Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica en Toegepaste Informatica

Programmeertaalondersteuning voor foutafhandeling in mobiele gedistribueerde systemen

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Informatica

Door: Jo Bigaré
Promotor: Theo D'Hondt
Mei 2006

© Vrije Universiteit Brussel, all rights reserved.

Samenvatting

Doordat mobiele toestellen zoals GSM's en PDA's steeds in kracht en aantal toenemen en steeds goedkoper worden, groeit de vraag naar applicaties die kunnen omgaan met het mobiele karakter van de netwerken waarvan deze toestellen gebruik maken. Deze mobiele netwerken worden gekarakteriseerd door het gebruik van draadloze verbindingen. Zulke verbindingen hebben slechts een beperkt bereik, met als gevolg dat de kans op een onverwacht verbroken verbinding tussen twee communicerende mobiele toestellen sterk toeneemt in vergelijking met een traditioneel netwerk. Om met dit frequenter voorkomen van verbroken verbindingen om te kunnen gaan is een aangepast foutafhandelingsmechanisme nodig in de programmeertaal die men gebruikt om toepassingen te implementeren voor mobiele netwerken. Zo wil men in een mobiel netwerk bijvoorbeeld tijdelijke verbroken verbindingen kunnen tolereren en abstractie maken over een verbroken verbindingsfout. Er is dus onderzoek nodig naar toegewijde foutafhandelingsmechanismen in mobiele netwerken.
In deze scriptie worden vooreerst criteria vooropgesteld die de toepasbaarheid van een foutafhandelingmechanisme in een mobiel netwerk evalueert. Naast het toetsen van reeds bestaande foutafhandelingsmechanismen uit gedistribueerde programmeertalen aan deze criteria, stellen we verschillende taalconstructies voor om aan foutafhandeling te doen in mobieke netwerken. Deze taalconstructies werden ontworpen voor en geïmplementeerd in de programmeertaal AmbientTalk. AmbientTalk is een taal die speciaal ontworpen werd om applicaties te implementeren voor mobiele netwerken. De taalconstructies worden ook op basis van de voorheen aangehaalde criteria geëvalueerd. Hieruit zal blijken dat de taalconstructies geschikt zijn om aan foutafhandeling te doen in mobiele netwerken, maar dat zij tegelijk slechts de eerste stap zijn in de ontwikkeling van meer geavanceerde abstractiemechanismen.

Inhoudsopgave

1 Inleiding **6**
1.1 Doel 6
1.2 Methodologie 7
1.3 Contributies 7
1.4 Overzicht 8

2 Foutafhandelingsmechanismen in Gedistribueerde Programmeertalen **10**
2.1 Inleiding 10
2.2 Java RMI 11
2.2.1 Gedistribueerd programmeren in Java RMI 11
2.2.2 Exception handling 14
2.2.3 Karakterisatie 14
2.3 ProActive 15
2.3.1 Gedistribueerd programmeren in ProActive 15
2.3.2 Exception handling 16
2.3.3 Conclusie 18
2.4 Emerald 18
2.4.1 Gedistribueerd programmeren in Emerald 19
2.4.2 failure handling 20
2.4.3 Conclusie 21
2.5 Argus 21
2.5.1 Gedistribueerd programmeren in Argus 21
2.5.2 Failure handling 22
2.5.3 Conclusie 24
2.6 E 24
2.6.1 Gedistribueerd programmeren in E 24
2.6.2 Failure handling 28
2.6.3 Conclusie 30
2.7 Conclusie 30

3 Ambient-georienteerd programmeren **31**
3.1 Inleiding 31
3.2 Software-ontwikkelingsproblemen in mobiele netwerken 32
3.3 Ambient Oriented Programming 33
3.3.1 Klasseloze objectmodellen 33
3.3.2 Niet-blokkerende communicatie primitieven 34
3.3.3 Gereïficeerde communicatiesporen 34

3.3.4 Ambient kennissenbeheer 34
3.3.5 Time-based failure handling 35
3.3.6 Conclusie 35
3.4 AmbientTalk 35
3.4.1 Pico 35
3.4.2 Prototype-gebaseerd objectmodel 36
3.4.3 Tweeledig objectmodel: actieve vs. passieve objecten . . . 37
3.4.4 Passieve object laag 38
3.4.5 Actieve objectlaag 39
3.4.6 Mailboxen 40
3.4.7 Meta-Object Protocol 42
3.5 Conclusie 45

4 Evaluatie van foutafhandelingsmechanismen voor ambient-georiënteerd programmeren 48
4.1 Criteria voor ambient-georiënteerde foutafhandelingsmechanismen 49
4.2 Try-catch exception handling 50
4.2.1 Exceptions zijn niet langer uitzonderlijk 50
4.2.2 Abstractie over tijdelijke verbroken verbindingen 50
4.2.3 Niet blokkerende communicatie 50
4.2.4 Try-catch als taalconstructie 51
4.3 When-catch exception handling 51
4.3.1 When-catch in E 51
4.3.2 Except-clauses in Argus 52
4.4 Observer-based exception handling 53
4.4.1 ProActive 53
4.4.2 E 54
4.5 Fatale failure handling 54
4.5.1 Fatale failure handling in ProActive 54
4.5.2 Fatale failure handling in E 54
4.6 Unavailable handlers 55
4.7 Conclusie 55

5 Experimenten met taalconstructies voor foutenafhandeling in mobiele netwerken 57
5.1 Analyse van asynchrone berichtenversturing in AmbientTalk . . . 58
5.2 De `due` taal constructie 60
5.2.1 Syntax en semantiek 60
5.2.2 Het design 65
5.2.3 Implementatie 65
5.2.4 Niet-tijdsgebaseerde vervalcondities 72
5.2.5 Evaluatie 75
5.2.6 Conclusie 76
5.3 De `receive_due_otherwise` taal constructie 76
5.3.1 Syntax en semantiek 76
5.3.2 Het design 78
5.3.3 Implementatie 79
5.3.4 Evaluatie 83
5.3.5 Conclusie 83
5.4 `reply_before` taal constructie 83

5.4.1 Futures in AmbientTalk 84
5.4.2 Syntax en semantiek 84
5.4.3 Het design 87
5.4.4 Implementatie 87
5.4.5 Promise pipelining 90
5.4.6 Evaluatie 91
5.4.7 Conclusie 92
5.5 Conclusie 92

6 Conclusie **93**
6.1 Contributies 93
6.2 Methodologie 96
6.3 Beperkingen 97
6.4 Toekomstperspectieven 98
6.5 Conclusie 98

Lijst van figuren

2.1 Een representatie van een vat [14]. 25
2.2 Een overzicht van welke soorten references er mogelijk zijn in verschillende omstandigheden. Zoals uit de figuur duidelijk wordt bestaan er twee soorten promises die dan ofwel resolven naar een near of far reference. Bij een failure wordt de promise echter resolved naar een broken reference. [14] 27
2.3 Voorbeeld van pipelining [14] 27

3.1 Overzicht van de syntax van zowel Pico als Pic% [5]. 37
3.2 Hier kan men zien wat er gebeurd wanneer men geen expliciete delegatie gaat invoeren in een mixin. De vierkante boxen stellen objecten voor en de opwaartse pijlen kan men beschouwen als de delegatie link. Het eerste geval toont wat er kan gebeuren als een mixin geen expliciete delegatie bevat. Het grote probleem hier is het feit dat het standaard MOP geen aanroepen meer zal krijgen, wat de goede werking van bepaalde mixins kan schaden. 46
3.3 Een voorbeeld van een compatibele mixin compositie. 47

5.1 Levencyclus van een bