BPEL, constateren we dat dit enkele tekortkomingen met zich meebrengt. Eerst en vooral is er een gebrek aan modulariteit wanneer we verschillende crosscutting concerns modelleren. Veronderstel dat we een workflow hebben in BPEL die samengesteld is uit verschillende webservices, en we willen weten hoeveel keer een webservice wordt opgeroepen. De code die we moeten toevoegen om dit te bereiken is verspreid rond alle webservices in verschillende plaatsen in de workflow. Dit komt omdat BPEL geen enkele manier voorziet om op een modulaire wijze uit te drukken bij welke uitvoering van verschillende services er, bijvoorbeeld, extra informatie moet worden verzameld zoals de uitvoeringstijd. Een ander voorbeeld stelt zich wanneer we

verschillende webservices hebben die elk afzonderlijk een prijs moeten berekenen, zoals bijvoorbeeld de prijs voor een vlucht of een overnachting bij het plannen van een reis. De totale prijs wordt berekend door een andere webservice, maar deze kan pas worden uitgevoerd wanneer alle prijzen apart berekend zijn. Dit wordt ook wel een *business rule* genoemd in de samenstelling van webservices. Om dit te verwezenlijken zullen er in de workflow op verschillende plaatsen aanpassingen moeten gebeuren. Wanneer in dit geval maar twee webservices worden gebruikt, valt het aantal aanpassingen nog mee, maar wanneer er veel meer zijn, zal dit al snel onduidelijk worden. Deze *business rules* zijn voorbeelden van crosscutting concerns in webservice composities. Deze veranderen regelmatig maar kunnen niet goed worden gemodulariseerd in BPEL. Wanneer er nieuwe *rules* worden toegevoegd of gewijzigd, moeten er aanpassingen gebeuren doorheen het proces.

Een tweede tekortkoming in BPEL is dat we de samenstelling van webservices niet kunnen wijzigen wanneer deze reeds worden uitgevoerd. Wanneer er voor een bestaande webservice bijvoorbeeld een geüpdate versie is, is het onmogelijk in BPEL om deze te wijzigen wanneer de webservice reeds wordt uitgevoerd. Wanneer tijdens de uitvoering van het proces een webservice moet gewijzigd worden, moet het proces gestopt worden, de webservice aangepast, en het proces weer herstart. Het onderbreken van een proces brengt enkele nadelen met zich mee. Door het stoppen van de webservice kunnen er klanten verloren gaan. Hiernaast kan een samengesteld webservice complex zijn en de uitvoering ervan langdurig. Wanneer deze reeds vergevorderd is en onderbroken wordt, verliezen we alle bestaande resultaten, en moeten we volledig opnieuw beginnen. Een derde nadeel is dat het aanpassen van de samenstelling van webservices mogelijk verschillende excepties met zich meebrengt.

AO4BPEL is een aspectgeoriënteerde uitbreiding van BPEL die aspecten voorziet die kunnen worden toegepast tijdens de uitvoering van processen. BPEL processen bestaan uit een verzameling van activiteiten, waarbij elke BPEL activiteit een mogelijk joinpoint is. Omdat BPEL processen worden gedefinieerd in XML-files, zal XPath [Clark, 1999] gebruikt worden om een pointcut te selecteren. Doordat Xpath logische operatoren voorziet is het ook mogelijk om pointcuts te combineren. Net zoals in AspjectJ voorziet AO4BPEL een *before*, *after* en *around* advice. Een advice is een activiteit, gedefinieerd zoals in BPEL, die moet worden uitgevoerd voor, na of in plaats van een andere activiteit. Het *around* advice laat ons toe om een activiteit te vervangen door een andere. Een voorbeeld van een aspect in AO4BPEL kan gevonden worden in listing 2.10.

```
1  <aspect name="Counting">
2    <partnerLinks>
3      <partnerLink name="JavaExecWSLink" .../>
4    </partnerLinks>
5    <variables>
6      <variable name="invokeMethodRequest" .../>
7    </variables>
8    <pointcutandadvice type="after">
9      <pointcut name="Lufthansa Invocations">
10      //process//invoke[@portType ="LufthansaPT" and @operation ="
```

```
        searchFlight"]
11    </pointcut>
12    <advice>
13     <sequence>
14      <assign>
15        <copy>
16          <from>increaseCounter</from>
17          <to variable="invokeMethodRequest" part="methodName"/>
18        </copy>  ...
19      </assign>
20      <invoke partnerLink="JavaExecWSLink" portType="JavaExecPT"
21        operation="invokeMethod" inputVariable="invokeMethodRequest"/>
22     </sequence>
23    </advice>
24  </pointcutandadvice>
25 </aspect>
```

Listing 2.10: Voorbeeld aspect AO4BPEL.

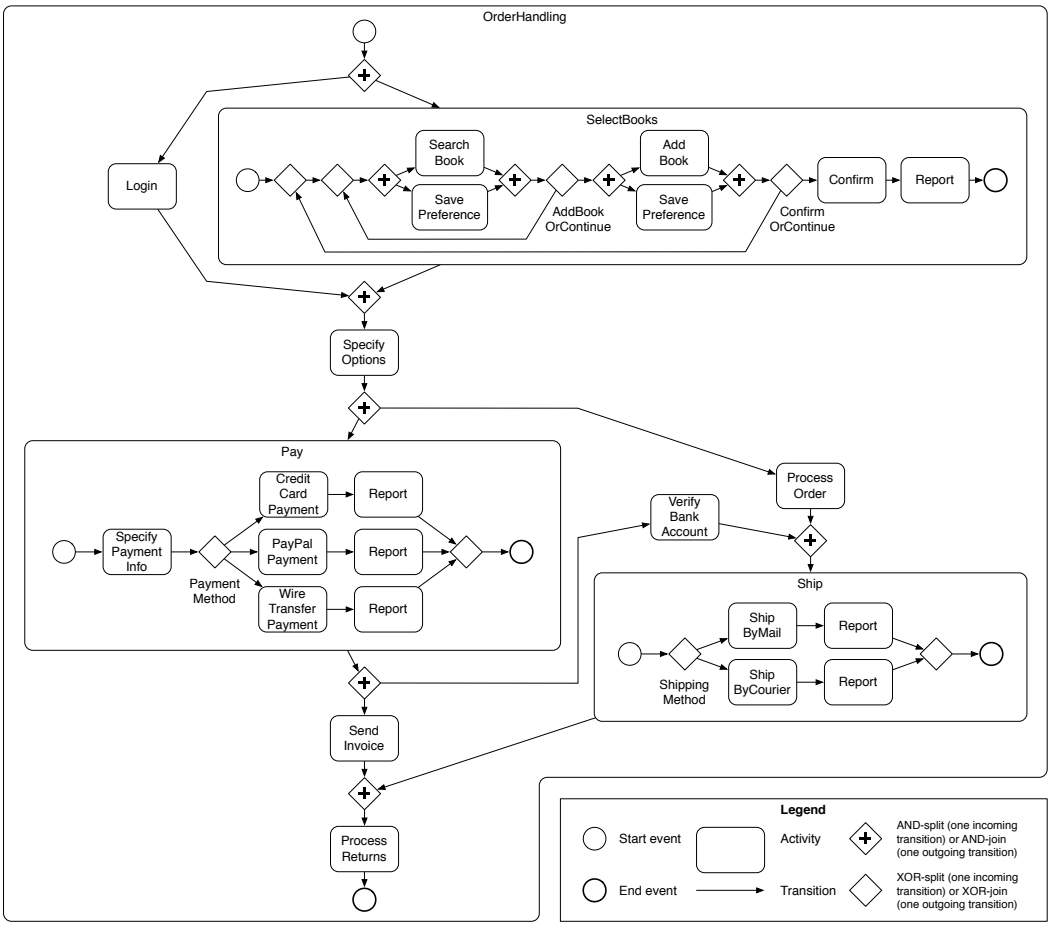
Dit aspect zal bijhouden hoeveel keer de operatie *searchFlight* van Lufthansa wordt geïnvokeerd. Het advice hiervan zal dus worden uitgevoerd iedere keer dat *searchFlight* wordt opgeroepen. Het totaal aantal invocaties wordt bijgehouden in een file. Omdat het niet mogelijk is files te openen, lezen, wijzigen en sluiten in BPEL, wordt de *Java code execution web service* gebruikt. Deze service wordt gedefinieerd op lijn 3. Op lijn 6 definiëren we een variabele, *invokeMethodRequest*, die gebruikt wordt in het advice. Vervolgens definiëren we op lijn 8 het advice type, namelijk *after*, waarna we op lijn 10 het pointcut definiëren. Dit zal voldaan zijn wanneer de operatie *searchFlight* wordt uitgevoerd op het poorttype *LufthansaPT*. In het advice definiëren we dan op lijn 16 een functie *increaseCounter*, die zal worden geïnvokeerd door de *Java execution web service* op lijn 20.

2.5.2 Unify

De meeste workflowtalen voorzien een beperkt aantal modularisatie mechanismen. Hierdoor worden verschillende concerns verspreid over de gehele workflow wat het design en de herbruikbaarheid van de workflows beperkt. Dit kan worden verholpen door gebruik te maken van het Unify framework [Joncheere and Van Der Straeten, 2011], welke uniforme modularisatie van workflows ondersteunt door alle workflow concerns, inclusief de crosscutting concerns, apart van elkaar te specificeren.

Het doel van Unify is het vereenvoudigen van onafhankelijke evolutie en hergebruik van alle workflow concerns. Hierbij ligt de focus dus niet enkel op crosscutting concerns. Om dit te bereiken wordt er een beter modularisatie mechanisme voorzien. Het laat ons toe om verschillende workflowconcerns met elkaar te verbinden op een workflow specifieke manier, namelijk door een connector mechanisme die een aantal concern connectiepatronen bevat. Dit connector mechanisme is gedefinieerd in functie van een algemeen, uitbreidbaar basis meta-model. Unify

definieert een duidelijke semantiek voor haar modularisatie mechanismen. De implementatie van Unify kan worden gebruikt als een aparte workflow engine, of als een pre-processor die verenigbaar is met bestaande workflow engines.



Figuur 2.7: Voorbeeld workflow orderafhandeling., afbeelding uit [Joncheere and Van Der Straeten, 2011]

In figuur 2.7 zien we een voorbeeld van een workflow die een order zal afhandelen. Nadat de workflow gestart is, zal de *Login* en *SelectBooks* activiteit in parallel worden uitgevoerd. Hierna wordt de *SpecifyOptions* activiteit uitgevoerd, waarna we opnieuw een parallelle split hebben. Deze zal opnieuw twee branches bevatten, waarvan één de *Pay* en *SendInvoice* activiteiten bevat, en de ander de *ProcessOrder* en *Ship* activiteiten. Deze worden gesynchroniseerd door de *VerifyBankAccount* activiteit. Tenslotte wordt hierna nog de *ProcessReturns* activiteit uitgevoerd, waarna de workflow eindigt.

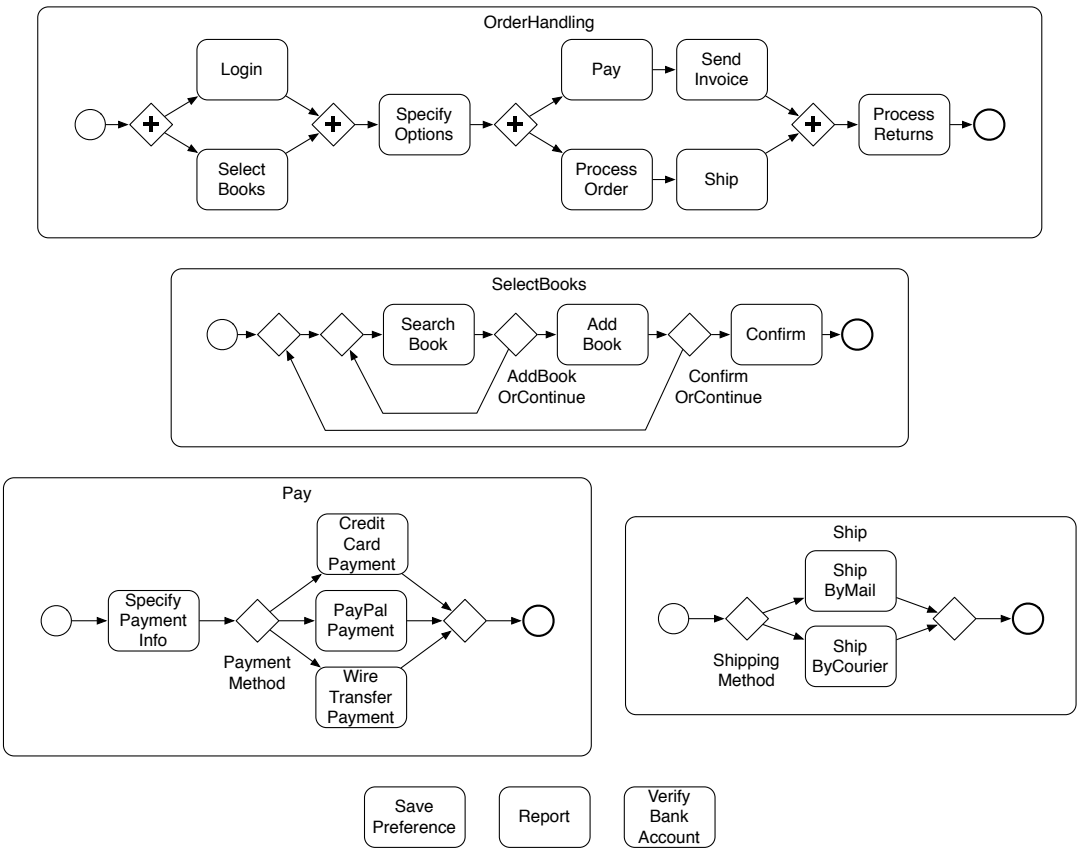
Uit dit voorbeeld kunnen we nu verschillende concerns identificeren. Zo is het duidelijk dat het belangrijkste concern order afhandeling is. Dit concern is hiërarchisch opgesplitst in sub con-

cerns, zoals ondermeer boekselectie en betaling, door gebruik te maken van de samengestelde activiteitconstruct. Bij *separation of concerns* gaan we de applicatie zoveel mogelijk proberen op te splitsen zodat elk concern apart kan worden gemanipuleerd. Dit is echter niet altijd mogelijk, verschillende workflowtalen laten niet toe om workflows op te splitsen in verschillende modules. Hiernaast moeten we ook nog de verschillende workflowconcerns met elkaar kunnen verbinden. In bestaande workflowtalen is dit echter beperkt, hier kunnen we alleen in een mainworkflow specificeren dat er op een bepaald ogenblik een subworkflow moet worden uitgevoerd. Dit moet reeds vastliggen bij het ontwerp van de workflow en is later nog moeilijk te wijzigen. Door te voorzien dat de keuze, voor het bepalen van welke subworkflow er moet worden uitgevoerd, kan worden uitgesteld zal er een hogere mate van separation of concerns zijn.

Wanneer sommige concerns moeten worden toegepast op verschillende plaatsen in de workflow is het niet mogelijk om dit op een modulaire manier te doen, en is er nood aan een tweede soort connectie. Wanneer we naar het voorbeeld kijken in figuur 2.7, merken we op dat de *report* activiteit zich bevindt op verschillende plaatsen in de workflow. Omdat de subworkflow construct hier niet kan worden toegepast, subworkflows worden expliciet opgeroepen vanuit de main workflow, wordt er gebruik gemaakt van een aspectgeoriënteerde oplossing. Dit laat ons toe om specifieke crosscutting concerns te definiëren in specifieke aspecten. Hiernaast laat dit ons ook toe om een specifiek workflowfragment, een advice, uit te voeren voor, na of rond een verzameling van activiteiten. Dit zijn echter algemene aspectgeoriënteerde technieken die niet alle mogelijkheden in workflows adresseren, zoals ondermeer het uitvoeren van een advice in parallel.

Wanneer we nu een workflow ontwikkelen door gebruik te maken van Unify moeten we eerst de verschillende concerns identificeren. In het vorige voorbeeld kunnen we de volgende concerns identificeren: order afhandeling, selecteren van boeken, verschepen, betalen, voorkeur opslagen, bank account verifiëren en rapporteren. Deze concerns zijn weergegeven in figuur 2.8. Unify stimuleert het implementeren van workflow concerns als gescheiden modules. Dit kan verwezenlijkt worden door gebruik te maken van de *samengestelde activiteit* construct. Het voordeel van workflowconcerns te definiëren als gescheiden samengestelde activiteiten is dat verschillende delen van een concern niet langer verspreid zijn doorheen de workflow, wat ons een betere scheiding van concerns geeft. Wanneer de verschillende concerns geïdentificeerd en geïmplementeerd zijn, kunnen we de connectie tussen deze concerns bepalen. Er zijn twee hoofdcategorieën van connecties tussen concerns. De verwachte concernconnectie is een connectie waarbij reeds bij het design duidelijk is dat deze een connectie zal hebben met een andere concern op een bepaald punt in de uitvoering van de workflow. Bij de onverwachte concernconnectie is dit niet op voorhand het geval.

Unify voorziet connector constructs die beide connecties zullen verwezenlijken. De *before connector* wordt gebruikt om te specificeren dat een advice activiteit moet worden toegevoegd voor elk van de joinpoints die gespecificeerd zijn door de pointcut van een connector. De *replace connector* wordt gebruikt om een activiteit aangeduid door een pointcut te vervangen door een andere activiteit. De *parallelle connector* kan worden gebruikt om aan te geven dat de advice



Figuur 2.8: Onafhankelijk gespecificeerde workflow concerns., afbeelding uit [Joncheere and Van Der Straeten, 2011]

activiteit in de basis workflow moet worden geïntroduceerd in parallel met elk joinpoint gespecificeerd door het joinpointFragmentPointcut. Dit zijn slechts enkele van de connectors die Unify voorziet.

Unify is zo ontwikkeld dat het toepasbaar is op een reeks van concrete workflowtalen wanneer deze aan een aantal basisveronderstellingen voldoen. Deze worden uitgedrukt in een meta-model voor de workflowconcerns. Het meta-model laat ons ook toe om arbitraire workflows uit te drukken, welke workflows zijn die niet beperkt zijn tot een aantal control flow patronen. Daarom is het meta-model ook verenigbaar met meer beperkte workflows zoals gestructureerde workflows. Omdat in Unify meer de nadruk wordt gelegd op de expressiviteit van het modularisatie mechanisme dan op de expressiviteit van de individuele modules, streeft het meta-model niet naar de voorziening van alle beschikbare control flow patronen. Doordat alle basis control flow patronen voorzien zijn kunnen de meeste workflows worden uitgedrukt.

Hoofdstuk 3

Samenwerking van workflows op een aspectgeoriënteerde manier

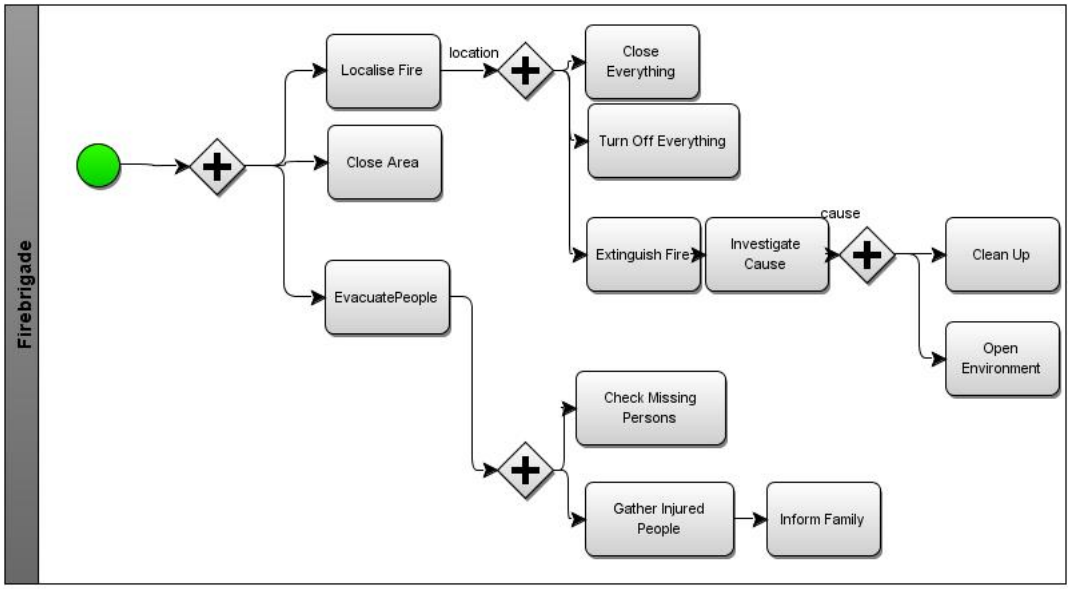
In dit hoofdstuk bespreken we de aanpak waarmee we ons probleem oplossen. Eerst overlopen we de verschillende vereisten waaraan onze aanpak moet voldoen. Daarna bespreken we de voorgestelde oplossing, die aspectgeoriënteerd is, en overlopen we het joinpoint model, de pointcut taal, het advice- en aspectmodel. Tenslotte bespreken we nog welke conflicten er kunnen plaatsvinden wanneer aspecten worden samengesteld.

3.1 Motivatie en vereisten

In deze sectie introduceren we eerst een voorbeeldapplicatie die geldt als een motivatie voor het onderzoek dat we in deze thesis beschrijven. Het scenario dat we beschrijven focust zich op de orchestratie van verschillende reddingsteams in een rampgebied. Veronderstel dat een grote brand plaatsvindt in een shoppingcentrum, waarbij de brand is uitgebreid naar nabijgelegen appartemenentsgebouwen. Door de omvang van deze ramp worden allerlei reddingswerkers, zoals de brandweer, politie, medische dienst,... naar dit gebied gestuurd. Om optimaal te functioneren dienen deze verschillende teams samen te werken. Zo is de brandweer verantwoordelijk voor het afsluiten van de omgeving, het in veiligheid brengen van alle personen, en uiteraard het blussen van de brand. Hiervoor zal men eerst de brand moeten lokaliseren, waarna men alle nutsvoorzieningen, ramen en deuren afsluit, en start met het blussen van de brand. De medische diensten zullen alle gewonde personen controleren, en indien mogelijk eerste hulp ter plaatse toedienen, ofwel het slachtoffer evacueren naar het ziekenhuis. Wanneer de brand aangestoken is, zal de politie eerst een onderzoek openen en getuigen ondervragen, wanneer dit niet het geval is kan men onmiddellijk de pers inlichten.

Er is een aparte workflow voor de brandweer (figuur 3.1), de politie (figuur 3.3) en de medische diensten (figuur 3.2). De brandweer moet eerst drie taken in parallel uitvoeren, namelijk het vuur lokaliseren, alle personen evacueren en de omgeving afzetten zodat onbevoegde personen geen toegang hebben tot de gevarenzone. Dit kan gemodelleerd worden door gebruik te maken

van het parallelle split patroon. In figuur 3.1 zien we dat de workflow van de brandweer start met dit patroon. Wanneer het vuur gelokaliseerd is, wordt de locatie toegevoegd aan de omgeving van de workflow, en kan men starten met het bestrijden van de brand. Hier gaan we opnieuw enkele taken in parallel uitvoeren, waaronder alle ramen en deuren sluiten, gas en elektriciteit afsluiten en starten met het blussen van het vuur. Wanneer het vuur geblust is, gaat men de oorzaak bepalen en deze informatie toevoegen aan de workflow. Wanneer men alle personen geëvacueerd heeft, gaat men alle gewonde personen op één plaats verzamelen, en controleren of er nog personen vermist zijn.

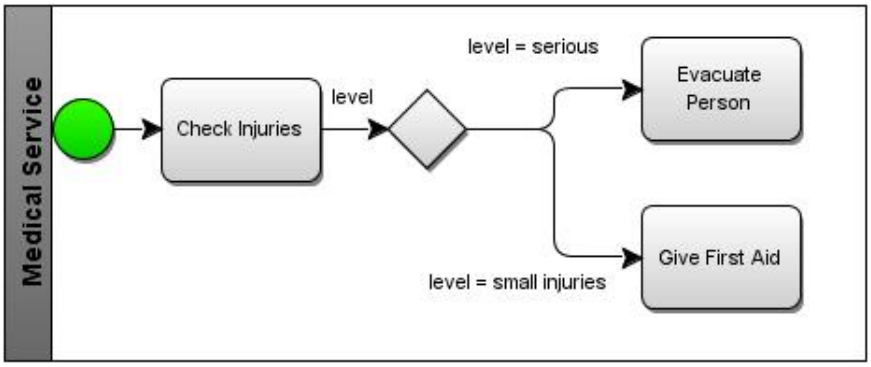


Figuur 3.1: Voorbeeld workflow brandweer.

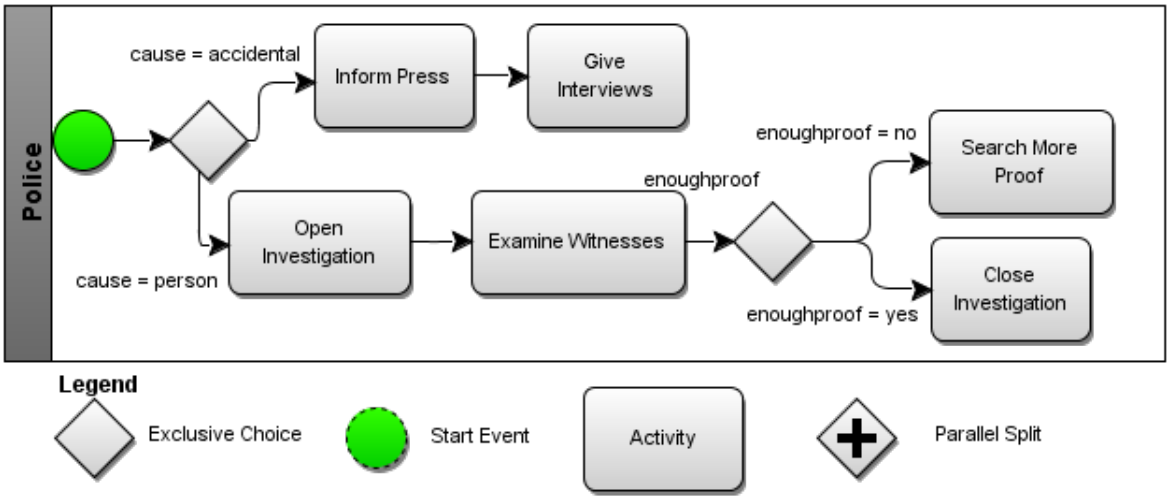
De medische diensten kunnen hun workflow starten wanneer alle gewonde personen op één plaats verzameld zijn. Deze zullen dan één voor één medisch onderzocht worden, waarbij eerst de graad van hun verwondingen zal bepaald worden. Als een persoon lichtgewond is, wordt deze ter plaatse verzorgd, terwijl zwaargewonden onmiddellijk naar het ziekenhuis worden gebracht voor verdere verzorging.

Als derde workflow hebben we nog deze van het politieteam. Hun taak zal afhankelijk zijn van de oorzaak van de brand. Als deze is aangestoken, zal men eerst een onderzoek starten en daarna getuigen verhoren om meer te weten te komen over de precieze oorzaak hiervan. Wanneer de brand niet is aangestoken, wordt de pers onmiddellijk ingelicht. Deze taak, afhankelijk van de oorzaak, kan worden gemodelleerd door gebruik te maken van een *exclusive choice* patroon.

Wanneer we het voorbeeld van dichterbij bekijken, merken we op dat deze verschillende workflows niet met elkaar verbonden zijn, maar er wel een vorm van samenwerking vereist is. De



Figuur 3.2: Voorbeeld workflow medische dienst.



Figuur 3.3: Voorbeeld workflow politie.

workflow van de medische diensten kan maar starten wanneer de brandweer alle gewonde personen op één plaats verzameld heeft. Hetzelfde geldt voor de politie, die hun job pas kunnen uitvoeren wanneer de oorzaak door de brandweer bepaald is. We zien dus dat deze workflows pas starten wanneer een aangeduide activiteit is afgerond. Hieruit kunnen we volgende vereiste afleiden:

Vereiste 1. Een manier voorzien om een activiteit in de workflow aan te duiden.

Naast het aanduiden van een activiteit is het ook mogelijk dat een andere workflow moet starten wanneer een bepaalde variabele is toegevoegd aan een workflow door een activiteit. Als we naar het voorbeeld kijken, kunnen we bepalen dat de workflow van de politie moet starten wanneer de oorzaak, die bepaald wordt door de brandweer in de '`onderzoek oorzaak`' activiteit, gekend is. Dit kan ook het geval zijn wanneer een variabele wijzigt. Als we bijvoorbeeld in de

workflowomgeving een variabele '`brandweercommandant`' hebben, die de naam van deze bevat, kunnen we een workflow definiëren die alle andere hulpdiensten waarschuwt indien het leiderschap tijdelijk wordt overgenomen door een andere persoon, omdat bijvoorbeeld de brandweercommandant al lange tijd aanwezig is en nood heeft aan rust. Hieruit kunnen we volgende vereiste afleiden:

Vereiste 2. Een manier voorzien om data in onze workflow aan te duiden.

We hebben reeds het geval besproken dat er iets moet gebeuren wanneer een bepaalde activiteit is uitgevoerd, maar dit kan nog worden uitgebreid. In het voorbeeld dat we tonen in figuur 3.1 zien we dat de workflow van de brandweer start met een parallelle split, en dus meerdere activiteiten tegelijk kunnen worden uitgevoerd. Veronderstel dat we nu een vierde workflow hebben, namelijk die van de pers. De pers krijgt pas toegang tot de locatie wanneer het vuur geblust is en de familieleden zijn verwittigd. Daarom moeten we ook de mogelijkheid hebben om meerdere activiteiten te combineren. Hieruit kunnen we volgende vereiste afleiden:

Vereiste 3. Een manier voorzien om een combinatie van verschillende activiteiten aan te duiden.

Deze vorige 3 vereisten worden gebruikt om aan te duiden wanneer een workflow aan bepaalde voorwaarden heeft voldaan. Nu moeten we kunnen bepalen wat er moet gebeuren wanneer deze voorwaarden voldaan zijn. Bijvoorbeeld wanneer de brandweer de oorzaak van de brand bepaald heeft, moet de workflow van de politie worden opgestart. Hetzelfde geldt voor de medische diensten. Deze workflow kan worden gestart wanneer alle gewonde personen zijn verzameld op één plaats. Zo zal er een andere workflow gestart worden wanneer een workflow een bepaald punt bereikt heeft. Hieruit kunnen we volgende vereiste afleiden:

Vereiste 4. Een manier voorzien om extra activiteiten uit te voeren .

Wanneer we verschillende workflows hebben die in dezelfde omgeving worden uitgevoerd, is het mogelijk dat hetgeen een workflow moet uitvoeren, afhankelijk is van het resultaat van een andere workflow. Wanneer we naar workflows van de brandweer en de politie kijken, zien we dat de politie pas mag starten met hun werk wanneer de oorzaak bepaald is, wat wordt gedaan door de brandweer. Wanneer de oorzaak nu opzettelijk is, bijvoorbeeld de brand is aangestoken, en er zijn niet genoeg aanwijzingen over wie de dader zou kunnen zijn, dan kan de politie aan de brandweer vragen om extra aanwijzingen te zoeken. We moeten dus extra activiteiten aan andere workflows kunnen toekennen door de structuur van de andere workflow uit te breiden of te wijzigen. Een voorbeeld hiervan is het toevoegen van een extra activiteit aan de workflow. Hieruit kunnen we volgende vereiste afleiden:

Vereiste 5. Een manier voorzien om structuur van andere workflows te wijzigen.

Doordat de verschillende workflows afhankelijk van elkaar kunnen zijn, is het mogelijk dat deze opnieuw met elkaar moeten gesynchroniseerd worden. Het is mogelijk dat de brandweer al zijn taken uitgevoerd heeft, maar dat de politie nog steeds bezig is met het onderzoeken van de brand. Dan mag de brandweer nog niet terugkeren naar de kazerne maar moet deze nog steeds ter plaatse beschikbaar zijn voor het geval deze nog een extra activiteit moet uitvoeren in opdracht van de politie, bijvoorbeeld extra bewijsmateriaal zoeken. Daarom moeten we een synchronisatiepunt toevoegen op het einde van de workflow van de brandweer. Deze zal op dit punt wachten totdat de politie zijn taken heeft uitgevoerd. Hieruit kunnen we volgende vereiste afleiden:

Vereiste 6. Een manier voorzien om nieuwe synchronisatiepunten toe te voegen in de workflow.

Doordat de workflows worden uitgevoerd in een nomadisch netwerk, moeten we rekening houden met een grote kans op netwerkdisconnecties. Hierdoor is het mogelijk dat het resultaat van een activiteit niet bekend gemaakt wordt of een activiteit niet wordt voltooid. Daardoor kunnen andere workflows die hierop rekenen niet verdergaan of een bepaalde actie niet uitvoeren. Wanneer de politie wacht op de oorzaak van de brand, maar de brandweer gedisconnecteerd raakt, wordt de oorzaak niet ontvangen, maar men krijgt wel een bericht dat er een disconnectie heeft plaatsgevonden. Dit kan tot gevolg hebben dat de politie de oorzaak niet ontvangt, terwijl de brandweer deze al lang bepaald heeft. Daarom moeten we dus rekening houden met mogelijke disconnecties, en eventueel een alternatieve actie definiëren. We kunnen bijvoorbeeld bepalen dat wanneer de politie na een bepaalde tijd de oorzaak nog steeds niet weet, een politieagent op onderzoek wordt uitgestuurd. Hieruit kunnen we volgende vereiste afleiden:

Vereiste 7. Voorzien van foutafhandeling.

3.2 Overzicht van de aanpak

Om deze problemen op te lossen maken we een uitbreiding op NOW. Deze uitbreiding zal voldoen aan alle vereisten die we zojuist hebben overlopen. NOW [Philips et al., 2013] is een nomadische workflowtaal waarbij workflows kunnen worden gedefinieerd op een vaste infrastructuur, en waarbij verschillende services de taken van deze workflow zullen uitvoeren. We hebben voor NOW gekozen omdat dit ons toelaat op een eenvoudige manier workflows te definiëren in nomadische netwerken, en reeds foutafhandeling voorziet. Onze oplossing is gebaseerd op *aspectgeoriënteerd programmeren* [Kiczales et al., 1997]. We kiezen hiervoor omdat dit ons op een eenvoudige manier toelaat om *inversion of control* [Pisa, 2009] toe te passen. Dit geeft ons de mogelijkheid om op één plaats te definiëren waar er iets moet gebeuren in de workflow, alsook wat er op deze plaatsen moet gebeuren. Het voordeel hiervan is dat we niet al de workflows op verschillende plaatsen moeten wijzigen, wat ons een verhoogde mate van *separation of concerns* [Parnas, 1972] geeft. Omdat deze workflows worden uitgevoerd in nomadische netwerken, hebben we op voorhand geen weet van alle mogelijke workflows die beschikbaar kunnen zijn.

Daarom is het ook onmogelijk om al deze workflows op voorhand te wijzigen. Door gebruik te maken van aspecten, kunnen deze makkelijker worden toegepast op nieuwe workflows die in het nomadisch netwerk ontdekt worden. We kunnen ook onmogelijk alle workflows in één grote workflow combineren omdat deze workflows niet altijd samen worden uitgevoerd. Wanneer bijvoorbeeld een politieagent het verkeer staat te regelen, moet de workflow van de brandweer niet worden uitgevoerd op de vaste infrastructuur van de politie. Indien de workflow van de politie moet worden aangepast, moeten we dit aanpassen op alle vaste infrastructuren waarop deze workflow wordt uitgevoerd. Wanneer de workflow van de politie enkel wordt uitgevoerd op zijn eigen vaste infrastructuur, is het veel eenvoudiger om deze aan te passen.

We gaan een aspect definiëren welke een pointcut en advice zal bevatten. In workflows kunnen we verschillende types van joinpoints identificeren (vereiste 1 en 2), die we kunnen beschrijven aan de hand van pointcuts. Voorts kunnen we ook een combinatie nemen van joinpoints (vereiste 3) door gebruik te maken van booleaanse operatoren. De acties die moeten worden uitgevoerd, zullen we definiëren aan de hand van advices (vereiste 4, 5 en 6). We voorzien faling afhandelingen wanneer een advice niet binnen een bepaalde tijd is uitgevoerd, en wanneer bepaalde services gedisconnecteerd zijn (vereiste 7). Om dit te beschrijven maken we ook gebruik van pointcuts. In de volgende secties bespreken we elk van deze onderdelen meer in detail.

3.3 Joinpoint model en pointcut taal

Aangezien we gebruik maken van een aspectgeoriënteerde oplossing hebben we een joinpoint model nodig. Wanneer we naar bestaande aspectgeoriënteerde programmeertalen kijken, zoals AspectJ [Kiczales et al., 2001], zien we dat deze vooral op methoden en variabelen gericht zijn. Zo zijn er joinpoints op de oproepen, ontvangers en uitvoeringen van methoden en constructors. Hiernaast zijn er ook nog joinpoints bij het lezen of wijzigen van een variabele van een object, en de initialisatie van een object en een klasse. Tenslotte is er ook nog een joinpoint wanneer er een exception handler wordt opgeroepen. Deze joinpoints zijn voornamelijk gericht op objecten en methoden, aangezien Java een objectgeoriënteerde programmeertaal is en AspectJ hierop wordt toegepast.

Laten we nu eens kijken naar een aspectgeörienteerde taal specifiek voor workflows, zoals AO4BPEL [Charfi and Mezini, 2007]. Wanneer we naar de joinpoints van deze taal kijken, merken we meteen op dat deze verschillen ten opzichte van AspectJ. De joinpoints voorzien door AO4BPEL leggen de nadruk op activiteiten van workflows, terwijl in AspectJ voornamelijk gefocust wordt op objecten en methoden. Zo zijn de joinpoints in AO4BPEL verschillende workflowactiviteiten. Deze kunnen worden beschreven door hun type en attributen van de activiteit. AO4BPEL zal dus een activiteitgedreven joinpoint model hebben dat specifiek is voor workflows.

Wanneer we nu naar de vereisten in sectie 3.1 kijken, zien we dat we in NOW activiteiten en data moeten kunnen selecteren. We moeten joinpoints kunnen identificeren op ondermeer de activiteiten die worden uitgevoerd in de workflow. Deze activiteiten kunnen worden geïdentificeerd

Joinpoint	Beschrijving
Activiteit	activiteit die in een workflow wordt uitgevoerd
Workflowpatronen	alle patronen die NOW bevat, zoals bijvoorbeeld een parallelle split
Data	data die wordt gemanipuleerd in de workflow
Disconnecties	disconnecties tussen workflows en services

Tabel 3.1: Joinpoints.

aan de hand van hun naam. Naast activiteiten is er ook een joinpoint op de data die wordt gemanipuleerd in de workflow. Dit joinpoint kunnen we identificeren aan de naam van de variabele. Omdat de workflows worden uitgevoerd in nomadische netwerken, moeten we ook rekening houden met disconnecties van uitvoerders of disconnecties tussen verschillende workflows. Daarom zullen disconnecties in ons netwerk ook een joinpoint vormen.

De pointcuts die we definiëren zijn expressies die ons toelaten de joinpoints aan te duiden in de workflows. Wanneer we naar AspectJ kijken, merken we op dat er enkele belangrijke soorten van verschillende pointcuts voorzien zijn. Eerst zijn er de pointcuts die functies selecteren zoals bijvoorbeeld de pointcuts *call* en *execution*. Een tweede soort van pointcuts werkt op variabelen. Zo zullen de pointcuts *get* en *set* voldaan zijn wanneer er respectievelijk een variabele wordt gelezen en gewijzigd. Hiernaast hebben we ook pointcuts die voldaan zijn bij een bepaalde initialisatie, zoals ondermeer *initialization* en *preinitialization*. Voorts kunnen we ook nog een object dat een target is selecteren als een pointcut, alsook annotaties die in de code werden toegevoegd. Al deze pointcuts kunnen gecombineerd worden door gebruik te maken van booleaanse operatoren, zoals `and(&&)` en `or(||)`.

De pointcuts in AO4BPEL worden gedefinieerd door gebruik te maken van XPath, een querytaal voor XML documenten. Aangezien BPEL volledig is gedefinieerd in XML kan XPath zonder problemen worden toegepast in AO4BPEL. Zo ziet een poincut in XPath er als volgt uit:

```
1 <pointcut name="Lufthansa Invocations">
2 //process//invoke[@portType="LufthansaPT" and @operation="searchFlight"]
3 </pointcut>
```

Listing 3.1: Voorbeeld pointcut AO4BPEL.

In listing 3.1 zien we op lijn 1 dat we eerst het pointcut een naam geven. Vervolgens wordt op lijn 2 het pointcut gedefinieerd. Het pointcut is voldaan wanneer er een proces wordt geïnvokeerd met poorttype "`LufthansaPT`" en met operatie "`searchFlight`".

Nu kunnen we verschillende pointcuts voor NOW definiëren die de mogelijkheid geven om joinpoints te selecteren. Eerst hebben we een pointcut die een activiteit aanduidt in een workflow. De pointcut `PointcutMethod("InvestigateCause")` is een pointcut die de activiteit "`InvestigateCause`" aanduidt die voorkomt in het voorbeeld van figuur 3.1. Dit pointcut

kan ook gebruikt worden om activiteiten aan te duiden die aan een patroon voldoen door activiteiten te vergelijken met reguliere expressies. Zo zal `PointcutMethodContains(".*Cause.*")` elke activiteit in de workflow aanduiden die '`Cause`' bevat in zijn naam. Een ander voorbeeld is de pointcut `PointcutMethodPattern("investigate.*Cause")`, die alle activiteiten aanduidt die beginnen met '`investigate`' en eindigen met '`Cause`'. Twee mogelijke activiteiten die hiermee zouden overeenkomen zijn "`investigateFireCause`" en "`investigateCrashCause`". Dit pointcut laten ons toe om aan vereiste 1 te voldoen.

Naast pointcuts op activiteiten, hebben we ook een pointcut die werkt op data. Data die de eerste keer wordt toegevoegd aan de workflow kan worden geïdentificeerd door de pointcut `PointcutDataAdded("Cause")`, waar '`Cause`' de naam van de variabele is die wordt toegevoegd aan de omgeving van de workflow. Dit kan gebruikt worden om de workflow van de politie te starten wanneer de brandweer de oorzaak van de brand bepaald heeft, en deze als variabele '`Cause`' heeft toegevoegd aan de omgeving. Hiernaast hebben we ook nog de pointcut `PointcutDataChanged("Cause")`, die voldaan is iedere keer wanneer de variabele '`Cause`' is gewijzigd, en de pointcut `PointcutData("Cause")`, die de vorige twee pointcuts combineert. Deze is dus voldaan wanneer de data wordt toegevoegd of gewijzigd. Het gebruik van reguliere expressies is ook toegelaten in deze drie pointcuts. Zo is de pointcut `PointcutDataAdded(".*Cause")` voldaan bij alle data die wordt toegevoegd en eindigt op '`Cause`'. Deze pointcuts laten ons toe om aan vereiste 2 te voldoen.

De vorige beschreven pointcuts dienen voornamelijk om statische plaatsen aan te duiden in de workflow. Aangezien deze workflows worden uitgevoerd in nomadische netwerken, moeten we ook rekening houden met verschillende falingen die kunnen voorkomen, zoals disconnecties tussen de uitvoerders en het netwerk, disconnecties tussen de verschillende workflows, of timeouts indien bepaalde activiteiten niet binnen een voorziene tijd worden uitgevoerd. Om deze falingen te kunnen beschrijven voorzien we nog enkele andere pointcuts. We hebben een pointcut `PointcutTimeout(time, startTime)` die twee argumenten meekrijgt. Het `time` argument geeft aan na hoelang de timeout verlopen is, terwijl het `startTime` argument aangeeft vanaf wanneer de timeout mag beginnen tellen. Dit argument kan een tijdsobject zijn met een specifiek tijdstip, ofwel een andere pointcut. De timeout zal dan beginnen met aftellen wanneer dit pointcut bereikt is in de workflow. Wanneer de tijd verstreken is, zal de pointcut voldaan zijn en het advice worden uitgevoerd. Hiernaast hebben we ook een pointcut `PointcutDisconnected(`ExtinguishService)`, die voltooid is wanneer de `ExtinguishService` gedisconnecteerd is. De verschillende patronen in NOW kunnen we selecteren door het pointcut `PointcutPattern(patternType)`. Hierbij is patternType het type van een patroon, zoals bijvoorbeeld ParallelSplit.

Aangezien alle vorige pointcuts maar één plaats in een workflow aanduiden, willen we ook meerdere plaatsen kunnen combineren. Wanneer we kijken naar het rampscenario, besproken in sectie 3.1, kunnen we de workflow van de politie pas starten wanneer de brandweer de oorzaak heeft bepaald, en alle personen heeft geëvacueerd. Hier moeten dus twee pointcuts voldaan zijn voor het advice mag worden uitgevoerd. Daarom definiëren we een intersectie van pointcuts, `in-`

Pointcut	Beschrijving
PointcutMethod	pointcut op activiteit waarvan naam bepaald woord bevat of overeenkomt met reguliere expressie
PointcutDataAdded	pointcut op toevoeging van data
PointcutDataChanged	pointcut op het wijzigen van data
PointcutData	pointcut op het manipuleren van data
PointcutTimeOut	pointcut wanneer er een timeout is
PointcutDisconnected	pointcut op disconnectie
PointcutPattern	pointcut op patronen

Tabel 3.2: Pointcuts overzicht.

`tersection(PointcutMethod("InvestigateCause"), PointcutMethod("EvacuatePeople"))`. Wanneer beide pointcuts voldaan zijn, is deze intersection pointcut ook voldaan. We kunnen ook definiëren dat het advice mag worden uitgevoerd elke keer wanneer één van de pointcuts is voldaan. Dit kan gedefinieerd worden door `union(PointcutMethod("InvestigateCause"), PointcutMethod("EvacuatePeople"))`. Nu zal het advice worden uitgevoerd wanneer ofwel het eerste pointcut, ofwel het tweede pointcut is voldaan. Het is ook mogelijk om deze pointcuts te nesten. Zo kan een union deel uitmaken als argument van een intersection. Deze twee pointcuts laten ons toe om aan vereiste 3 te voldoen.

Deze union kunnen we ook eenvoudig gebruiken in combinatie met de pointcuts voor falingen. Zo kunnen we een union pointcut nemen die twee pointcuts bevat, een `PointcutMethod` en een `PointcutDisconnected`. Union zorgt ervoor dat het advice wordt uitgevoerd wanneer deze activiteit voltooid is. Wanneer we zeker en vast willen dat het advice wordt uitgevoerd, wordt dit gegarandeerd door het disconnectie pointcut. Dit zorgt ervoor dat ook wanneer er een bepaalde disconnectie is, we er zeker van zijn dat het advice wordt uitgevoerd. Deze twee pointcuts helpen ons bij het vervullen van vereiste 7.

3.4 Advice model en taal

In deze sectie bespreken we welke types van advices we allemaal hebben en wanneer deze kunnen worden uitgevoerd. Aangezien we aspecten toepassen op workflows in nomadische netwerken, voorzien we enkele specifieke advices.

Laten we eerst eens kijken naar bestaande soorten van advices die aspectgeoriënteerde talen voorzien. AspectJ [Kiczales et al., 2001] voorziet een aantal advices waarvan we de meest voorkomende nu kort zullen overlopen. Het *before* advice zal code uitvoeren juist voor het joinpoint bereikt is, terwijl het *after* advice dit zal doen wanneer het joinpoint is uitgevoerd. Hiernaast hebben we ook nog het *after throwing* en *after returning* advice, die zullen worden uitgevoerd wanneer tijdens het uitvoeren van een joinpoint er een exceptie wordt geworpen of na de uitvoe-

ring er een resultaat wordt teruggegeven. Een ander veel gebruikt advice dat AspectJ voorziet is het *around* advice, dat een stuk gedrag rond het joinpoint uitvoert. In dit gedrag kunnen we aangeven wanneer het joinpoint mag worden uitgevoerd door gebruik te maken van *proceed*. Het is niet altijd zeker dat *proceed* wel wordt uitgevoerd in het advice, bijvoorbeeld wanneer eerst wordt gecontroleerd of iemand wel toegang mag krijgen tot dat joinpoint.

In een aspectgeörienteerde workflowtaal, zoals AO4BPEL, is er slechts één soort advice beschikbaar, zoals beschreven in [Brichau and Haupt, 2005]. Hier wordt er een BPEL activiteit toegevoegd aan de joinpoint die geselecteerd is door een pointcut. AO4BPEL voorziet, net zoals AspectJ, de mogelijkheid tot *collectie* en *reflectie* van de context. Zo is het mogelijk dat het advice informatie verkrijgt over het joinpoint, zoals bijvoorbeeld zijn activiteit type of naam.

De body van het advice bepaalt wat er juist wordt uitgevoerd, terwijl het type van het advice bepaalt wanneer dit advice specifiek moet worden uitgevoerd ten opzichte van het joinpoint. De vorige besproken talen zullen maar één body van advice voorzien, namelijk het uitvoeren van extra code of een activiteit. Wij zullen, naast een body die enkel code uitvoert, nog twee andere verschillende body's van advices voorzien. De eerste body van advice voert, vergelijkbaar met de vorige talen, enkel een anonieme functie[1] uit. Deze anonieme functie bevat code die acties gaat uitvoeren in AmbientTalk of NOW. Aangezien AmbientTalk volledig geïmplementeerd is in Java, kunnen we ook gebruik maken van de symbiose tussen Java en AmbientTalk [Van Cutsem et al., 2009] en is het mogelijk om Java code uit te voeren in deze anonieme functie. Een voorbeeld van wat een anonieme functie kan uitvoeren is het starten van de uitvoering van een workflow. Een tweede body van advice dat we nodig hebben is voor het toevoegen van een synchronisatiepunt. Wanneer we in een omgeving komen waar verschillende workflows samenwerken is het mogelijk dat een workflow pas kan verdergaan wanneer een andere workflow een bepaald punt heeft bereikt. Daarom kunnen we een synchronisatiepunt toevoegen dat wacht tot de andere workflow dat bepaald punt bereikt heeft. Een derde body van advice is verantwoordelijk voor het wijzigen van andere workflows. Dit geeft ons de mogelijkheid om bepaalde patronen van workflows te wijzigen. Een voorbeeld hiervan is het toevoegen van een extra branch in een parallelle split patroon van een andere workflow. Deze body verschilt van de eerste in het feit dat andere workflows wijzigen in NOW terwijl ze worden uitgevoerd niet mogelijk is.

We overlopen nu de verschillende advice types en bij welke body deze kunnen gebruikt worden. Eerst zullen we de verschillende types van advices bespreken die bepalen wanneer de body mag worden uitgevoerd ten opzichte van het joinpoint. We voorzien twee algemeen gekende advice types die al eerder besproken zijn: *before* en *after*. Het *before* advice type voert de body uit vlak voor het joinpoint bereikt wordt, terwijl het *after* advice type de body zal uitvoeren wanneer het joinpoint is uitgevoerd.

Wanneer de body een block code is, wordt deze uitgevoerd juist voor of na het joinpoint. In listing 3.2 zien we hiervan een voorbeeld. Op lijn 1 maken we een nieuw aspect aan. Hierna

[1] gekend als lambda's

Figuur 3.4: Before advice.

Figuur 3.5: After advice.

voegen we het pointcut toe aan het aspect, namelijk een `PointcutMethod`. Het pointcut zal dus voldaan zijn wanneer "`InvestigateCause`" wordt uitgevoerd. Op lijn 3 bepalen we het type van het advice en geven we de body mee als argument. Aangezien we een block code willen uitvoeren, zal de body een anonieme functie zijn die wordt meegegeven als argument. Deze anonieme functie zal één parameter bevatten, namelijk `env`, die de data bevat van de workflow waar het pointcut voldaan is, en eventueel kan worden toegevoegd aan de eigen omgeving.

```
1 def aspect := Aspect();
2 aspect.setPointcut(PointcutMethod("InvestigateCause"));
3 aspect.setAdviceBefore({|env| ...});
```

Listing 3.2: Voorbeeld before advice code uitvoeren.

Deze twee advice types kunnen ook worden gebruikt bij de body die verantwoordelijk is voor het toevoegen van een synchronisatiepunt. Wanneer we naar een bestaande oplossing kijken, zoals [Heinlein, 2002], merken we op dat hier een extra interactiemanager wordt toegevoegd. Hieraan moet dan toestemming gevraagd worden om een activiteit van de workflow uit te voeren. Wanneer dit nog niet toegestaan is, bijvoorbeeld omdat deze activiteit het resultaat van een andere activiteit nodig heeft waarvan de uitvoering nog steeds bezig is, wordt er geen toestemming gekregen van de interactiemanager om de uitvoering van de activiteit te starten. Het nadeel hieraan is dat er een centrale interactiemanager moet zijn, en aangezien wij verschillende workflows hebben die elk op een ander apparaat worden gedefinieerd en uitgevoerd, is het onmogelijk om zo een manager op één enkele infrastructuur te definiëren. Daarom willen we een synchronisatiepunt kunnen toevoegen door een advice, waarbij we specifiek bepalen aan de hand van een pointcut wanneer de synchronisatie geslaagd is.

De specifieke advice types voor het toevoegen van het synchronisatiepunt zijn veel beperkter dan deze voor code uit te voeren. We moeten specifiek bepalen waar we het synchronisatiepunt willen toevoegen. Om te bepalen waar het synchronisatiepunt moet worden toegevoegd, maken

advice type	code	synchronisatie	wijzigen
before	x	x	
after	x	x	

Tabel 3.3: Toepasbaarheid body met before en after advice type.

we enkel gebruik van het before en after advice type. Het pointcut van het aspect bepaalt waar het synchronisatiepunt wordt toegevoegd, en het advice type bepaalt of dit voor of na dit joinpoint moet gebeuren. Hiernaast moeten we ook nog bepalen wanneer het synchronisatiepunt mag worden opgeheven. Dit wordt als een extra parameter meegegeven aan het advice type. Deze parameter kan alle mogelijke pointcuts zijn. Het synchronisatiepunt kan worden opgeheven wanneer een activiteit bereikt is, of bepaalde data is toegevoegd aan de omgeving. We kunnen ook de timeout pointcut gebruiken, zodat er na een bepaalde tijd kan worden verdergegaan. In het voorbeeld van de brandweer wordt er een synchronisatiepunt toegevoegd wanneer er een onderzoek wordt geopend.

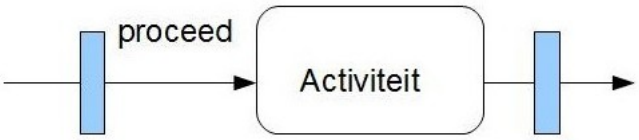
```
1 def aspect := Aspect();
2 aspect.setPointcut(PointcutMethod("CleanUp"));
3 aspect.setAdviceBefore(PointcutMethod("FinishInvestigation"));
```

Listing 3.3: Voorbeeld before advice synchronisatiepunt toevoegen.

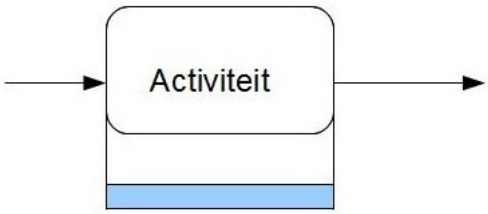
We definiëren eerst een nieuw aspect, waarna we het pointcut toevoegen dat beschrijft waar het synchronisatiepunt moet worden toegevoegd. Dit wordt geïmplementeerd op lijn 1 en 2 in listing 3.3. Hierna bepalen we op lijn 3 het type van advice. We kiezen hier voor het before advice, dus het synchronisatiepunt zal worden toegevoegd juist voor het joinpoint bereikt is. De parameter van ons advice type op lijn 3, `PointcutMethod("FinishInvestigation")`, geeft aan wanneer het synchronisatiepunt mag worden opgeheven. Dus zodra de politie het onderzoek ter plaatse heeft afgesloten en er zeker van is dat er geen extra bewijsmateriaal meer gezocht moet worden, mag het synchronisatiepunt worden opgeheven en kan de uitvoering van de workflow van de brandweer verdergaan. Met het voorgestelde advice body wordt vereiste 6 vervuld.

Naast de vorige twee advice types, before en after, voorzien we ook het around advice type. Dit advice zal juist voor het joinpoint bereikt wordt een taak uitvoeren. Wanneer in deze taak *proceed* wordt opgeroepen wordt het joinpoint uitgevoerd. Na de uitvoering hiervan wordt nog een taak uitgevoerd. Met dit advice body vervullen we vereiste 4.

Het vierde advice type dat we voorzien, is het *during* advice type. Dit advice type zal een taak uitvoeren zolang aan het joinpoint voldaan is. Het advice zal worden uitgevoerd van zodra het joinpoint bereikt is, en dit zal beëindigd worden wanneer de pointcut volledig is afgehandeld. Een voorbeeld hiervan is dat terwijl de brandweer aan het blussen is, zich een collega beschikbaar houdt zodat deze onmiddellijk iemand zijn taak kan overnemen wanneer deze een probleem heeft.

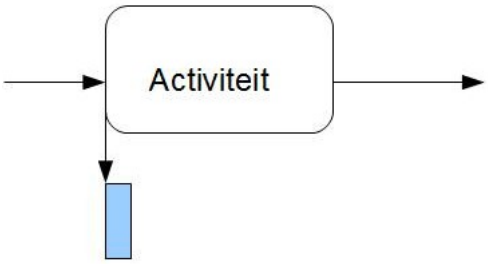


Figuur 3.6: Around advice.



Figuur 3.7: During advice.

Het vijfde advice type dat we voorzien is het *action* advice. Dit advice zal één keer worden uitgevoerd wanneer er een joinpoint bereikt is. Dit zal het geval zijn wanneer er een activiteit gestart is. Het verschil met het before advice is dat de activiteit reeds gestart moet zijn, terwijl het before advice wordt uitgevoerd voor de activiteit. Dit action advice kan dus alleen maar gebruikt worden bij joinpoints op activiteiten.



Figuur 3.8: Action advice.

Vervolgens hebben we nog enkele advice types die specifiek bij de body horen die verantwoordelijk is voor het wijzigen van bepaalde patronen in workflows. Een pointcut bepaalt welk patroon we willen wijzigen. NOW bevat verschillende patronen zoals ondermeer een parallelle split, een sequence, een exclusive choice en een multichoice. Het advice zal een onderdeel toevoegen, verwijderen of vervangen aan deze patronen. Zo is het bijvoorbeeld mogelijk om een extra branch toe te voegen/verwijderen in een parallelle split, net als een keuze te vervangen of toe te voegen in een exclusive/multi choice.

Om deze wijzigingen mogelijk te maken voorzien we verschillende advice types. Om de exclusive choice en multi choice te wijzigen worden er drie advice types voorzien. Het *add* advice

advice type	code	synchronisatie	wijzigen
around	x		
action	x		
during	x		

Tabel 3.4: Toepasbaarheid body met around, action en during advice type.

advice type	code	synchronisatie	wijzigen
add			x
remove			x
replace			x

Tabel 3.5: Toepasbaarheid body met add, remove en replace advice type.

type voegt een extra keuze toe, terwijl *remove* een keuze verwijdert. Het *replace* advice vervangt een keuze. We kunnen hier zowel de voorwaarde vervangen waaraan moet voldaan zijn, alsook wat moet worden uitgevoerd wanneer aan de voorwaarde voldaan is. Om een parallelle split te wijzigen voegt het add advice een extra branch toe aan de parallelle split, terwijl het remove advice een branch verwijdert. Het replace advice zal hier een bepaalde branch vervangen door een andere branch.

In listing 3.4 hebben we een aspect die als pointcut een patroon op een parallelle split neemt. Het add advice voegt aan deze parallelle split een extra branch toe. Op lijn 1 definiëren we het aspect, waarna we op lijn 2 het pointcut gaan definiëren. Ditmaal is het pointcut pas voldaan wanneer er een parallelle split bereikt is. Wanneer we enkel het type van het patroon meegeven, zoals hier het geval is, selecteert het pointcut het patroon dat de eerste keer wordt uitgevoerd. In plaats van het type kunnen we ook een referentie naar een specifieke parallelle split meegeven. Tenslotte bepalen we op lijn 3 het advice type, wat in het voorbeeld iets zal toevoegen. Aangezien het pointcut werkt op een parallelle split wordt dus een sequence toegevoegd als een extra branch. Met dit voorgestelde advice body vervullen we vereiste 5.

```
1 def aspect := Aspect();
2 aspect.setPointcut(PointcutPattern(ParallelSplit));
3 aspect.setAdviceAdd(`Sequence(ExtinguishService.turnOffEverything()));
```

Listing 3.4: Voorbeeld wijzigen workflow.

In volgende tabel hebben we een overzicht van alle advice types die er zijn per advice body. We merken op dat het before en after advice type zowel bij het code als synchronisatie advice voorkomt.

	code	synchronisatie	wijzigen
before	X	X	
after	X	X	
during	X		
action	X		
around	X		
add			X
remove			X
replace			X

Tabel 3.6: Advice types per body.

3.5 Aspect module model

Wanneer we de pointcuts en advices geïdentificeerd hebben gaan we deze combineren in een aspect. Wanneer aan de pointcut voldaan is worden de advices in het aspect uitgevoerd. De volgende listing is een voorbeeld van een aspect gedefinieerd in NOW. We moeten eerst een nieuw aspect object aanmaken, en daarna kunnen we de pointcut en advices initialiseren.

```
1 def aspect := Aspect();
2 aspect.setPointcut(PointcutMethod("InvestigateCause"));
3 aspect.setAdviceBefore({|env| addToEnv(env, WFenv);
4                               policeworkflow.start(WFenv);});
```

Listing 3.5: Aspect voorbeeld.

We maken eerst op lijn 1 een nieuw aspect aan, en voegen dan het pointcut toe aan dat aspect op lijn 2. Vervolgens voegen we op lijn 3 een anonieme functie toe als het before advice. Deze anonieme functie voegt eerst de data toe aan de omgeving waar het pointcut bereikt is, en start dan de workflow van de politie (lijn 4). Dit advice wordt uitgevoerd juist voordat de activiteit *"InvestigateCause"* wordt gestart. Wanneer we het aspect gedefinieerd hebben, wordt dit toegepast op alle workflows die binnen bereik zijn. In sommige gevallen is het mogelijk dat het aspect maar voor een bepaalde periode beschikbaar moet zijn. Wanneer we een workflow hebben die dagelijks herhaald wordt, is het mogelijk dat het aspect alleen maar op bepaalde dagen moet worden toegepast. Een andere mogelijkheid is dat we zeggen dat het aspect maar actief wordt nadat een bepaald punt bereikt is, of dat het niet meer mag worden toegepast na een bepaalde activiteit. Een voorbeeld is dat we een aspect definiëren dat actief is tijdens een brand, maar wanneer de brand geblust is moet dit aspect niet meer worden toegepast. Standaard is een aspect actief vanaf het ogenblik dat het aspect wordt aangemaakt. Als we de start- en eindfase willen wijzigen, kunnen we dit doen door de functie `setStart` en `setEnd` te gebruiken. Hier geven we als parameter een pointcut mee die bepaalt wanneer het aspect actief wordt en zijn advice uitvoert wanneer de pointcut bereikt wordt.

Een andere eigenschap van het aspect is dat we bepalen hoeveel keer het mag worden uitgevoerd. Standaard wordt het advice van een aspect altijd uitgevoerd wanneer aan het pointcut voldaan is. We kunnen ook zeggen dat ons advice maar een aantal keer mag worden uitgevoerd. Wanneer we bijvoorbeeld een branch willen toevoegen aan een parallelle split, is het mogelijk dat dit maar één keer mag worden toegevoegd. Daarom kunnen we zeggen dat een advice maar *n-keer* kan worden uitgevoerd. Wanneer dit aantal bereikt is, wordt het aspect *passief* en niet meer toegepast op de workflows.

3.6 Aspect compositie

Wanneer we verschillende aspecten definiëren, is het mogelijk dat deze op dezelfde pointcuts worden toegepast waardoor er enkele conflicten met de advices kunnen plaatsvinden [Tian et al., 2009]. Zo is het mogelijk dat twee advices van verschillende aspecten afhankelijk zijn van elkaar en er een advice moet worden uitgevoerd voor een ander. Een voorbeeld hiervan is dat er twee aspecten gedefinieerd zijn met verschillende advices maar elk met hetzelfde pointcut. Eén advice gaat een synchronisatiepunt toevoegen aan het begin van een workflow, terwijl het advice van het andere aspect de workflow waar het synchronisatiepunt moet worden toegevoegd gaat starten. Wanneer eerst het advice wordt uitgevoerd die de workflow gaat starten, is het mogelijk dat de workflow waar het synchronisatiepunt moet worden toegevoegd reeds voltooid is. Een gevolg hiervan kan zijn dat deze workflow niet correct gesynchroniseerd is met een andere workflow en niet de nodige data bevat om verder te gaan.

Deze conflicten kunnen we vermijden door aan te geven in welke specifieke volgorde deze moeten worden uitgevoerd. Om dit aan te geven moeten we rekening houden met twee situaties. Indien alle aspecten gekend zijn, kunnen we deze eenvoudig rangschikken volgens hun prioriteit. Wanneer aspect1 het synchronisatiepunt toevoegt, en aspect2 de workflow start, zullen we de volgorde op volgende manier bepalen:

```
1         orderExecutionAspects(aspect1, aspect2);
```

Listing 3.6: Rangschikken uitvoeren van aspecten.

Dit zal ervoor zorgen dat aspect1 wordt uitgevoerd voor aspect2 indien hun pointcuts op hetzelfde moment voldaan zijn. De tweede situatie is dat we alle aspecten niet op voorhand kennen, doordat deze eventueel zijn gedefinieerd op een andere vaste infrastructuur. Dan hebben we geen weet van deze aspecten en kunnen we deze niet rangschikken volgens belangrijkheid. We kunnen aan het aspect wel een prioriteit geven die bepaalt wanneer en hoe belangrijk dit aspect is. Wanneer de prioriteit negatief is geeft dit aan hoe belangrijk het is dat het advice van het aspect als eerste wordt uitgevoerd, terwijl een positieve prioriteit aangeeft hoe belangrijk het is dat het advice als laatste wordt uitgevoerd. De prioriteit zal dus liggen in een interval van -100 tot 100. Wanneer we het aspect nemen die een synchronisatiepunt moet toevoegen, heeft dit aspect dus een negatieve prioriteit zodat het als eerste wordt uitgevoerd.

```
1        aspect1.setPriority(-100);
```

Listing 3.7: Aspect prioriteit geven.

3.7 Samenvatting

In dit hoofdstuk hebben we besproken hoe we ons probleem hebben aangepakt. We hebben gekozen voor een aspectgeoriënteerde oplossing waarbij we in een aspect definiëren waar en wanneer er iets moet worden uitgevoerd. In sectie 3.1 bespreken we de motivatie en vereisten waaraan onze oplossing moet voldoen. Vervolgens hebben we in sectie 3.2 een overzicht gegeven van hoe we het probleem aanpakken. In de daaropvolgende secties hebben we de effectieve aanpak besproken.

In sectie 3.3 hebben we het joinpoint model en de bijhorende pointcuts besproken. We identificeren verschillende joinpoints op activiteiten en data van workflows, alsook workflow patronen en disconnecties van services. Deze joinpoints kunnen we identificeren aan de hand van pointcuts. Zo zijn er pointcuts die activiteiten identificeren, zoals `PointcutMethod`, en vereiste 1 vervullen. Andere pointcuts, zoals `PointcutDataAdded`, zijn voldaan wanneer bepaalde data wordt toegevoegd aan de workflow en vervullen vereiste 2. Tenslotte hebben we nog pointcuts die een disconnectie aanduiden, `PointcutDisconnected` en een pointcut die voldaan is wanneer er een timeout is verstreken, `PointcutTimeout`. Hierdoor is vereiste 7 voldaan. De pointcut predikaten `intersection` en `union` laten ons toe om combinaties van verschillende activiteiten aan te duiden, waardoor aan vereiste 3 is voldaan.

In sectie 3.4 bespreken we het advice model. We identificeren drie verschillende body's van advices. Er is een body die enkel een anonieme functie uitvoert waardoor vereiste 4 voldaan is. De tweede advice body voegt een synchronisatiepunt toe. Hierdoor hebben we de mogelijkheid om verschillende workflows te synchroniseren, waardoor vereiste 6 voldaan is. De derde advice body laat ons toe om andere workflows te wijzigen. Zo kunnen we aan bepaalde workflow patronen iets gaan toevoegen, verwijderen of veranderen, zoals bijvoorbeeld een extra branch in een parallelle split. Hierdoor is er aan vereiste 5 voldaan.

Vervolgens hebben we in sectie 3.5 besproken hoe een aspect wordt gedefinieerd en wanneer dit actief zal worden. Hierna hebben we nog sectie 3.6 waarin we de mogelijke conflicten bespreken tussen verschillende aspecten en hoe we dit eventueel kunnen oplossen door de aspecten een volgorde of prioriteit te geven.

	oplossing	sectie
Vereiste 1	PointcutMethod	3.3
Vereiste 2	PointcutData, PointcutDataAdded, PointcutDataChanged	3.3
Vereiste 3	intersection, union	3.3
Vereiste 4	advice types: before, after, during, action, around	3.4
Vereiste 5	advice types: add, remove, replace	3.4
Vereiste 6	advice types: before, after	3.4
Vereiste 7	PointcutDisconnected, PointcutTimeOut	3.3

Tabel 3.7: Overzicht vereisten en hun oplossing.

Hoofdstuk 4

Implementatie

In dit hoofdstuk overlopen we de implementatie van de aspectgeoriënteerde oplossing. Hierbij overlopen we eerst AmbientTalk en bespreken we de belangrijkste technieken die we hiervan gebruiken. Vervolgens leggen we het design uit van de aspectgeoriënteerde oplossing, waarna we de implementatie hiervan overlopen, gevolgd door de implementatie van de verschillende pointcuts en de verschillende advice types. Hierna identificeren we in de implementatie van NOW de verschillende joinpoints. Tenslotte bespreken we hoe aspecten worden ontdekt door andere vaste infrastructuren.

4.1 AmbientTalk

Momenteel beschikt bijna iedereen wel over een mobiel toestel dat gebruikt wordt om informatie op te zoeken, applicaties te gebruiken en om met elkaar te communiceren. Hierdoor is er dan ook een steeds grotere vraag naar mobiele applicaties die probleemloos kunnen worden gebruikt en die communiceren met andere toestellen. Een eigenschap van mobiele ad-hoc netwerken is dat een gebruiker pas verbinding kan maken met een andere gebruiker wanneer deze in elkaars buurt komen. Gebruikers worden dus dynamisch ontdekt waarna deze met elkaar verbonden worden door een peer-to-peer verbinding. Deze verbinding tussen de verschillende toestellen is zeer volatiel. Wanneer toestellen te ver uit elkaar gaan, kan de verbinding verbroken worden. Hierbij weet men niet of de oorzaak hiervan voor altijd is, of de verbinding zo dadelijk opnieuw hersteld gaat worden. In een mobiel ad-hoc netwerk is de kans op disconnecties dus zeer groot, en we gaan deze beschouwen alsof het normaal is dat dit voorkomt in plaats van dat het een uitzondering is.

AmbientTalk [Van Cutsem et al., 2007] is een programmeertaal specifiek ontwikkeld voor de ontwikkeling van programma's in mobiele ad-hoc netwerken. Om aan vorige vereisten te voldoen, voldoet AmbientTalk aan een aantal eigenschappen [Eugster et al., 2003]. Eerst en vooral is er een *ontkoppeling in tijd* nodig. Verschillende gebruikers moeten niet op hetzelfde moment online zijn. Een gebruiker moet de mogelijkheid hebben om een bericht te sturen naar iemand die offline is, alsook een gebruiker een bericht moet kunnen ontvangen van een zender die op

dat ogenblik offline is. Ten tweede heeft AmbientTalk ook een *ontkoppeling in synchronisatie* nodig. De control flow van gebruikers mag niet geblokkeerd worden tijdens het verzenden en ontvangen van berichten. Wanneer iemand een bericht verzonden heeft, moet deze kunnen verder werken in plaats van te blijven wachten tot het bericht is aangekomen. Tenslotte moet er ook een *ontkoppeling in ruimte* zijn. Verschillende gebruikers moeten elkaar niet op voorhand kennen. Gebruikers die bij elkaar in de buurt komen, moeten de mogelijkheid hebben om verbinding te maken met de aanwezige gebruikers. In de volgende secties bespreken we hoe AmbientTalk deze eigenschappen vervult.

4.1.1 Objectgeoriënteerd programmeren in AmbientTalk

AmbientTalk is een dynamisch getypeerde, objectgeoriënteerde programmeertaal. In tegenstelling tot de meeste andere objectgeoriënteerde talen zullen objecten niet geïnstantieerd worden uit een klasse, maar eerder uit het niets, of gekloond/gewijzigd uit een ander object.

```
1  def Item := object: {
2      def category; // a type classifying the item
3      def description; // a string describing the item
4      def init(cat, desc) {
5          category := cat;
6          description := desc;
7      };
8      def getContactInfo() {
9          [category, description]; // return the contact details
10     };
11 };
```

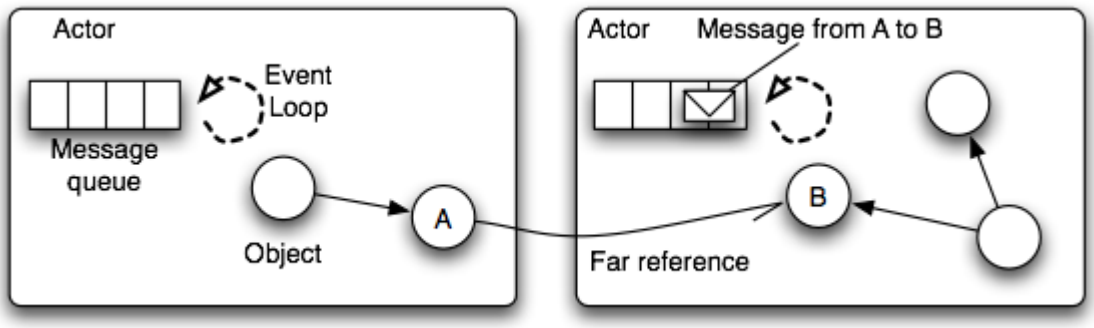
Listing 4.1: AmbientTalk object Item.

In listing 4.1 hebben we een object, `Item`, die twee variabelen bevat, namelijk een `category` en een `description` (lijn 2 en 3). De `init` procedure wordt gebruikt om een nieuw object te initialiseren (lijn 4). Zo kloont `Item.new(Phone, "this is a new phone")` het Item object, en hierdoor wordt de `init` functie opgeroepen als constructor. Zo wordt het nieuw object geïnitialiseerd met twee argumenten. Het eerste argument hiervan, `Phone`, is een tag die we kunnen definiëren door `deftype Phone`. Wanneer dit gebeurd is, kunnen er functies worden opgeroepen, zoals `getContactInfo()`, die een tabel[1], met de twee variabelen die het object bevat, teruggeeft.

4.1.2 AmbientTalk actors

AmbientTalk's concurrency model is actor gebaseerd. Een AmbientTalk virtuele machine heeft de mogelijkheid om verschillende actoren parallel te laten uitvoeren. Deze actoren worden gerepresenteerd als een event loop. Hierbij worden alle events opgevangen binnen de berichtenqueue van de actor, waarbij ook events gerepresenteerd worden door berichten, event notificaties door

[1] AT heeft tables als containers.



Figuur 4.1: Event loop model AmbientTalk.

asynchrone berichten en event handlers door reguliere objecten. Wanneer een bericht naar een actor wordt verstuurd, wordt dit toegevoegd aan zijn queue, en de actor zal deze berichten één voor één verwerken.

Objecten binnen een actor kunnen sequentieel berichten versturen naar elkaar door `O.message()`. Het is ook mogelijk om vanuit een actor asynchrone berichten naar een object in een andere actor te sturen. Dit object in de andere actor wordt dan een *far reference* genoemd. Asynchrone berichten zullen gestuurd worden door `o<-message()`, en deze komen dan in de queue van de actor terecht die dit bericht zelf gaat verwerken. Wanneer zo een asynchroon bericht wordt verzonden, wordt er een *future* teruggeven, welke een plaatsvervanger is voor de echte return waarde. Wanneer de return waarde effectief berekend is, wordt de future hierdoor vervangen. Wanneer er asynchrone berichten worden verstuurd naar het future object, worden deze eerst opgeslagen tot de future opgelost is. Wanneer dit gebeurd is, worden alle opgeslagen berichten in de juiste volgorde uitgevoerd. Hiernaast voorziet AmbientTalk ook een constructie die de mogelijkheid geeft om iets uit te voeren van zodra de future opgelost is.

```
1 when: item<-getContactInfo() becomes: { |contactInfo|
2     // execution is postponed until future is resolved
3     system.println("Found item, contact: " + contactInfo[2]);
4 } catch: { |exception| ... };
5 // code following when: is processed immediately
```

Listing 4.2: Voorbeeld future in AmbientTalk.

Wanneer we `when: future becomes: block` gebruiken zal deze een block code uitvoeren wanneer de future is opgelost. Dit block code is een anonieme functie die één argument bevat, namelijk het resultaat van de future. Aangezien deze future ook verschillende falingen kan teruggeven, kunnen deze worden opgevangen door `catch`, die een block code bevat die als argument de exceptie binnenkrijgt.

4.1.3 Objecten exporteren en afhandelen van falingen

Aangezien AmbientTalk voornamelijk gebruikt wordt in mobiele ad-hoc netwerken, moet het mogelijk zijn om op een eenvoudige manier verbinding te maken met nieuwe gebruikers. Dit wordt mogelijk gemaakt door het exporteren van objecten.

```
1 deftype Phone;
2 def phone := object: {
3     def number := "911";
4     def owner := "Jeroen";
5
6     def getNumber(){
7         number
8     };
9     def getOwner(){
10        owner
11    };
12    def updateNumber(nw){
13        number = nw;
14    };
15
16 };
17 export: phone as: Phone;
```

Listing 4.3: Voorbeeld exporteren van object.

In listing 4.3 definiëren we eerst een `phone` object, waarna we dit exporteren (lijn 17). Dit object kan dan door andere actoren ontdekt worden. Wanneer dit gebeurt ontvangen we een far reference naar dit `phone` object. Hiernaar worden asynchrone berichten gestuurd. In het volgende voorbeeld ontdekken we een `phone` object en vragen wie de eigenaar is.

```
1 when: Phone discovered: {|phone|
2     when: phone<-getOwner() becomes: {|name|
3             system.println("The owner of the phone is: " + name);
4     };
5     whenever: phone disconnected: {...};
6     whenever: phone reconnected: {...};
7 }
```

Listing 4.4: Voorbeeld ontdekken van object en versturen van bericht.

Op lijn 1 zien we de ontdekking van één `phone` object door `when: discovered:`, waarna we op lijn 2 een asynchroon bericht naar dit object sturen. Asynchrone berichten worden gestuurd door `<-`. Naast de ondekking van objecten moeten we ook rekening houden met de disconnectie en reconnectie wanneer er een faling voorkomt. Hiervoor kunnen we `whenever: disconnected:` gebruiken die een block uitvoert elke keer als er een disconnectie is met een *far reference* (lijn 5), en `whenever: reconnected` (lijn 6), die een block uitvoert elke keer de verbinding met een *far reference* opnieuw hersteld is.

Wanneer we in AmbientTalk een *far reference* hebben, blijft deze bestand tegen netwerk falingen. Wanneer er een disconnectie is met het object dat de far reference vertegenwoordigt, zal deze far reference blijven bestaan. De berichten die naar dit remote object ondertussen worden verzonden, worden bijgehouden tot de verbinding opnieuw hersteld is. Wanneer dit het geval is, worden alle opgeslagen berichten één voor één uitgevoerd. Hierdoor hebben tijdelijke netwerk disconnecties geen effect op de control flow van een applicatie.

4.1.4 Symbiose met Java

Aangezien AmbientTalk volledig is geïmplementeerd in Java en wordt uitgevoerd door de Java Virtual Machine, kunnen we in AmbientTalk gebruik maken van de verschillende *libraries* die voorzien zijn door Java. Zo is het mogelijk om Java klassen te instantiëren vanuit AmbientTalk en berichten te sturen naar deze Java objecten. In de omgekeerde richting werken is ook mogelijk. Zo kunnen AmbientTalk objecten gebruikt worden in een Java object wanneer deze geïmplementeerd is door een Java interface. Wanneer we kijken naar de variabelen in Java en AmbientTalk, merken we meteen op dat Java een statisch getypeerde taal is, terwijl AmbientTalk dynamisch getypeerd is. AmbientTalk voorziet echter een ingebouwde conversie van data tussen de twee talen. Een getal in AmbientTalk wordt automatisch geconverteerd naar een `int` in Java, en omgekeerd. Hetzelfde geldt voor een string in Java, die wordt geconverteerd naar een tekst in AmbientTalk.

Wanneer we in AmbientTalk een Java klasse nodig hebben, maken we gebruik van het keywoord `jlobby`. We kunnen nu een Java klasse selecteren door het juiste pad op te geven waarbij `jlobby` de topklasse is. Zo zullen we via `def Vector := jlobby.java.util.Vector;` de functies van de Vector klasse kunnen raadplegen. Nu kunnen we een nieuwe vector instantiëren door `aVector := Vector.new()` en hierop functies oproepen zoals bij een Java object. Zo zal `aVector.add(1)` het cijfer 1 toevoegen aan de vector. Wanneer we een functie oproepen op een Java object vanuit AmbientTalk, moeten we er steeds rekening mee houden dat er functie overlading kan plaatsvinden. Wanneer het aantal argumenten verschillen, is er geen probleem en wordt de functie met het juiste aantal argumenten uitgevoerd. Wanneer het type argument verschilt, hebben we een probleem omdat er misschien meerdere mogelijkheden zijn. Zo komt een nummer in AmbientTalk overeen met zowel een `int` als `Object`. Daarom moeten we in AmbientTalk specifiek aangeven naar welk type het argument moet gecast worden. Zo zien we in volgend voorbeeld dat we de `remove` functie van een vector object specifiek casten naar de functie met een integer als argument.

```
1 def remove := aVector.&remove;
2 remove.cast(jlobby.java.lang.Integer.TYPE)(0);
```

Listing 4.5: Voorbeeld functie overlading.

Wanneer we vanuit Java een AmbientTalk object willen gebruiken, moeten we een interface definiëren. Deze moet de functies bevatten die kunnen worden opgeroepen op het AmbientTalk object. In het volgende voorbeeld zien we de definitie van een AmbientTalk object die twee

functies bevat. Om deze functies te kunnen oproepen vanuit een Java object moeten we deze declareren in een interface. Het AmbientTalk object wordt dan als parameter meegegeven (listing 4.7 lijn 8) wanneer een functie van het Java object wordt opgeroepen, waarna op dit AmbientTalk object zijn functies kunnen worden opgeroepen vanuit een Java klasse (listing 4.6 lijn 7 en 10).

```
1  public class SymbiosisDemo {
2          public interface PingPong {
3                  public int ping();
4                  public int pong();
5          }
6          public int run(PingPong pp) {
7                  return pp.ping();
8          }
9          public int run2(PingPong pp) {
10                 return pp.pong();
11         }
12 }
```

Listing 4.6: Definitie klasse en interface in Java.

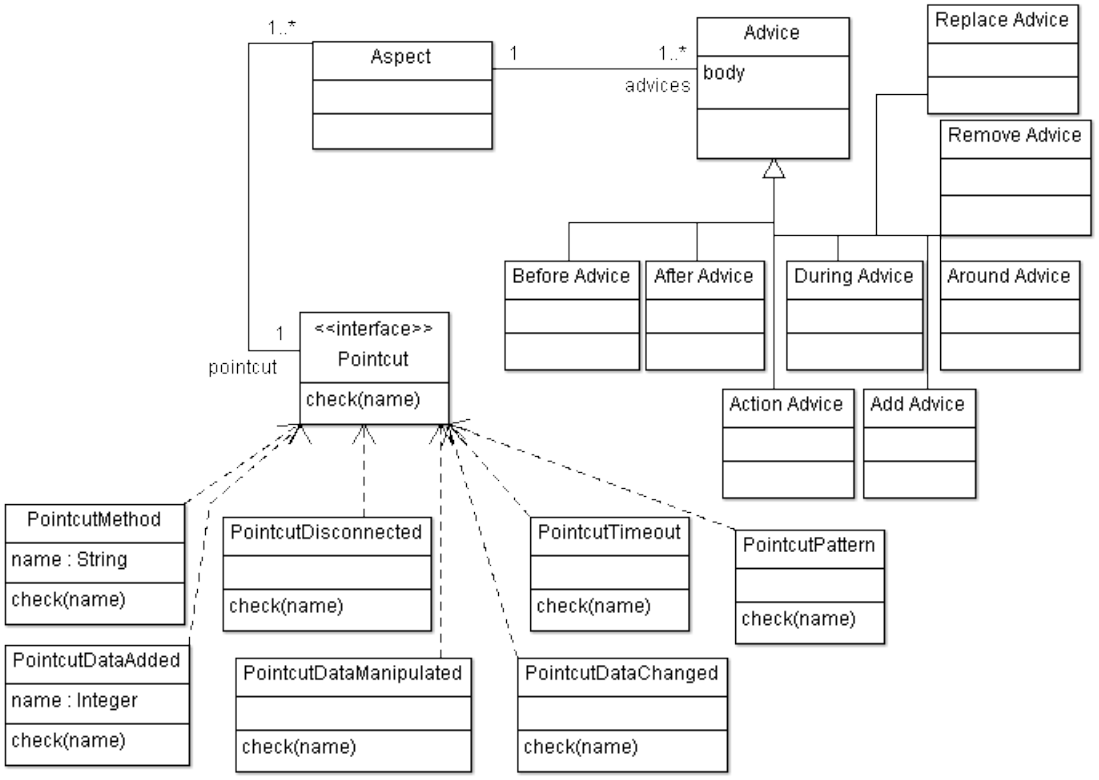
```
1 def SymbiosisDemo := jlobby.at.tutorial.SymbiosisDemo;
2 def showSymbiosis() {
3   def javaDemo := SymbiosisDemo.new();
4   def atObject := object: {
5     def ping() { ... };
6     def pong() { ... };
7   };
8   javaDemo.run(atObject);
9 };
```

Listing 4.7: Voorbeeld symbiose met Java.

4.2 Aspecten in NOW

Een aspect wordt, net zoals een pointcut, voorgesteld door een object in AmbientTalk. In figuur 4.2 hebben we een UML diagram die de structuur weergeeft van het aspect, pointcut en advice. Wanneer we kijken naar de Pointcut interface, merken we op dat deze een abstracte functie `check` heeft. Elke pointcut heeft dus een specifieke implementatie voor deze functie die kijkt of aan het pointcut voldaan is. Verder zien we dat sommige pointcuts een attribuut `name` bevatten. Deze wordt gebruikt in de `check` functie om te kijken welk specifiek joinpoint we met dit pointcut willen selecteren. De pointcuts `union` en `intersection` bevatten meerdere pointcuts waarvan respectievelijk één of meerdere moeten voldaan zijn voor dit pointcut bereikt is.

De aspect klasse bevat als attributen ondermeer een pointcut, een tabel met advices, en een tabel met alle referenties van andere vaste infrastructuren waarop het aspect wordt toegepast. Deze worden dan gebruikt door het pointcut `disconnected` om te bepalen wanneer er een disconnectie is. Hiernaast bevat het aspect ook een collectie van advices, gerepresenteerd door de



Figuur 4.2: Design aspecten, pointcuts en advices.

Advice klasse. Deze heeft één attribuut, namelijk de body van het advice die moet worden uitgevoerd. Er zijn verschillende types van advices, zoals het *Before Advice*, die een subklasse is van de *Advice* klasse.

Om het aspect toe te passen op verschillende vaste infrastructuren, maken we gebruik van het exporteren van objecten. Doordat we het aspect, gerepresenteerd door een object, exporteren kan dit worden ontdekt door andere vaste infrastructuren. Doorheen de uitvoering van een workflow in NOW wordt dan steeds gekeken wanneer er een joinpoint wordt uitgevoerd of dit joinpoint geselecteerd is door het pointcut van een aspect. Wanneer dit het geval is, zullen de advice types worden uitgevoerd op het juiste moment.

We bespreken nu de implementatie van een aspect in listing 4.8. Om een nieuw aspect te verkrijgen roepen we de functie `Aspect` op, die een nieuw object aanmaakt die het aspect voorstelt. We geven als argument eventueel de workflowomgeving mee waarop dit aspect moet worden toegepast. Wanneer er geen workflowomgeving wordt meegegeven wordt het aspect alleen geëxporteerd naar andere vaste infrastructuren, maar wordt het niet toegepast op de workflows van de vaste infrastructuur waarop het aspect zelf gedefinieerd is. Een aspect wordt dus gerepresenteerd door een object in AmbientTalk. Eerst declareren we enkele variabelen in het object,

zoals de pointcut van het aspect (lijn 3), een tabel die de *far refs* bevat van externe vaste infrastructuren waarop het aspect ook wordt toegepast (lijn 4), en een tabel met de advices die moeten worden uitgevoerd (lijn 5). Hiernaast hebben we ook nog een variabele `syncPoint` (lijn 6) die het pointcut bijhoudt dat bepaalt wanneer een synchronisatiepunt mag worden opgeheven. Wanneer dit het geval is, wordt een future opgelost die wordt bijgehouden in `syncResolver` (lijn 7). De functies betreffende advices worden verder uitgelegd in sectie 4.2.2.

Als er in het aspect reeds een pointcut toegevoegd is moet er gecontroleerd worden of bij de uitvoering van een joinpoint dit geselecteerd is door het pointcut. Dit gebeurt op lijn 50 in de functie `check`. Wanneer er een synchronisatiepunt gedefinieerd is, moet ook gecontroleerd worden of dit mag worden opgeheven. Hiervoor wordt de functie `checkSyncPointReached` opgeroepen op lijn 51. Nu de definitie van het object voltooid is, kan het object geëxporteerd worden, wat gebeurt op lijn 77. Hierdoor kunnen andere vaste infrastructuren het aspect ontdekken waardoor het ook daarop zal worden toegepast. Om het aspect toe te passen op de infrastructuur waarop het gedefinieerd is voegen we het aspect toe aan de workflow omgeving (lijn 79). Tenslotte geven we op lijn 82 nog het object terug die het aspect voorstelt.

```
1  def Aspect(wfenv) {
2      def obj := object: {
3          def pointcut := nil;
4          def farrefs := [];
5          def advices := [];
6          def syncPoint := nil;
7          def syncResolver := nil;
8
9          def addAdvice(type, adv){
10             def obj := object: {
11                 def body := adv;
12             } taggedAs: [type];
13             advices := advices + [obj];
14         };
15
16         def executeAdvice(type, nEnv := nil){
17             def [result, resolver] := makeFuture();
18             def relevantAdvices := advices.filter({ |ele|
19                                                 is: ele taggedAs: type });
20             if: (relevantAdvice != nil) then: {
21                 relevantAdvices.each: {|adv|
22                     if: (is: adv.body taggedAs: Pointcut) then: {
23                         syncPoint := adv.body;
24                         syncResolver := resolver;
25                     } else: {
26                         adv.body(nEnv);
27                         resolver.resolve(true);
28                     };
29                 };
```

```
30              };
31              result
32          };
33
34          def setAdviceBefore(adv){
35              addAdvice(before, adv);
36          };
37
38          ...
39
40
41          def checkSyncPointReached(ele){
42              if: !(syncPoint == nil){
43                  if: syncPoint.check(ele){
44                      syncResolver.resolve(true);
45                  };
46              };
47          };
48
49
50          def check(ele){
51              checkSyncPointReached(ele);
52              if: (pointcut == nil) then: {
53                  false
54              } else: {
55                  pointcut.check(ele)
56              }
57          };
58
59          def disconnection(servicetag){
60              if: check(servicetag){
61                  executeAdvice(after);
62              };
63          };
64
65          def getPatternAdvices(){
66              def addadvices := advices.filter({ |ele|
67                                                  is: ele taggedAs: add });
68              def removeadvices := advices.filter({ |ele|
69                                                  is: ele taggedAs: remove });
70              def replaceadvices := advices.filter({ |ele|
71                                                  is: ele taggedAs: replace });
72              [addadvices, removeadvices, replacedvices]
73
74          };
75      };
76
77      export: obj as: Aspect;
78      if: (wfenv != nil) then: {
79          wfenv.addAspect(obj);
80      };
```

```
81
82      obj
83 };
```

Listing 4.8: Implementatie aspect.

4.2.1 Pointcuts

De verschillende pointcuts definiëren we, net zoals bij het aspect, als een object. Het eerste pointcut in listing 4.9 is een pointcut die een activiteit selecteert. De naam van deze activiteit wordt meegegeven als een argument op lijn 1, en bijgehouden in de variabele `name` op lijn 3. Voorts bevat dit object slechts één functie, namelijk `check`. Deze zal controleren of de naam van het joinpoint, dat als argument wordt meegegeven, overeenkomt met deze van het pointcut, waardoor er steeds een boolean wordt teruggegeven. Als het joinpoint (jp) getagged is als een `dataPointcut`, wordt *false* teruggegeven omdat het joinpoint geen activiteit aanduidt. Indien dit wel het geval is, controleren we op lijn 10 of de activiteit getagged is als een symbool. Wanneer dit zo is nemen we de tekst van dit symbool (lijn 11), zodat we deze kunnen vergelijken met de naam van het pointcut, bijgehouden in `name`. Doordat we gebruik maken van pattern matching door $\sim$= kan de naam van het pointcut ook een reguliere expressie zijn.

```
1  def PointcutMethod(nam){
2   object: {
3      def name := nam;
4
5      def check(jp){
6          if: (is: jp taggedAs: dataPointcut) then: {
7               false
8          } else: {
9              def toCompare := jp;
10             if: (is: jp taggedAs: Symbol) then: {
11                  toCompare := jp.text;
12             };
13             toCompare ~= name;
14         };
15     };
16   };
17 };
```

Listing 4.9: Implementatie PointcutMethod.

De pointcut *PointcutDataAdded* controleert of data die wordt toegevoegd overeenkomt met deze van het pointcut. De functie `check` op lijn 5 controleert opnieuw of dit het geval is. Eerst controleren we of het meegegeven argument wel getagged is als `added` op lijn 6. Indien dit niet het geval is wordt false teruggegeven op lijn 10. Wanneer dit wel het geval is zal `jp` een `isolate` zijn met één variabele, namelijk `name`. Deze naam houden we dan bij in de variabele *value* op lijn 7, waarna we op lijn 8 de string van deze waarde vergelijken met deze van het pointcut.

```
1 def PointcutDataAdded(nam){
2  object: {
3     def name := nam;
4
5     def check(jp){
6             if: (is: jp taggedAs: added) then: {
7             def value := jp.name;
8             value.text == name
9         } else: {
10            false
11        };
12    };
13 };
14 };
```

Listing 4.10: Implementatie PointcutDataAdded.

De pointcut `PointcutIntersection` is pas voldaan als alle pointcuts die er aan worden meegegeven zijn voldaan. Door gebruik te maken van `@arg` op lijn 1 kunnen we een variabel aantal argumenten meegeven die worden bijgehouden. Op lijn 3 houden we al deze pointcuts bij in de variabele `allPointcuts`. De variabele `reached` op lijn 4 is een tabel die bijhoudt welke van de pointcuts reeds voldaan zijn, terwijl de variabele `handled` een boolean is die zegt of alle pointcuts reeds voldaan zijn. Vervolgens hebben we op lijn 7 de functie `check`. Wanneer alle pointcuts reeds voldaan zijn, wordt false teruggegeven. Anders kijken we voor elk pointcut apart of het voldaan is (lijn 9 en 10). Wanneer dit het geval is, voegen we het pointcut toe aan de tabel `reached` (lijn 12) als dit pointcut nog niet eerder bereikt was (lijn 11). Vervolgens kijken we op lijn 16 of alle pointcuts voldaan zijn. Indien dit zo is maken we de variabele `handled` true (lijn 17) en geven we true terug (lijn 18), anders geven we false terug (lijn 20).

```
1 def PointcutIntersection(@arg){
2  object: {
3     def allPointcuts := arg;
4     def reached := [];
5     def handled := false;
6
7     def check(jp){
8         if: (! handled) then: {
9             allPointcuts.each: {|pc|
10                if: (pc.check(jp)) then: {
11                    if: (! reached.contains(pc)) then: {
12                        reached := reached + [pc]
13                    }
14                }
15            };
16            if: (allPointcuts.length == reached.length) then: {
```

```
17                  handled := true;
18                  true
19              } else: {
20                  false
21              }
22          } else: {
23              false
24          };
25      };
26  };
27 };
```

Listing 4.11: Implementatie PointcutIntersection.

Aan de volgende `PointcutUnion` kunnen we, net zoals bij `PointcutIntersection`, meerdere pointcuts als argument meegeven. Al deze pointcuts worden bijgehouden in de variabele `allPointcuts` op lijn 3. De functie `check` op lijn 5 controleert opnieuw of aan dit pointcut voldaan is. Hiervoor moeten we de functie `check` oproepen op alle pointcuts in *allPointcuts* (lijn 7). Wanneer één van deze pointcuts voldaan is (lijn 8), zal het `PointcutUnion` ook voldaan zijn.

```
1 def PointcutUnion(@arg){
2  object: {
3      def allPointcuts := arg;
4
5      def check(jp){
6          def result := false;
7          allPointcuts.each: {|pc|
8              if: (pc.check(jp)) then: {
9                  result := true;
10              }
11          };
12          result
13      };
14  };
15 };
```

Listing 4.12: Implementatie PointcutUnion.

Tenslotte hebben we nog de pointcut `PointcutDisconnected` in listing 4.13. Hierbij controleert de `check` functie of de naam van de service die gedisconnecteerd is overeenkomt met deze van het pointcut (lijn 6).

4.2.2 Advices

Een advice wordt voorgesteld door een object die één variabele bevat, namelijk de `body` van het advice. De functies die advices toevoegen en uitvoeren zijn allen gedefinieerd in het aspect object (listing 4.8). Wanneer een advice wordt toegevoegd aan het aspect, zoals bijvoorbeeld het

```
1 def PointcutDisconnected(servicetag){
2  object: {
3     def name := servicetag;
4
5     def check(jp){
6         jp == name;
7     };
8  };
9 };
```

Listing 4.13: Implementatie PointcutDisconnected.

before advice door de functie `setAdviceBefore` (lijn 34), wordt hierin de functie `addAdvice` opgeroepen. Deze functie (lijn 9) heeft twee argumenten: de body van het advice en een typetag die het advice type representeert. We maken een nieuw advice object aan en taggen dit met de typetag van het advice type (lijn 10-12). Tenslotte voegen we dit advice toe aan de lijst met alle advices (lijn 13). Voor elk advice type is er een typetag gedefinieerd, waarbij elk type een subtype is van een advice.

Het uitvoeren van de body van advices wordt verwezenlijkt door de functie `executeAdvice` (lijn 16). Het argument `type` is de typetag van het advice type dat moet worden uitgevoerd. Deze functie geeft een future terug die wordt opgelost wanneer het advice is uitgevoerd. Het juiste advice type wordt eerst opgezocht op lijn 18 aan de hand van zijn typetag. Als er een advice gedefinieerd is voor dat type, kijken we eerst of de body van dit advice een synchronisatiepunt moet toevoegen. Als dit het geval is (lijn 22) houden we het pointcut bij dat het synchronisatiepunt opheft in de variabele `syncPoint`, alsook de future die moet worden opgelost wanneer dit het geval is en de uitvoering van het advice voltooid is. Wanneer de body geen pointcut bevat, voeren we de body van het advice uit, en lossen we de future meteen op.

Als er een synchronisatiepunt is toegevoegd aan een workflow door een advice, moeten we steeds nagaan wanneer er een joinpoint wordt uitgevoerd of dit synchronisatiepunt kan worden opgeheven. Dit wordt gecontroleerd door de functie `checkSyncPointReached` (lijn 41). Wanneer er een synchronisatiepunt gedefinieerd is, controleren we of dit mag worden opgeheven (lijn 43). Indien dit het geval is wordt de future opgelost (lijn 44) zodat de uitvoering van de workflow waar het synchronisatiepunt is gedefinieerd kan verdergaan. Deze future werd gedefinieerd in de functie `executeAdvice` (lijn 16).

Wanneer het advice verantwoordelijk voor het wijzigen van workflows wordt uitgevoerd, wordt de functie `getPatternAdvices` (lijn 65) opgeroepen. Deze functie geeft een tabel terug die de body's bevat van het add, remove, en replace advice. Deze body's worden pas geëvalueerd in de workflow waar ze worden toegevoegd. Dit is nodig omdat alle services niet overal zijn gedefinieerd. Wanneer er een extra branch wordt toegevoegd die één activiteit bevat, bijvoorbeeld `SupplyService.provideMaterial()`, is het mogelijk dat de service `Supply-`

`Service` wel gedefinieerd is in de workflow waar deze branch wordt toegevoegd, maar niet op de plaats waar het aspect gedefinieerd is. Daarom is het niet mogelijk om deze service hier te definiëren.

4.3 Joinpoints en uitvoering van advices in NOW

In deze sectie zoeken we de verschillende joinpoints in de implementatie van NOW waar er gecontroleerd wordt of er iets extra moet worden uitgevoerd. Wanneer er een joinpoint bereikt is, wordt er naar alle aspecten een bericht gestuurd die gaat controleren of het pointcut van een aspect dit joinpoint selecteert. We maken hiervoor gebruik van een functie `executeFunctionOnGroup` die een functie uitvoert op een verzameling van *far references*. Deze functie geeft een future terug, gedefinieerd op lijn 12 in listing 4.14, die wordt opgelost wanneer het resultaat van de functie oproep op alle *far references* bekend is. Wanneer `grp` een lege tabel is en geen enkele *far reference* bevat (lijn 13), wordt de future meteen opgelost (lijn 14). Anders wordt voor elk element van de tabel `grp` de functie `getRemoteResult` opgeroepen. Deze functie definieert een future door de functie `makeFuture()` (lijn 2), welke twee objecten terug geeft: `result` en `resolver`. Hierbij is `result` een onopgeloste future, en `resolver` een object dat wordt gebruikt om de future op te lossen, door `resolver.resolve(res)` (lijn 6). Hiervan wordt `result` teruggeven op het einde van de functie (lijn 8) en wanneer het resultaat van deze functie bekend is, wordt dit object vervangen door het echte resultaat. Op lijn 4 wordt een asynchroon bericht gestuurd naar de *far reference* door gebruik te maken van `<+`. Wanneer het resultaat hiervan bekend is, wordt eerst een anonieme functie `action` uitgevoerd, die wordt meegegeven als parameter, waarna de future van de functie wordt opgelost met het resultaat (lijn 5 en 6). Wanneer we nu terug gaan naar de `executeFunctionOnGroup` functie, zien we op lijn 19 een `when: becomes:` functie. Wanneer de resultaten van `getRemoteResult` op alle *far references* bekend zijn, wordt de future opgelost die wordt teruggeven door `executeFunctionOnGroup`. Dit gebeurt op lijn 20. Deze functie wordt gebruikt wanneer er een joinpoint bereikt is. Dan wordt naar alle beschikbare aspecten de functie `check` gestuurd, die kijkt of het joinpoint geselecteerd is door een pointcut.

```
1  def getRemoteResult(ele, function, selector, action){
2      def [result, resolver] := makeFuture();
3      def msg := createAsyncMsg( function, selector);
4      when: ele <+ msg becomes: {|res|
5          action(res, ele);
6          resolver.resolve(res);
7      };
8      result;
9  };
10 //function that runs over all aspects, and stores the relevant advices that
       need to be executed
11 def executeFunctionOnGroup(grp, function,selector, action){
12     def [result, resolver] := makeFuture();
13     if: (grp ==  []) then: {
14         resolver.resolve(true);
```

```
15      } else: {
16          def futanswers := grp.map: {|ele|
17               getRemoteResult(ele, function,selector,action);
18          };
19          when: (group: futanswers) becomes: {|res|
20               resolver.resolve(res);
21          };
22      };
23      result
24  };
```

Listing 4.14: Bericht sturen naar groep van far references.

We overlopen nu waar we in NOW de joinpoints kunnen identificeren van een activiteit, data, patroon en disconnectie. Hiernaast bespreken we ook waar we de verschillende advices moeten toevoegen die moeten worden uitgevoerd. De toevoegingen aan NOW zijn aangeduid in het blauw.

4.3.1 Joinpoint activiteit

De joinpoints die betrekking hebben bij het uitvoeren van activiteiten kunnen we identificeren in het `ServiceActivity` object in NOW. Dit object is verantwoordelijk voor het uitvoeren van een activiteit door een service in de workflow. Wanneer de uitvoering van de activiteit wordt gestart, wordt de functie `execute` opgeroepen. Als de service verantwoordelijk voor de uitvoering van de activiteit reeds ontdekt is (lijn 4), kan de activiteit worden gestart. Het uitvoeren van een activiteit is één van de joinpoints zoals gedefinieerd in sectie 3.3. Op deze positie in execute moeten we de advices voor, na, rond of gedurende de activiteit uitvoeren.

Eerst selecteren we alle aspecten waarvan het pointcut dit joinpoint selecteert. Daarom roepen we op alle aspecten de functie `check` op (lijn 14), die kijkt of het pointcut van een aspect dit joinpoint selecteert. Wanneer dit het geval is, wordt het aspect bijgehouden in de variabele `relevantAspects` op lijn 8. Als alle relevante aspecten bekend zijn, kunnen we beginnen met de uitvoering van verschillende advice types. Eerst voeren we het `before` advice uit, wat gebeurt op lijn 16. Wanneer dit `before` advice is uitgevoerd, kunnen we de uitvoering van het `around` advice starten (lijn 18). Als in dit `around` advice `proceed` wordt opgeroepen, wordt de future (lijn 18) opgelost waarna we het `during` advice starten op lijn 20. Nu kunnen we de effectieve uitvoering van de activiteit zelf starten (lijn 21). Dit wordt gedaan door de functie `invokeService` op te roepen, welke gedefinieerd is in het super-object van `ServiceActivity`, namelijk `Activity`. De functie invocatie wordt dus gedelegeerd naar het super object door `super^`. Wanneer deze activiteit is afgerond en zijn resultaat bekend is, moet het `during` advice gestopt worden (lijn 22). Hierna kan de uitvoering van het `around` advice worden verdergezet (lijn 23) en tenslotte moet nog het `after` advice worden uitgevoerd (lijn 24). Wanneer de uitvoering van dit `after` advice is voltooid, mag de activiteit zelf worden opgelost (lijn 26).

```
1  def execute(env) {
2      def notFoundTriggered := false;
3      def discov? := false;
4      when: service.tag discovered: { |s|
5          discov? := true;
6          if: !notFoundTriggered then: {
7              //added
8              def relevantAspects := [];
9              def act := {|res, aspect|
10                 if: res then: {
11                     relevantAspects := relevantAspects + [aspect];
12                 }
13             };
14             when: executeFunctionOnGroup(env.Aspects, `check , [selector],
15                                          act) becomes: {|res|
16                 when: executeAdvicesBefore(relevantAspects, env)
17                   becomes: {|resTwo|
18                     when: executeAdvicesAroundBefore(relevantAspects, env)
19                       becomes:{ |resThree|
20                         startAdviceDuring();
21                         when: super^invokeService(s, env)  becomes: { |nEnv|
22                             stopAdviceDuring();
23                             executeAdvicesAroundAfter(relevantAspects,nEnv);
24                             when: executeAdvicesAfter(relevantAspects,nEnv)
25                                 becomes: {|resfour|
26                                     super.resolver.resolve(nEnv);
27                                 };
28                             };
29                         };
30                     };
31                 };
32             };
33         };
34     };
35     def duration := getNotFoundDuration(env);
36     when: seconds(duration) elapsed: {
37         // The service 's not found.
38         if: !discov? then: {
39             // In order to deactivate the when: discovered: event handler.
40             notFoundTriggered := true;
41             notFoundOccured(env);
42         };
43     };
44 };
```

Listing 4.15: Joinpoint activiteit.

4.3.2 Joinpoint data

Het joinpoint waar de data in de omgeving van de workflow wordt gewijzigd vinden we terug in de functie `invokeService`, welke zich bevindt in het object `Activity`. Het `ServiceActivity` object beschreven in de vorige sectie is een uitbreiding van dit `Activity` object. We selecteren eerst alle relevante aspecten waarvan de pointcuts dit joinpoint selecteren. Dit gebeurt door de functie `checkDataAddedPointcut`, welke een future teruggeeft (lijn 2 en 36). Bij de argumenten op lijn 1 is `reply` het resultaat van de activiteit die is uitgevoerd, en `output` de variabelen waaraan deze resultaten moeten gekoppeld worden. Wanneer de activiteit geen resultaat moet teruggeven (lijn 4), lossen we de future van deze functie op met een lege tabel (lijn 34). Wanneer er wel resultaten zijn, kijken we of er pointcuts zijn die deze data selecteren. Daarom nemen we eerst de naam van deze variabele waaraan het resultaat wordt toegekend op lijn 10, waarna we kijken of deze variabele reeds is toegevoegd aan de omgeving van de workflow. Indien dit het geval is, taggen we een `isolate` object die de naam van de variabele bevat met de typetag `changed`. Wanneer de variabele voor het eerst wordt toegevoegd aan de omgeving, gebruiken we de typetag `added`. We taggen deze `isolate` omdat dit ons eenvoudig toelaat een onderscheid te maken tussen data die voor het eerst wordt toegevoegd of niet. Hierna wordt deze `isolate` naar het aspect gestuurd zodat kan gecontroleerd worden of zijn pointcut dit joinpoint selecteert, wat gebeurt door de functie `executeFunctionOnGroup` (lijn 24). Als dit het geval is, wordt dit aspect bijgehouden in de variabele `relevantAspects` (lijn 3). Wanneer alle aspecten het pointcut gecontroleerd hebben, wordt de future van deze functie opgelost met alle relevante aspecten. Dit gebeurt op lijn 31.

```
1 def checkDataAddedPointcut(reply, output, aspects, env){
2  def [result, resolver] := makeFuture();
3  def relevantAspects := [];
4  if: !(reply == nil) then: {
5      if: (output.length == 1) then: {
6          reply := [reply];
7      };
8      def allFut := output.map: { |val|
9          def [resultTwo, resolverTwo] := makeFuture();
10         def dataname := val.variable;
11         def dataAdvices := [];
12         def insertType := added;
13         if: (env.find(dataname) != nil) then: {
14             insertType := changed
15         };
16         def dataObject := isolate: {
17               def name := dataname;
18         } taggedAs: [insertType];
19         def act := {|res, aspect|
20              if: res then: {
21                  relevantAspects := relevantAspects + [aspect];
22              }
23         };
24         when: executeFunctionOnGroup(aspects, `check ,
```

```
25                                      [dataObject], act) becomes: {|res|
26                 resolverTwo.resolve(true);
27             };
28             resultTwo
29         };
30             when: (group: allFut) becomes: {|res|
31               resolver.resolve(relevantAspects)
32         };
33     } else: {
34         resolver.resolve([]);
35     };
36     result
37 };
```

Listing 4.16: Helpfunctie checkDataAddedPointcut.

Wanneer we nu naar de functie `invokeService` kijken, merken we op dat de nieuwe data wordt toegevoegd aan de omgeving wanneer de activiteit is afgerond (lijn 15). Hierna moeten we dus eerst controleren of er aspecten zijn die een pointcut bevat die dit joinpoint selecteert. Dit wordt gedaan door de functie `checkDataAddedPointcut` uit te voeren (lijn 17), die een future teruggeeft. Wanneer het resultaat hiervan bekend is en de future is opgelost, alle relevante aspecten zijn nu beschikbaar (lijn 19), kunnen we de advices uitvoeren. Bij dit joinpoint worden er twee advice types uitgevoerd, het `before` en `after` advice. Eerst wordt het `before` advice uitgevoerd (lijn 20), waarna de data wordt toegevoegd aan de omgeving (lijn 22), en tenslotte het `after` advice wordt uitgevoerd (lijn 23).

```
1  // Invocation of a service.
2  def invokeService(service, env) {
3      def [result, resolver] := makeFuture();
4      def arguments := self.parameters;
5      def output := self.output;
6      def msg := createAsyncMsg(self.selector, env.bind(arguments));
7      def duration := self.getTimeoutDuration(env);
8      // When a timeout failure is specified, create asynchronous message with
           @Due annotation
9      if: !(duration == nil) then: {
10         msg := createTimeoutMsg(self.selector, env.bind(arguments), duration
               );
11     };
12     // In order to cancel TimeoutException in case of a disconnection error.
13     def disconnection := false;
14
15     when: service <+ msg becomes: { |reply|
16         if: !disconnection then: {
17             def relevantAspectsAnswer :=
18                   checkDataAddedPointcut(reply, output, env.Aspects, env);
19             when: relevantAspectsAnswer becomes: { |relevantAspects|
20                 when: executeAdvicesBefore(relevantAspects, env)
```

```
21                    becomes:{ |res|
22                        env.insertOutputValues(reply, output);
23                        when: executeAdvicesAfter(relevantAspects,env)
24                          becomes:{ |resTwo|
25                              resolver.resolve(env);
26                    };
27                };
28            };
29        };
30    } catch: Exception using: { |exception|
31        ...
32    };
33    when: service disconnected: {
34        //see listing 4.19
35    };
36    result;
37 };
```

Listing 4.17: Functie invokeService in object Activity.

4.3.3 Joinpoint patroon

Wanneer we de structuur van de workflow wijzigen, zoals bijvoorbeeld de branches in een parallelle split, moeten we kijken naar de implementatie van dat patroon. In listing 4.18 zien we twee functies die behoren tot het object van een parallelle split. De functie `ev` op lijn 1 evalueert een branch in de omgeving waar deze workflow zich bevindt. We kunnen dit niet doen op de plaats waar het aspect gedefinieerd is omdat sommige services niet op alle vaste infrastructuren zijn gedefinieerd. Wanneer de parallelle split wordt uitgevoerd door de `execute` functie op lijn 5, selecteren we eerst de aspecten die een pointcut bevatten die deze parallelle split selecteert (lijn 12). Wanneer deze aspecten gekend zijn, vragen we hieraan het *add* advice en het *remove* advice. Dit gebeurt door de functie `executeFunctionOnGroup` (lijn 20) die de functie `getPatternAdvices` oproept op alle relevante aspecten. De advices die dit als resultaat teruggeven worden bijgehouden in de variabele `relevantAdvices`. Elk element van `relevantAdvices` is een tabel van grootte drie, die eventueel een branch bevat die moet worden toegevoegd, verwijderd of gewijzigd. We evalueren eerst elk van deze branch (lijn 23 en 24), waarna we de branches die moeten verwijderd worden effectief verwijderen op lijn 25, en de branches die moeten worden toegevoegd toevoegen op lijn 26. Hierna kan de uitvoering van alle branches gestart worden (lijn 28 tot 42).

```
1 def ev(branch, env){
2     eval: branch in: env.getLexRoot().context.lexicalScope;
3 };
4
5 def execute(env) {
6     def relevantAspects := [];
7     def act := {|res, aspect|
```

```
8           if: res then: {
9               relevantAspects := relevantAspects + [aspect];
10          }
11      };
12      when: executeFunctionOnGroup(env.aspects, `check , [ParallelSplit], act)
13       becomes: {|res|
14          def relevantAdvices := [];
15          def act2 := {|res, aspect|
16              if: res then: {
17                  relevantAdvices := relevantAdvices + [res];
18              }
19          };
20          when: executeFunctionOnGroup(relevantaspect, `getPatternAdvices, [],
21                                       act2) becomes: { |res|
22              relevantAdvices.each: {|adv|
23                  def toAdd := ev( adv[1], env);
24                  def toRemove := ev( adv[2], env);
25                  components := components.filter: {|ele| ele != toRemove};
26                  components := components + [toAdd];
27              };
28              1.to: components.length+1 do: { | idx |
29                  def clonedEnv := SystemEnvironment.new(env);
30                  clonedEnv.id := env.id;
31                  clonedEnv.splitEnv := env;
32                  clonedEnv.branchNr := idx;
33                  def component :=
34                     components[idx].getClone(env, components[idx]);
35                  instanceComponents := instanceComponents + [component];
36                  when: component.start(clonedEnv) becomes: { |nEnv|
37                              finished := finished + 1;
38                      if: (finished == components.length) then: {
39                           super.resolver.resolve(nEnv);
40                      };
41                  };
42              };
43          };
44      };
45  };
```

Listing 4.18: Joinpoint parallelle split patroon.

4.3.4 Joinpoint disconnectie

Wanneer er een disconnectie van een service plaatsvindt, wordt dit opgevangen door `when: disconnected:` in de functie `invokeService`. Deze functie is reeds besproken in sectie 4.3.2 en listing 4.17. We laten nu aan alle aspecten weten dat er een disconnectie is van een service, waarvoor wordt gezorgd op lijn 9 en 10. Hierna wordt de functie `disconnectio-`

nOccured opgeroepen (lijn 12) die eventueel een compensatie uitvoert in NOW wanneer deze gedefinieerd is.

```
1  def invokeService(service, env) {
2          ...
3      when: service disconnected: {
4          // In order to cancel TimeoutException in case of a disconnection
               error.
5          disconnection := true;
6
7          ...
8
9          env.Aspects.each{ |aspect|
10             aspect<-disconnection(servicetag);
11         };
12         self.disconnectionOccured(service, env);
13     };
14     result
15 };
```

Listing 4.19: Joinpoint disconnectie.

4.4 Ontdekken van aspecten

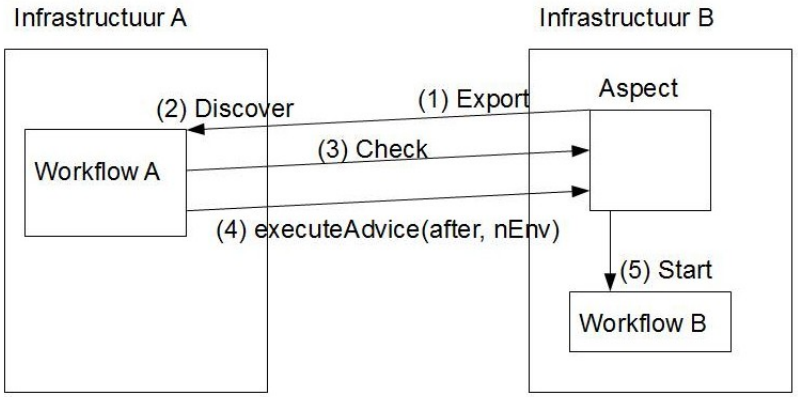
Wanneer een nieuw aspect is aangemaakt, wordt dit meteen geëxporteerd, waardoor het kan ontdekt worden door andere vaste infrastructuren. Dit gebeurt door het SystemEnvironment object in NOW (listing 4.20), dat specifieke workflow informatie bevat, zoals data, en wordt doorgegeven doorheen de volledige workflow. Als een nieuw SystemEnvironment wordt aangemaakt, wordt er eerst de init methode uitgevoerd. Deze methode zal het nieuwe object initialiseren. Wanneer er een kopie moet gemaakt worden van een andere omgeving, wordt deze meegegeven als argument. Dit kan voorkomen wanneer er een parallelle split wordt uitgevoerd waarbij iedere branch zijn eigen omgeving heeft. Aan het einde van deze functie maken we gebruik van whenever: discovered: (lijn 20) die elk object, geëxporteerd als Aspect, gaat ontdekken. asp is dan een far reference naar het ontdekte aspect. We slagen deze allemaal op in de tabel van de variable Aspects. We kijken eerst of dit object nog niet ontdekt is (lijn 21). Wanneer dit niet het geval is, voegen we de far reference van dit aspect toe aan Aspects (lijn 23), en laten het aspect weten dat het ontdekt is door deze omgeving (lijn 24), zodat het ook kan weten wanneer er hiermee een disconnectie plaatsvindt.

Figuur 4.3 geeft een overzicht van het exporteren van een aspect naar een andere infrastructuur. Wanneer een aspect, gedefinieerd op vaste infrastructuur B, geëxporteerd is (1) kunnen andere vaste infrastructuren, in het voorbeeld A, dit aspect ontdekken (2). Tijdens de uitvoering van een workflow, gedefinieerd op vaste infrastructuur A, wordt wanneer een joinpoint bereikt is steeds de functie check opgeroepen op het aspect (3). Wanneer het aspect zijn pointcut dit

```
1  def SystemEnvironment := extend: Dictionary with: {
2
3          ...
4
5      def Aspects := [];
6
7      /**
8      * Initialise a new system environment
9      * @param envsToCopy: optional environments whose values that need to be
           inserted in this one
10     */
11     def init(@envsToCopy) {
12
13         ...
14
15         if: (envsToCopy.length > 0) then: {
16             ...
17             self.lexRoot := env.lexRoot;
18             self.Aspects := env.Aspects;
19         };
20         whenever: Aspect discovered: { |asp|
21             def idx := Aspects.find: { |s| s == asp};
22             if: (idx == nil) then: {
23                 Aspects := Aspects + [asp];
24                 asp<-addFar(self);
25             };
26         }
27     };
28
29     ...
30
31 } taggedAs: [Environment];
```

Listing 4.20: Ontdekken van aspecten.

joinpoint selecteert wordt het advice uitgevoerd, in het voorbeeld het after advice (4). Dit advice zal de uitvoering van een nieuwe workflow starten op vaste infrastructuur B (5).



Figuur 4.3: Overzicht exporteren van aspecten.

4.5 Besluit

In dit hoofdstuk hebben we de implementatie van onze aspectgeoriënteerde oplossing overlopen. We bespraken eerst de belangrijkste technieken van AmbientTalk die we gebruikt hebben. Hierna hebben we het design van het aspect overlopen, waarna we de implementatie van het aspect gepresenteerd hebben. Hierbij lag de nadruk op de implementatie van pointcuts en advices. Vervolgens identificeerden we de verschillende joinpoints in NOW, en waar we juist de uitvoering van de verschillende advices moeten toevoegen. Tenslotte bespraken we hoe de aspecten kunnen worden toegepast op andere vaste infrastructuren, namelijk door het exporteren ervan.

	oplossing	sectie
Vereiste 1	PointcutMethod	4.2.1
Vereiste 2	PointcutData, PointcutDataAdded, PointcutDataChanged	4.2.1
Vereiste 3	intersection, union	4.2.1
Vereiste 4	advice types: before, after, during, action, around	4.2.2 - 4.3.1
Vereiste 5	advice types: add, remove, replace	4.2.2 - 4.3.3
Vereiste 6	advice types: before, after	4.2.2
Vereiste 7	PointcutDisconnected	4.2.1

Tabel 4.1: Overzicht implementatie van vereisten.

Hoofdstuk 5

Validatie

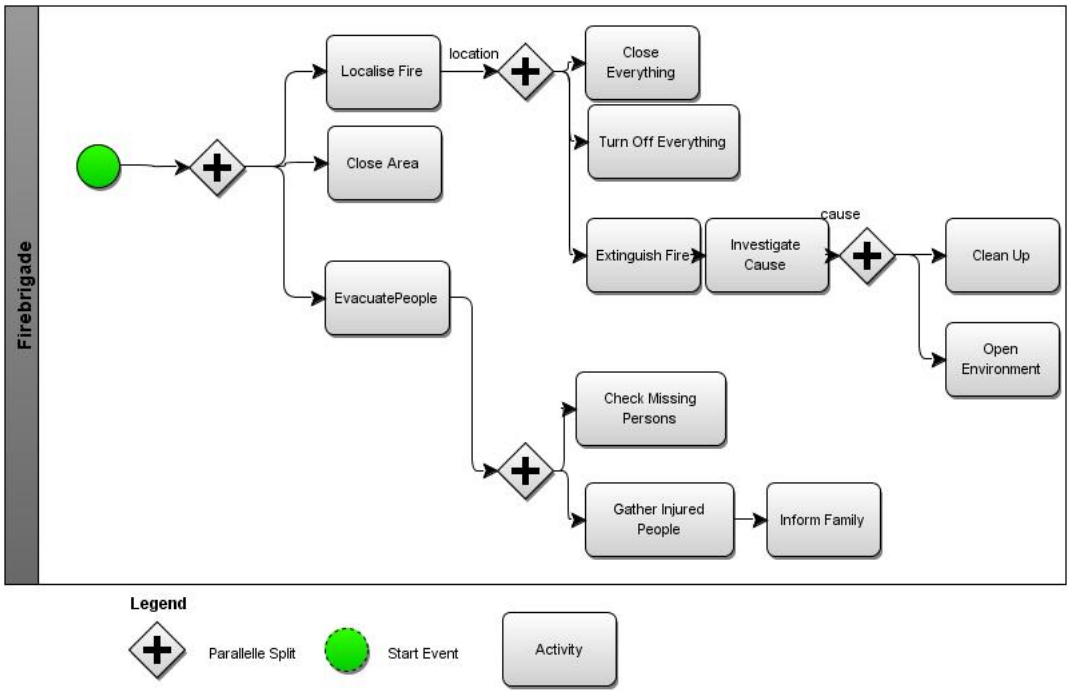
In dit hoofdstuk valideren we onze oplossing en kijken we of ze aan alle nodige voorwaarden, besproken in sectie 3.2, voldoet. Hiernaast vergelijken we onze oplossing met NOW, en kijken we naar de voordelen en nadelen van beiden. We bespreken twee verschillende scenario's. Het eerste scenario gaat over een uitslaande brand waarbij de hulpdiensten ter plaatse komen en starten met het bestrijden ervan. Dit scenario is reeds beschreven en gebruikt in sectie 3.1. Het tweede scenario gaat over het uitvoeren van wegenwerken. We implementeren de scenario's eerst door enkel gebruik te maken van NOW, terwijl we hierna ook gebruik maken van de aspectgeoriënteerde oplossing.

5.1 Scenario brand

Het eerste scenario vindt plaats wanneer de hulpdiensten moeten langskomen omdat er een gebouw in brand staat. De verschillende hulpdiensten komen ter plaatse om deze brand te bestrijden. De brandweer zal de brand blussen, de medische dienst is verantwoordelijk voor het verzorgen van de gewonden en de politie zal de oorzaak en eventueel verantwoordelijke van de brand onderzoeken. Elk van deze drie hulpdiensten hun workflow is reeds besproken in sectie 3.1. Deze hulpdiensten zullen nu ook samenwerken. Wanneer de brandweer de oorzaak van de brand onderzocht heeft, kan de workflow van de politie starten, en wanneer alle gewonde personen zijn verzameld op één plaats, kan de workflow van de ambulance starten.

Wanneer de oorzaak van de brand onderzocht is wordt de workflow van de politie gestart. Deze zal eerst kijken naar de oorzaak en indien nodig een onderzoek openen. Nu is het mogelijk dat de politie nog extra bewijs nodig heeft en aangezien de brandweer veel meer expertise ter plaatse heeft zullen zij dit bewijs zoeken. Daarom mag de brandweer na het onderzoeken van de brand nog niet meteen starten met alles op te ruimen, maar moeten ze wachten tot de politie bepaald heeft of er kwaad opzet in het spel is. Aangezien er iets extra moet gedaan worden door de brandweer, wat bepaald is door de politie, zal dit niet aanwezig zijn in de workflow, en moet dit nog worden toegevoegd.

Eerst definiëren we deze drie verschillende workflows in NOW, welke we zowel gebruiken in de aanpak met enkel NOW, als wanneer we de aspectgeoriënteerde oplossing toepassen. De eerste workflow is deze van de brandweer.



Figuur 5.1: Workflow brandweer.

```
1 def WFenv := SystemEnvironment.new();
2
3 def ExtinguishService := defService('ExtinguishService);
4 def SupplyService := defService('SupplyService);
5 //================================
6
7 def parSplit1 := ParallelSplit(
8      SupplyService.checkMissingPersons(),
9      Sequence( SupplyService.gatherInjuredPeople(),
10               SupplyService.informFamily() ) );
11
12 def parSplit2 := ParallelSplit(
13     Sequence( ExtinguishService.extinguishFire(),
14               ExtinguishService.investigateCause()@Output(Env.cause),
15               ParallelSplit(
16                   ExtinguishService.cleanUp(),
17                   SupplyService.openEnvironment() ) ),
18     Sequence( ExtinguishService.closeEverything() ),
19     Sequence( ExtinguishService.turnOffEverything() ) );
20
21 def ParSplitStart := ParallelSplit(
22     Sequence( ExtinguishService.localiseFire()@Output(Env.location),
```

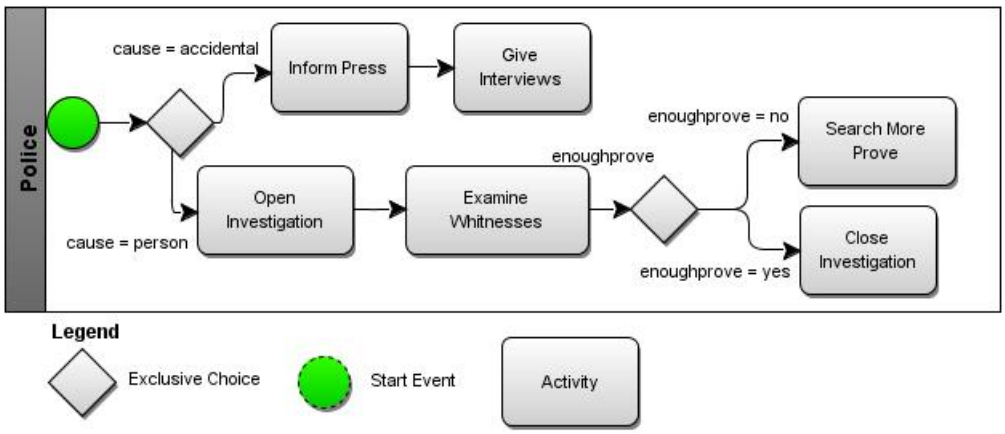
```
23                parSplit2 ),
24      Sequence( SupplyService.evacuatePeople(),
25                parSplit1 ),
26      ExtinguishService.closeArea() );
```

Listing 5.1: Workflow brandweer.

Op lijn 1 maken we eerst een nieuwe workflow omgeving aan, terwijl we op lijn 3 en 4 de verschillende services definiëren die nodig zijn. Vervolgens starten we met de definitie van de workflow voor de brandweer. Deze zal starten op lijn 21 met een parallelle split die 3 branches bevat. De eerste branch (lijn 22) is een sequence die eerst een activiteit uitvoert die de brand lokaliseert en hierna een parallelle split (gedefinieerd op lijn 12) start. Deze zal drie branches bevatten. De eerste branch (lijn 13) zal eerst de brand blussen, de oorzaak bepalen en daarna gelijktijdig beginnen opruimen en de afgezette omgeving terug openzetten. De tweede branch (lijn 18) voert een activiteit uit die alle ramen en deuren sluit, en de derde branch (lijn 18) zal alle nutsvoorzieningen, zoals gas en elektriciteit, afsluiten.

De tweede branch van de eerste parallelle split van de workflow, gedefinieerd op lijn 24, is een sequence die eerst alle personen evacueert, en daarna een parallelle split uitvoert. Deze parallelle split (lijn 7) bevat 2 branches, namelijk één die controleert of er vermiste personen zijn, en de andere zal eerst alle gewonde personen verzamelen op één plaats (lijn 9), gevolgd door het verwittigen van de familieleden (lijn 10).

De derde branch van de parallelle split op lijn 26 zal de volledige omgeving afzetten. Nu overlopen we eerst de workflow van de politie:



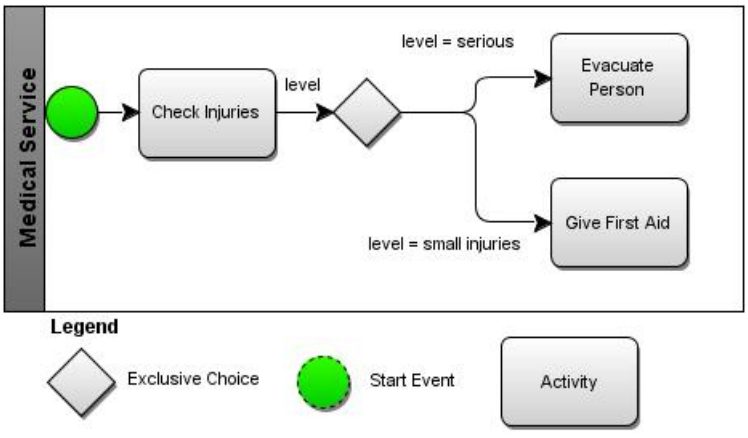
Figuur 5.2: Workflow politie.

```
1 def PoliceCommunicationService := defService('PoliceCommunicationService);
2 def PoliceInvestigateService := defService('PoliceInvestigateService);
3 //==================================
4
5 def seq1 := Sequence( PoliceCommunicationService.informPress(),
6                       PoliceCommunicationService.giveInterviews() );
7
8 def seq2 := Sequence( PoliceInvestigateService.openInvestigation(),
9                       PoliceInvestigateService.examineWhitnesses()@Output(
                           env.enoughProve),
10                      ExclusiveChoice({|enoughProve| enoughProve == "Yes" },
11                          PoliceInvestigateService.closeInvestigation(),
12                          PoliceInvestigateService.searchMoreProve() ) );
13
14
15 def policeStart := ExclusiveChoice( { |cause| cause == "accidental" }, seq1,
       seq2);
```

Listing 5.2: Workflow politie.

Op lijn 1 en 2 definiëren we eerst twee services. De workflow van de politie start met een exclusiveChoice op lijn 15. Als de oorzaak van de brand een ongeval is, zal de sequence op lijn 5 worden uitgevoerd. Hier zal eerst de pers worden ingelicht door de politie (lijn 5), waarna er interviews worden gegeven (lijn 6). Wanneer de brand geen ongeval is, wordt een sequence op lijn 8 uitgevoerd. Hier wordt eerst een onderzoek geopend door de politie (lijn 8), waarna getuigen ondervraagd worden (lijn 9) en bepaald wordt of er reeds voldoende bewijslast is gevonden (lijn 10). Als er voldoende bewijs gevonden is wordt het onderzoek afgerond (lijn 11), anders wordt er nog gezocht naar extra bewijs (lijn 12).

Tenslotte hebben we nog de workflow van de medische dienst:



Figuur 5.3: Workflow medische hulpdienst.

```
1 def AmbulanceService := defService('AmbulanceService);
2 //==================================
3
4 def seq1 := Sequence( AmbulanceService.evacuatePerson() );
5 def seq2 := Sequence( AmbulanceService.giveFirstAid() );
6
7 def exc := ExclusiveChoice( { |level| level == "serious" }, seq1, seq2);
8
9 def ambulanceStart := Sequence( AmbulanceService.checkInjuries()@Output(Env.
    level),
10                                  exc );
```

Listing 5.3: Workflow medische dienst.

We definiëren eerst de ambulance service op lijn 1. De start van de workflow is gedefinieerd op lijn 9. Eerst zullen de verwondingen van de gekwetste persoon worden onderzocht, waarbij de graad van deze verwondingen bepaald wordt. Vervolgens hebben we een exclusiveChoice (lijn 7), die deze graad nodig heeft. Wanneer de verwondingen ernstig zijn, zal de persoon worden geëvacueerd naar het ziekenhuis (lijn 4). Wanneer dit niet het geval is, zal de persoon eerste hulp ter plaatse krijgen (lijn 5).

5.1.1 Workflows starten

Laten we nu eerst kijken naar het geval waarbij een andere workflow moet gestart worden. Wanneer we dit willen verwezenlijken door enkel gebruik te maken van NOW, moeten we een trigger toevoegen die de naam van de workflow als parameter heeft op de plaats wanneer we deze willen starten. Daarom zullen we bij de workflow van de brandweer bij de sequence op lijn 13 een trigger toevoegen na de *investigateCause* activiteit, wat in de volgende listing op lijn 7 gebeurt. Wanneer dit punt bereikt is, zal de workflow van de politie, gedefinieerd als de variabele `policeStart`, gestart worden. We doen hetzelfde voor het starten van de workflow van de medische dienst. Deze kan gestart worden wanneer de activiteit `gatheredInjuredPeople` is afgerond in de workflow van de brandweer op lijn 9. Daarom zullen we na deze activiteit een trigger toevoegen die de workflow van de medische diensten, gedefinieerd als *ambulanceStart*, zal starten. Dit gebeurt in de volgende listing op lijn 2.

```
1 Sequence( SupplyService.gatherInjuredPeople(),
2           Trigger(ambulanceStart),
3           SupplyService.informFamily() );
4
5 Sequence( ExtinguishService.extinguishFire(),
6           ExtinguishService.investigateCause()@Output(Env.cause),
7           Trigger(policeStart),
8           ... );
```

Listing 5.4: Andere workflow starten in NOW.

Het nadeel hieraan is dat we op verschillende plaatsen in de workflow een trigger moeten toevoegen, die een andere workflow start gedefinieerd op dezelfde vaste infrastructuur. Wanneer we geen referentie hebben van een workflow die is gedefinieerd op een andere vaste infrastructuur, kunnen we deze niet starten. Wanneer elke hulpdienst zoals in ons voorbeeld zijn eigen vaste infrastructuur heeft, is dit dus niet mogelijk. Een ander nadeel is dat wanneer een workflow gestart is we deze niet meer kunnen wijzigen. Het is onmogelijk om dan nog een trigger toe te voegen.

Laten we nu een andere workflow starten door gebruik te maken van een aspect. Op lijn 1 maken we een nieuw aspect aan, waarbij we vervolgens zijn pointcut toevoegen op lijn 2. Hierna bepalen we op lijn 3 het after advice. De body die zal worden uitgevoerd, is een anonieme functie welke als parameter de omgeving meekrijgt van de workflow waar het pointcut bereikt wordt. Aangezien de workflow van de politie deze nodig heeft, zal de data van deze andere omgeving eerst worden toegevoegd aan de omgeving van de politie. Vervolgens kan de workflow van de politie worden gestart. Hetzelfde geldt voor de workflow van de medische dienst, waar het aspect gedefinieerd is op lijn 6. Daarna bepalen we op lijn 7 het pointcut, namelijk de activiteit `gatherInjuredPeople`, waarna we op lijn 8 het advice definiëren. Ook hier voegen we eerst de omgeving toe aan de workflow van de medische dienst, waarna we deze starten.

```
1 def aspectpolice := Aspect();
2 aspectpolice.setPointcut(PointcutMethod("InvestigateCause"));
3 aspectpolice.setAdviceAfter({|env| addToEnv(env, WFenv);
4                                   policeStart.start(WFenv);});
5
6 def aspectmedicalservice := Aspect();
7 aspectmedicalservice.setPointcut(PointcutMethod("gatherInjuredPeople"));
8 aspectmedicalservice.setAdviceBefore({|env| addToEnv(env, WFenv);
9                                             ambulanceStart.start(WFenv);});
```

Listing 5.5: Andere workflows starten door aspecten.

Deze aspecten zullen elk op hun eigen vaste infrastructuur worden gedefinieerd. Hierdoor moeten niet alle workflows op dezelfde vaste infrastructuur worden uitgevoerd, maar kunnen deze onafhankelijk van elkaar blijven. Het is ook mogelijk om een aspect toe te voegen wanneer een workflow reeds gestart is, zodat we niet alles op voorhand moeten definiëren, en wanneer de workflow gestart is nog steeds kunnen bepalen wanneer er andere workflows moeten starten.

5.1.2 Synchronisatiepunt toevoegen

Het tweede geval waarbij we een vergelijking maken tussen NOW en de aspectgeoriënteerde oplossing is wanneer er een synchronisatie nodig is tussen de verschillende workflows. Het is mogelijk dat de brandweer nog een taak extra moet uitvoeren afhankelijk van het onderzoek van de politie. Daarom zal er nadat de oorzaak onderzocht is door de brandweer een synchronisatiepunt nodig zijn, dat pas zal voltooid zijn wanneer ofwel de pers geïnformeerd is, ofwel de politie getuigen ondervraagd heeft en al weet of er nog iets extra's moet gebeuren.

Als we dit in NOW willen implementeren, moeten we connecties en synchronisaties toevoegen. Wanneer we een synchronisatiepunt hebben, moeten alle connecties met dit punt voltooid zijn voordat de synchronisatie geslaagd is. Op lijn 1 in listing 5.6 zullen we een nieuw synchronisatiepunt aanmaken dat wordt toegevoegd na de *investigateCause* activiteit (lijn 3), en wanneer dit voltooid is een parallelle split `parSplit3` zal starten. Vervolgens kunnen we met dit synchronisatiepunt een connectie maken. Dit gebeurt op lijn 7, welke een aanpassing is van de workflow van de politie. Aangezien we naast de volgende activiteit uit te voeren, *giveInterviews*, ook een connectie moeten maken met het synchronisatiepunt, zullen we een parallelle split toevoegen. Het synchronisatiepunt zal voldaan zijn wanneer deze branch in de workflow bereikt is. Wanneer we in NOW een synchronisatiepunt verbinden met meerdere connecties, en bepalen dat het synchronisatiepunt voltooid is wanneer juist één van deze connecties bereikt is, moeten we gebruik maken van een `CancellingDiscriminator`.

```
1 def sync := Synchronize(parSplit3);
2 Sequence( ExtinguishService.extinguishFire(),
3           ExtinguishService.investigateCause()@Output(Env.cause),
4           sync);
5
6 Sequence( PoliceCommunicationService.informPress(),
7           ParallelSplit(Connection(sync),
8                         PoliceCommunicationService.giveInterviews()));
```

Listing 5.6: Synchronisatie in NOW.

Wanneer we gebruik maken van een aspect, moeten we de workflows op geen enkele plaats wijzigen. Nadat we het aspect gedefinieerd hebben op lijn 1, zullen we het pointcut toevoegen op lijn 2 dat bepaalt waar het synchronisatiepunt moet worden toegevoegd. Dit zal na de activiteit `investigateCause` zijn. Vervolgens definiëren we in het advice wanneer het synchronisatiepunt voltooid is, wat ook wordt aangegeven door een pointcut. Dit gebeurt op lijn 3 waar we de unie nemen van twee pointcuts. Het synchronisatiepunt zal dus voltooid zijn wanneer of de activiteit `informPress`, of de activiteit `examineWhitnesses` voltooid is.

```
1 def aspect := Aspect();
2 aspect.setPointcut(PointcutMethod("investigateCause"));
3 aspect.setAdviceAfter( union( PointcutMethod("informPress"),
4                               PointcutMethod("examineWhitnesses")) );
```

Listing 5.7: Aspect dat synchronisatiepunt toevoegt.

Door gebruik te maken van een aspect kunnen we op eenvoudige manier definiëren waar er een synchronisatiepunt moet worden toegevoegd, en wanneer dit voltooid is. Dit kan verwezenlijkt worden zonder dat we in de verschillende workflows aanpassingen hoeven te doen. De workflows kunnen elk op hun eigen vaste infrastructuur worden uitgevoerd, aangezien het aspect niet alleen op de eigen vaste infrastructuur, maar ook op andere van toepassing is. Dit in tegenstelling tot NOW dat, wanneer we een synchronisatiepunt willen toevoegen afhankelijk van een andere

workflow, dit verwezenlijkt door gebruik te maken van *Synchronize* en *Connection*, en deze workflow enkel kan worden uitgevoerd op dezelfde vaste infrastructuur.

5.1.3 Workflow wijzigen

Als derde vergelijking hebben we nog het geval waarbij we de structuur van een andere workflow willen aanpassen. Zo is het mogelijk dat de brandweer extra bewijs moet zoeken in het afgebrande huis, zoals bijvoorbeeld brandversnellers, wanneer de politie nog niet voldoende bewijs heeft. In NOW kunnen we de workflow wijzigen voor deze wordt uitgevoerd, maar hierna is dit niet meer mogelijk. Door gebruik te maken van een aspect in NOW kunnen we dit verwezenlijken, waarvan een voorbeeld gedefinieerd is in listing 5.8. Waarna we op lijn 1 een aspect gedefinieerd hebben, bepalen we op lijn 2 het pointcut van dit aspect, namelijk de activiteit `searchMoreProve`. Hierna bepalen we op lijn 3 het advice type, namelijk een before advice. Juist voor de `searchMoreProve` activiteit wordt gestart, zal de anonieme functie worden uitgevoerd. Hierin zullen we dan een aspect definiëren dat een extra branch toevoegt in een parallelle split. Op lijn 4 wordt een nieuw aspect aangemaakt, waarna we op lijn 5 het pointcut definiëren die bepaalt welk patroon van de workflow moet gewijzigd worden, namelijk een parallelle split. Vervolgens bepalen we op lijn 6 wat er moet gewijzigd worden, namelijk iets toegevoegen. Omdat we reeds aangegeven hebben dat het advice op een parallelle split wordt toegepast zal er dus een branch worden toegevoegd. Deze branch zal één extra activiteit uitvoeren, namelijk de activiteit `searchProve`.

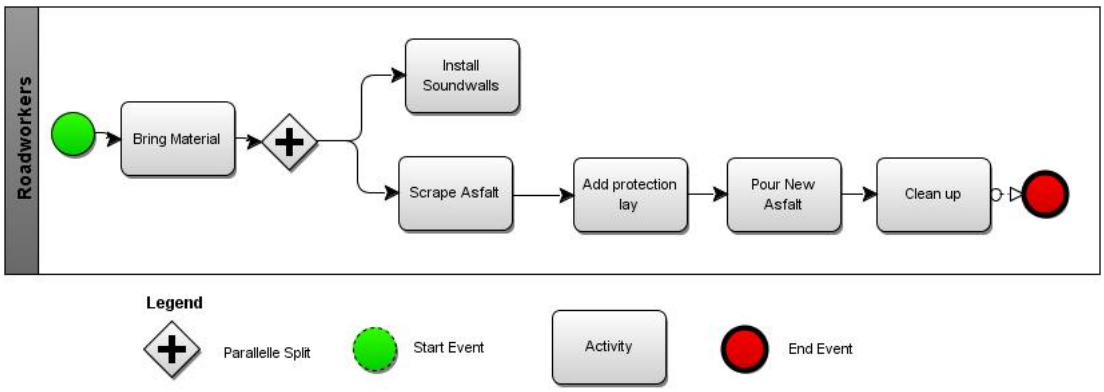
```
1 def aspect := Aspect();
2 aspect.setPointcut(PointcutMethod("searchMoreProve"));
3 aspect.setAdviceBefore( {|env|
4         def aspecttwo := Aspect();
5         aspecttwo.setPointcut(PointcutPattern(ParallelSplit));
6         aspecttwo.setAdviceAdd( `ExtinguishService.searchProve() ); });
```

Listing 5.8: Extra branch toevoegen.

5.2 Scenario wegenwerken

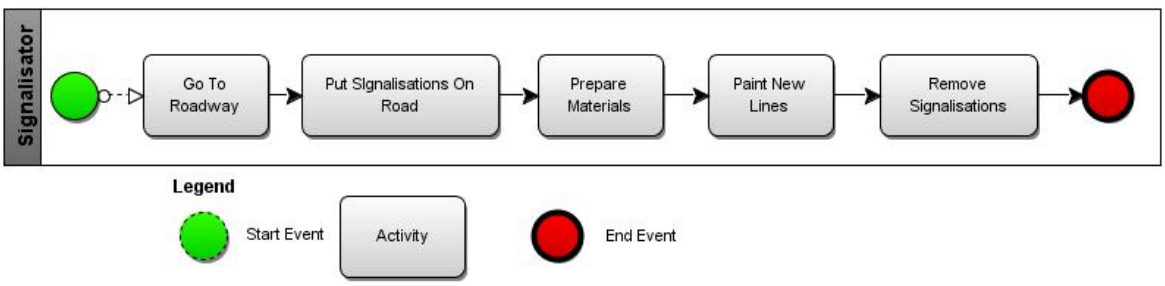
Voor het tweede scenario kijken we naar een omgeving waar een autosnelweg wordt vernieuwd. Om dit op een zo efficiënt mogelijke manier te verwezenlijken, werken verschillende diensten samen. Zo is er één team verantwoordelijk voor het heraanleggen van de nieuwe weg, waarbij het tweede team zorgt voor de signalisatie ervan. In figuur 5.4 hebben we een overzicht van de taken die de wegenarbeiders moeten uitvoeren voor het heraanleggen van de weg. Wanneer de werken starten, moeten ze eerst al het nodige werkmateriaal terplaatse brengen. Hierna kan effectief begonnen worden met de start van de werken. De arbeiders worden opgesplitst in twee groepen, waarbij één groep verantwoordelijk is voor het plaatsen of vervangen van geluidsmuren langs de zijkant van de weg, en de andere groep legt zich toe op het vernieuwen van de weg. Hierbij wordt eerst de bestaande asfaltlaag afgeschraapt, waarna de weg wordt bedekt met een speciale

laag wat zorgt voor een betere demping van de auto's die over de weg rijden. Hierna wordt de nieuwe laag asfalt gegoten. Wanneer dit afgerond is, kan worden gestart met het opkuisen en opruimen van alle materiaal. De workflow die dit scenario voorstelt, start met het uitvoeren van één activiteit, namelijk het terplaatse brengen van het werkmateriaal, waarna er een parallelle split is die aanduidt dat de werken aan de geluidsmuur en de wegenwerken in parallel uitgevoerd moeten worden.



Figuur 5.4: Workflow wegenarbeiders.

Het tweede team dat aanwezig is bij het uitvoeren van de wegenwerken is verantwoordelijk voor de signalisatie. Deze verplaatsen zich eerst naar het begin van de wegenwerken, waarna ze de nodige verkeersborden en nieuwe wegmarkeringen aanbrengen zodat het verkeer veilig kan passeren en alle werknemers in veiligheid kunnen werken. Wanneer dit afgerond is, moeten ze wachten totdat de eerste wegenwerken bijna zijn afgerond, waarna ze kunnen starten met de preparatie van het nodige materiaal, zoals bijvoorbeeld de verf voor het trekken van de lijnen. Wanneer het nieuwe asfalt gegoten is door het andere team, mag men beginnen met het verven van deze nieuwe lijnen. Nadat dit is afgerond moet men nog de geplaatste verkeersborden en markeringen opnieuw verwijderen. In figuur 5.5 zien we de workflow die de taken van het signalisatieteam in de juiste volgorde weergeeft. Er wordt enkel gebruik gemaakt van het *sequence* patroon waarbij alle taken na elkaar worden uitgevoerd.



Figuur 5.5: Workflow signalisators.

We definiëren nu eerst beide workflows in NOW. De workflow in listing 5.9 is de implementatie van de workflow van de wegenarbeiders in NOW. We definiëren eerst op lijn 1 en 2 de services die de taken van de wegenarbeiders uitvoeren. Hierna definiëren we op lijn 4 een `sequence` die een activiteit bevat waarbij de geluidsmuur wordt geïnstalleerd. Vervolgens definiëren we op lijn 5 een `sequence` die vier activiteiten na elkaar uitvoert. Op lijn 3 hebben we de start van de workflow van de wegenarbeiders, die eerst een activiteit uitvoert die al het nodige materiaal ter plaatse brengt, gevolgd door een parallelle split (lijn 10), die twee branches bevat. Deze zijn gedefinieerd op lijn 4 en 5, en voeren de wegenwerken en het plaatsen van geluidsmuren in parallel uit.

```
1  def WallWorkersService := defService('WallWorkersService);
2  def RoadWorkersService := defService('RoadWorkersService);
3
4  def seqWallTeam := Sequence( WallWorkersService.installSoundWall());
5  def seqRoadTeam := Sequence( RoadWorkersService.scrapeAsfalt(),
6                               RoadWorkersService.addProtectionLay(),
7                               RoadWorkersService.pourNewAsphalt(),
8                               RoadWorkersService.cleanUp());
9
10 def par := ParallelSplit(seqWallTeam, seqRoadTeam);
11 def roadWorkersStart := Sequence( RoadWorkersService.bringMaterial(), par);
```

Listing 5.9: Workflow wegenarbeiders.

In listing 5.10 is de workflow van de signalisators in NOW gedefinieerd. Op lijn 1 is de service gedefinieerd die de activiteiten van de signalisators uitvoert, namelijk `SignalService`. De workflow zelf bestaat uit één sequence op lijn 3, die vijf activiteiten na elkaar uitvoert. De eerste activiteit zorgt ervoor dat men naar de startplaats van de wegenwerken gaat (lijn 3). Vervolgens gaat men signalisatie op de weg plaatsen (lijn 4). Hierna maakt men het materiaal klaar (lijn 5) waarmee later de nieuwe verkeerslijnen kunnen worden getrokken (lijn 6). Tenslotte wordt de signalisatie verwijderd (lijn 7).

```
1 def SignalService := defService('SignalService);
2
3 def seqRoadTeam := Sequence( SignalService.goToRoadway(),
4                              SignalService.putSignalizationOnRoad(),
5                              SignalService.prepareMaterials(),
6                              SignalService.paintNewLines(),
7                              SignalService.removeSignalizations())
```

Listing 5.10: Workflow signalisators.

5.2.1 Falingen

Door het gebruik van mobiele netwerkverbindingen is de kans op netwerkdisconnecties zeer groot. Dit kan grote gevolgen hebben voor de uitvoering van workflows. Wanneer de service

```
1 def alt := SupplyService.checkNeeds();
2
3 def seqWallTeam := Sequence(
4                       Failure( WallWorkersService.installSoundWall()
5                                [ Description(Disconnection(), Alternative(
                                     alt)) ]));
```

Listing 5.11: Voorbeeld faling in NOW.

die verantwoordelijk is voor het plaatsen van geluidsmuren is gedisconnecteerd, is het mogelijk dat bepaald materiaal niet meer op tijd ter plaatse geraakt en men niet meer kan verder werken. Daarom wordt er in dat geval een extra arbeider ter plaatse gestuurd die vaststelt wat men nodig heeft en die de nodige acties onderneemt. In NOW kunnen we zo een faling definiëren door gebruik te maken van een faling en een compensatie. In listing 5.11 definiëren we een faling rond de activiteit die de geluidsmuren plaatst (lijn 4). Op lijn 5 definiëren we dat er een alternatief moet worden uitgevoerd wanneer een service, in dit geval `WallWorkersService`, gedisconnecteerd is. Het alternatief dat wordt uitgevoerd is gedefinieerd op lijn 1. De activiteit `checkNeeds` stuurt een extra arbeider ter plaatse die kijkt of er iets nodig is zodat men toch kan verder gaan met het plaatsen van de geluidsmuren.

Wanneer we nu gebruik maken van een, aspect moeten we geen faling toevoegen aan de workflow. We definiëren in listing 5.12 eerst een aspect (lijn 3), waarna we een pointcut toevoegen (lijn 4), namelijk `PointcutDisconnected('WallWorkersService)`. Dit pointcut selecteert het joinpoint waarbij de `WallWorkersService` is gedisconnecteerd. Hierna bepalen we in het advice (lijn 4) dat er een alternatieve workflow moet worden gestart, namelijk deze op lijn 1.

```
1 def alt := SupplyService.checkNeeds();
2
3 def aspect := Aspect();
4 aspect.setPointcut(PointcutDisconnected('WallWorkersService));
5 aspect.setAdviceAfter( {|env| alt.start(WFenv) });
```

Listing 5.12: Voorbeeld faling gedefinieerd in aspect.

Het voordeel hieraan is dat we de faling niet moet toevoegen aan de workflow zelf, maar deze onafhankelijk kunnen definiëren. Zo is het mogelijk om alle falingen tussen workflows apart te definiëren zonder dat we hiervoor aanpassingen moeten maken aan de workflows zelf.

Omdat beide workflows worden uitgevoerd in dezelfde omgeving, is het mogelijk dat falingen van één workflow een effect hebben op een andere workflow. Wanneer bijvoorbeeld de service die verantwoordelijk is voor de signalisatie gedisconnecteerd is, mogen de wegenwerken niet starten omdat er geen zekerheid is dat de nodige signalisatie is aangebracht, en de veiligheid voor de arbeiders niet kan gegarandeerd worden. In de workflow van de wegenarbeiders is het

```
1 def aspect := Aspect();
2 aspect.setPointcut(PointcutDisconnected(`SignalService));
3 aspect.setAdviceAfter( {|env|
4                       def syncAspect := Aspect();
5                       syncAspect.setPointcut(PointcutMethod("bringMaterial");
6                       syncAspect.setAdviceAfter(
7                                   PointcutMethod("putSignalizationOnRoad"));
8                         });
```

Listing 5.13: Voorbeeld faling waarna synchronisatiepunt wordt toegevoegd.

cruciaal dat de signalisatie is aangebracht nadat het nodige materiaal ter plaatse is en voor de werken effectief beginnen. Wanneer er een disconnectie is, wordt er op dit punt een synchronisatiepunt toegevoegd, dat pas kan worden opgeheven wanneer de signalisatie geplaatst is.

In listing 5.13 zien we de implementatie van dit scenario. We definiëren eerst een aspect (lijn 1), waarna we zijn pointcut bepalen (lijn 2), namelijk wanneer er een disconnectie plaatsvindt van de `SignalService`. Na een disconnectie van deze service wordt het advice uitgevoerd op lijn 3. In dit advice maken we een nieuw aspect aan (lijn 4) dat verantwoordelijk is voor het toevoegen van het synchronisatiepunt. Het pointcut op lijn 5 bepaalt waar dit synchronisatiepunt wordt toegevoegd, namelijk na de activiteit `bringMaterial`. Tenslotte wordt nog op lijn 6 bepaald wanneer het synchronisatiepunt mag worden opgeheven, namelijk wanneer de activiteit `putSignalizationOnRoad` is afgerond. Dit aspect verzekert ons dat de werken pas starten wanneer alle signalisatie is aangebracht. Wanneer de `SignalService` disconnecteert nadat de signalisatie is aangebracht, zal dit advice geen effect meer hebben omdat de workflow van de wegenarbeiders al mocht starten met de werken. In NOW is het niet mogelijk om te bepalen wat een workflow als compensatie moet uitvoeren wanneer een service die een taak van een andere workflow uitvoert is gedisconnecteerd.

5.2.2 Uitvoeren van ondersteunende activiteit

Bij het plaatsen van geluidsmuren moet er steeds een voortdurende aanwezigheid zijn van het nodige materiaal. Dit moet worden geleverd zolang er geluidsmuren worden geplaatst. We kunnen dus stellen dat zolang de activiteit `installSoundWalls` wordt uitgevoerd een andere activiteit, namelijk `provideWalls`, ook moet worden uitgevoerd. In NOW kunnen we niet rechtstreeks zeggen dat een activiteit evenlang moet worden uitgevoerd als een andere. We kunnen deze enkel in parallel laten uitvoeren zoals in de volgende listing, waarbij beide activiteiten op lijn 1 in parallel worden uitgevoerd. Het voltooien van beide activiteiten samen ligt dan volledig in handen van de uitvoerders.

```
1 def seqWallTeam := ParallelSplit( WallWorkersService.installSoundWall(),
2                                   TransportService.provideWalls());
```

Daarom hebben we het *during* advice geïntroduceerd. Dit advice voert een activiteit uit zolang een joinpoint wordt uitgevoerd. Een definitie hiervan bevindt zich in listing 5.14. We definiëren eerst op lijn 1 de activiteit die moet worden uitgevoerd zolang de geluidsmuren worden geïnstalleerd. Vervolgens definiëren we op lijn 3 het aspect, waarna op lijn 4 het pointcut wordt toegevoegd die de activiteit `installSoundWall` selecteert. Tenslotte bepalen we op lijn 5 het during advice, waarmee we de activiteit op lijn 1 meegeven als parameter die moet worden uitgevoerd.

```
1 def provision := TransportService.provideWalls();
2
3 def aspect := Aspect();
4 aspect.setPointcut(PointcutMethod("installSoundWall"));
5 aspect.setAdviceDuring( provision );
```

Listing 5.14: Voorbeeld during advice.

5.2.3 Uitvoeren extra service

In sommige scenario's is het mogelijk dat de uitvoering van een taak pas mag worden gestart wanneer men zeker is dat een andere taak reeds begonnen is. Een voorbeeld hiervan is dat wanneer wegenwerkers begonnen zijn met de opkuis van de werken, men dan pas de vrachtwagens die het asfalt aanbrengen naar huis kan laten gaan. Vroeger dan dit tijdstip is men niet zeker dat het volledige wegdek wel klaar is en men geen asfalt meer nodig heeft. In NOW kunnen we dit enkel formuleren door deze activiteit toe te voegen, wat in listing 5.15 gebeurt op lijn 4. Dit wordt echter uitgevoerd voor de activiteit `cleanUp`, waardoor er nog steeds problemen kunnen opduiken en de vrachtwagens te vroeg vertrekken.

```
1 def seqRoadTeam := Sequence( RoadWorkersService.scrapeAsfalt(),
2                              RoadWorkersService.addProtectionLay(),
3                              RoadWorkersService.pourNewAsphalt(),
4                              TransportService.goHome(),
5                              RoadWorkersService.cleanUp());
```

Listing 5.15: Voorbeeld extra service toevoegen in NOW.

We kunnen gebruik maken van het *action* advice, wat een advice uitvoert zodra een activiteit gestart is. In listing 5.16 definiëren we zo een action advice. We starten met de definitie van een nieuw aspect (lijn 1), waarna we het pointcut bepalen (lijn 2), wat in dit geval de activiteit `cleanUp` selecteert. Daarna bepalen we op lijn 3 het action advice, die dan de activiteit `goHome` uitvoert.

```
1 def aspect := Aspect();
2 aspect.setPointcut(PointcutMethod("cleanUp"));
3 aspect.setAdviceAction({ |env| TransportService.goHome().start(WFenv);
4                         });
```

Listing 5.16: Voorbeeld action advice.

5.3 Besluit

Wanneer we kijken naar de vergelijking tussen NOW en de aspectgeoriënteerde oplossing, kunnen we besluiten dat deze enkele tekortkomingen van NOW oplost. Zo is het mogelijk om verschillende onafhankelijke workflows te laten samenwerken zonder dat we deze workflows moeten combineren in één workflow of de workflow zelf te wijzigen. Om een andere workflow te starten moeten we in NOW expliciet een trigger toevoegen. Wanneer we een aspect gebruiken, kunnen we een andere workflow starten zonder dat we hiervoor aanpassingen moeten maken aan de andere workflow. Wanneer we verschillende workflows moeten synchroniseren, moeten deze in NOW worden gecombineerd tot één grote workflow waarin we gebruik maken van een parallelle split en synchronize. Als we een aspect gebruiken, moeten we enkel bepalen waar er een synchronisatiepunt moet worden toegevoegd, en wanneer dit mag worden opgeheven.

Het wijzigen van een workflow wanneer deze reeds wordt uitgevoerd is niet mogelijk in NOW. Deze kunnen enkel worden gewijzigd bij het design van de workflow, maar hierna is dit niet meer mogelijk. Het aspect geeft ons de mogelijkheid om de structuur van workflows te wijzigen wanneer ze reeds worden uitgevoerd. Zo selecteren we via het pointcut welk patroon er moet worden gewijzigd, en het advice bepaalt welke aanpassing er hieraan moet gebeuren. Hiernaast geeft het aspect ook de mogelijkheid falingen tussen verschillende workflows af te handelen. Wanneer er een disconnectie van een service plaatsvindt, en deze bijgevolg zijn activiteit niet kan voltooien, kan dit effect hebben op het resultaat van een andere workflow. We kunnen hiervoor een compensatie definiëren, zoals we in het vorige voorbeeld een synchronisatiepunt hebben toegevoegd.

Hoofdstuk 6

Conclusie

6.1 Toekomstig werk en limitaties

Er zijn nog steeds een aantal topics waar in de toekomst verder onderzoek naar kan gedaan worden. Zo is het momenteel enkel mogelijk om basis patronen in een workflow te wijzigen, zoals bij een parallelle split of een sequence, maar we zouden dit nog kunnen uitbreiden naar andere patronen. Een voorbeeld hiervan is dat we in een workflow een gestructureerde loop willen toevoegen. Een ander patroon dat we nog kunnen wijzigen is het *multi-choice* patroon. Hier kunnen we dan bestaande keuzes wijzigen, toevoegen of verwijderen. Aangezien er al een pointcut voorzien is voor het selecteren van patronen als joinpoint, net zoals advices die aan het patroon iets toevoegen, verwijderen, of wijzigen, moeten we enkel het joinpoint van het patroon in NOW identificeren en de verschillende advices op het juiste tijdstip uitvoeren.

Een tweede mogelijke uitbreiding is dat we het aspect ook toepassen op patronen specifiek voor groepsorchestratie. We moeten eerst de joinpoints van deze patronen in de implementatie van NOW identificeren, maar kunnen opnieuw het pointcut `PointcutPattern` gebruiken om dit joinpoint te selecteren. Door de advices kunnen we dan deze patronen opnieuw wijzigen. Een mogelijkheid is dat we de voorwaarde van het barrier patroon wijzigen wanneer een andere workflow een bepaalde taak heeft uitgevoerd, bijvoorbeeld wanneer een groep met vertraging op een festival aankomt, kunnen de arbeiders nog niet beginnen met het klaarzetten van al het materiaal waardoor het optreden later zal beginnen en de festivalbezoekers nog meer tijd krijgen om hun favoriete liedjes te kiezen. Het advice zal hier dan de tijdslimiet van het barrier patroon wijzigen.

Om verschillende workflows met elkaar te laten samenwerken hebben we aspecten geïntroduceerd. Deze moeten echter nog steeds door de ontwerper van de workflows gedefinieerd worden. In sommige gevallen kunnen deze aspecten automatisch gedefinieerd worden, bijvoorbeeld wanneer er een exclusive choice wordt uitgevoerd. Wanneer de data, die de keuze bepaald welke branch er wordt uitgevoerd, door een andere workflow wordt bepaald en nog niet gekend is moet er een synchronisatiepunt worden toegevoegd. Dit zal pas worden opgeheven wanneer de data toegevoegd is door een andere workflow in de omgeving. Deze samenwerking tussen workflows

is veel voorkomend en zou in de toekomst geautomatiseerd kunnen worden door automatisch een aspect te definiëren wanneer dit geval zich voordoet.

6.2 Contributie

Doorheen deze thesis hebben we onderzoek gedaan naar de samenwerking tussen workflows in nomadische netwerken. Wanneer verschillende workflows worden uitgevoerd in dezelfde omgeving, bestaat de mogelijkheid dat er tussen deze een afhankelijkheid bestaat. Zo kan een workflow data nodig hebben die wordt bepaald door een taak van een andere workflow, of deze kan pas worden gestart wanneer een andere workflow een bepaald punt bereikt heeft. Wanneer taken in verschillende workflows afhankelijk zijn van elkaar, is het mogelijk dat één taak moet wachten op het resultaat van de andere taak. Deze workflows kunnen worden gesynchroniseerd met elkaar door het toevoegen van een synchronisatiepunt. Doordat deze workflows worden uitgevoerd in nomadische netwerken, moeten we ook steeds rekening houden met mogelijke disconnecties. Wanneer er een bepaalde service gedisconnecteerd is, weten we niet of een taak reeds is uitgevoerd, wat ook een effect kan hebben op andere workflows die afhankelijk zijn van dit resultaat.

We kunnen workflows definiëren in NOW, een nomadische workflow taal die ons toelaat om workflows te implementeren in nomadische netwerken. We hebben de samenwerking tussen workflows verwezenlijkt door gebruik te maken van een aspectgeoriënteerde oplossing. Hiervoor hebben we NOW uitgebreid zodat we aspecten kunnen definiëren die worden toegepast op alle workflows in de omgeving. Dit laat ons toe om de samenwerking tussen workflows te definiëren in een aspect zonder dat we hiervoor wijzigingen moeten aanbrengen aan de workflow zelf. We hebben meerdere pointcuts voorzien zodat er verschillende mogelijkheden zijn om activiteiten en data aan te duiden in workflows. Er wordt ook een pointcut voorzien die het toelaat om een disconnectie aan te duiden.

Naast het aanduiden van plaatsen in workflows moeten we ook kunnen bepalen wat er op deze plaatsen moet gebeuren. Dit wordt bepaald in verschillende advices. Het advice type bepaalt waar er ten opzichte van de plaats in de workflow iets moet worden uitgevoerd (ervoor, erna, ...) terwijl de advice body bepaalt wat er moet gebeuren. Hierin onderscheiden we drie verschillende soorten. Zo is het mogelijk om extra code uit te voeren, wat ondermeer kan gebruikt worden om een nieuwe workflow te starten. De tweede body is verantwoordelijk voor het toevoegen van een synchronisatiepunt in een workflow. De derde body geeft ons de mogelijkheid om de structuur van een workflow te wijzigen.

We hebben onze oplossing toegepast op verschillende scenario's waarbij we de vergelijking konden maken met NOW. We voorzagen twee verschillende scenario's: één scenario implementeerde de samenwerking tussen hulpdiensten bij een ramp, terwijl het tweede scenario de samenwerking tussen verschillende teams bij wegenwerken implementeerde. Door de samenwerking tussen workflows te definiëren in een aspect, moeten we geen wijzigingen meer aanbrengen in

de workflows zelf. Dit is wel het geval wanneer we dit willen doen in NOW. Aspecten kunnen op elk moment worden toegepast op workflows, ook reeds wanneer de uitvoering van de workflow al gestart is. In NOW moeten we de volledige samenwerking tussen alle workflows gedefinieerd hebben voordat de workflows worden uitgevoerd. Wanneer men op voorhand geen weet heeft van de workflows die aanwezig zullen zijn, is het onmogelijk om deze op voorhand reeds te wijzigen. Door de samenwerking onafhankelijk te definiëren van de workflows wordt het definiëren van de samenwerking tussen workflows veel eenvoudiger.

Bibliografie

[Barkhuus and Polichar, 2011] Barkhuus, L. and Polichar, V. (2011). Empowerment through seamfulness: smart phones in everyday life. *Personal and Ubiquitous Computing*, 15(6):629–639.

[Beale, 2005] Beale, R. (2005). Supporting social interaction with smart phones. *Pervasive Computing, IEEE*, 4(2):35–41.

[Brichau and Haupt, 2005] Brichau, J. and Haupt, M. (2005). Survey of Aspect-oriented Languages and Execution Models. Technical report. AOSD-Europe-VUB-01.

[Charfi and Mezini, 2004] Charfi, A. and Mezini, M. (2004). Aspect-oriented web service composition with ao4bpel. In Zhang, L.-J. and Jeckle, M., editors, *Web Services*, volume 3250 of *Lecture Notes in Computer Science*, pages 168–182. Springer Berlin Heidelberg.

[Charfi and Mezini, 2007] Charfi, A. and Mezini, M. (2007). AO4BPEL: An aspect-oriented extension to BPEL. *World Wide Web*, 10(3):309–344.

[Clark, 1999] Clark, J. (1999). Xml path language (xpath). `http://www.w3.org/TR/xpath/`.

[Dijkstra, 1982] Dijkstra, E. W. (1982). EWD 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66.

[Eugster et al., 2003] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131.

[Georgakopoulos et al., 1995] Georgakopoulos, D., Hornick, M., and Sheth, A. (apr 1995). An overview of workflow management: From process modeling to workflow automation infrastructure. In *Distributed Parallel Databases*, pages 3(2): 119–153.

[Heinlein, 2002] Heinlein, C. (2002). Synchronization of concurrent workflows using interaction expressions and coordination protocols. In Meersman, R. and Tari, Z., editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 54–71. Springer Berlin Heidelberg.

[Hilsdale et al., 2001] Hilsdale, E., Hugunin, J., Kersten, M., Kiczales, G., and Palm, J. (2001). Aspect-oriented programming in Java with AspectJ. In *O'Reilly Conference on Enterprise Java*.

[Joncheere and Van Der Straeten, 2011] Joncheere, N. and Van Der Straeten, R. (2011). Uniform modularization of workflow concerns using unify. In *Proceedings of the 4th international conference on Software Language Engineering(SLE 2011)*, volume 6940 of *Lecture Notes in Computer Science*, pages 77–96, Braga, Portugal. Springer.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of ASPECTJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK. Springer-Verlag.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *ECOOP 1997 - Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg.

[Kiczales and Mezini, 2005] Kiczales, G. and Mezini, M. (2005). Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 49–58, New York, NY, USA. ACM.

[La Rosa et al., 2011a] La Rosa, M., ter Hofstede, A., Wohed, P., Reijers, H., Mendling, J., and van der Aalst, W. M. P. (2011a). Managing process model complexity via concrete syntax modifications. *Industrial Informatics, IEEE Transactions on*, 7(2):255–265.

[La Rosa et al., 2011b] La Rosa, M., Wohed, P., Mendling, J., ter Hofstede, A., Reijers, H., and van der Aalst, W. M. P. (2011b). Managing process model complexity via abstract syntax modifications. *Industrial Informatics, IEEE Transactions on*, 7(4):614–629.

[Mascolo et al., 2002] Mascolo, C., Capra, L., and Emmerich, W. (2002). Advanced lectures on networking. pages 20–58, New York, NY, USA. Springer-Verlag New York, Inc.

[Nick Russell and Hofstede, 2006] Nick Russell, W. M. v. d. A. and Hofstede, A. H. M. T. (2006). Exception handling patterns in process-aware information systems. Technical report, BPM Center Report BPM-06-04, BPM Center. URL $http://workflowpatterns.com/documentation/documents/BPM-06-04.pdf$.

[Ouyang et al., 2007] Ouyang, C., Verbeek, E., van der Aalst, W. M. P., Breutel, S., Dumas, M., and ter Hofstede, A. H. M. (2007). Formal semantics and analysis of control flow in ws-bpel. *Sci. Comput. Program.*, 67(2-3):162–198.

[Parnas, 1972] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

[Philips, 2013] Philips, E. (2013). *Workflow Abstractions for Orchestrating Services in Nomadic Networks*. PhD thesis, Vrije Universiteit Brussel.

[Philips et al., 2013] Philips, E., Straeten, R. V. D., and Jonckers, V. (2013). NOW: Orchestrating services in a nomadic network using a dedicated workflow language. *Science of Computer Programming*, 78(2):168 – 194. Coordination 2010.

[Philips et al., 2012] Philips, E., Vallejos, J., Van Der Straeten, R., and Jonckers, V. (2012). Group orchestration in a mobile environment. In Sirjani, M., editor, *Coordination Models and Languages*, volume 7274 of *Lecture Notes in Computer Science*, pages 181–195. Springer Berlin Heidelberg.

[Pisa, 2009] Pisa, F. (2009). Inversion of control. In *Beginning Java and Flex*, pages 89–129. Apress.

[Popovici et al., 2002] Popovici, A., Gross, T., and Alonso, G. (2002). Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, AOSD '02, pages 141–147, New York, NY, USA. ACM.

[Rao et al., 2007] Rao, R. R., Eisenberg, J., and Schmitt, T. (2007). *Improving Disaster Management: The Role of IT in Mitigation, Preparedness, Response, and Recovery*. The National Academies Press.

[Russell et al., 2004a] Russell, N., Hofstede, A. H. M. T., Edmond, D., and van der Aalst, W. M. P. (2004a). Workflow data patterns. Technical report, Queensland University of Technology, Brisbane, QLD, Australia. QUT Technical Report FITTR-2004-01, URL `http://www.workflowpatterns.com/documentation/documents/data_patterns%20BETA%20TR.pdf`.

[Russell et al., 2006] Russell, N., Hofstede, A. H. M. T., and Mulyar, N. (2006). Workflow controlflow patterns: A revised view. Technical report, BPM Center Report BPM-06-22.

[Russell et al., 2004b] Russell, N., Ter Hofstede, A. H., Edmond, D., and van der Aalst, W. M. (2004b). Workflow resource patterns. Technical report, BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, Netherlands. URL `http://www.workflowpatterns.com/documentation/documents/Resource%20Patterns%20BETA%20TR.pdf`.

[Sen et al., 2008] Sen, R., Roman, G.-C., and Gill, C. (2008). Cian: A workflow engine for manets. In Lea, D. and Zavattaro, G., editors, *Coordination Models and Languages*, volume 5052 of *Lecture Notes in Computer Science*, pages 280–295. Springer Berlin Heidelberg.

[Thomas et al., 2009] Thomas, L., Wilson, J., Roman, G.-C., and Gill, C. (2009). Achieving coordination through dynamic construction of open workflows. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 14:1–14:20, New York, NY, USA. Springer-Verlag New York, Inc.

[Tian et al., 2009] Tian, K., Cooper, K., Zhang, K., and Yu, H. (2009). A classification of aspect composition problems. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 101–109.

[Van Cutsem et al., 2007] Van Cutsem, T., Mostinckx, S., Boix, E. G., Dedecker, J., and De Meuter, W. (2007). Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, SCCC '07, pages 3–12, Washington, DC, USA. IEEE Computer Society.

[Van Cutsem et al., 2009] Van Cutsem, T., Mostinckx, S., and De Meuter, W. (2009). Linguistic symbiosis between event loop actors and threads. *Computer Languages, Systems & Structures*, 35(1):80 – 98. ESUG 2007 International Conference on Dynamic Languages (ESUG/ICDL 2007).

[van der Aalst and van Hee, 2002] van der Aalst, W. and van Hee, K. (2002). *Workflow management: models, methods, and systems*. MIT Press, Cambridge, MA, USA.

[van der Aalst and ter Hofstede, 2005] van der Aalst, W. M. P. and ter Hofstede, A. H. M. (2005). Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275.

[van der Aalst et al., 2012] van der Aalst, W. M. P., Ter Hofstede, A. H. M., and et al. (apr 2012). The workflow patterns initiative. *http://www.workflowpatterns.com*.

[van der Aalst et al., 2003] van der Aalst, W. M. P., Ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. (2003). Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51.

[van der Vlist, 2002] van der Vlist, E. (2002). *XML Schema*. O'Reilly Media, Inc.

[Workflow Management Coalition, 1999] Workflow Management Coalition (1999). *Workflow Management Coalition, Terminology & Glossary (Document No. WFMC-TC-1011)*. Workflow Management Coalition.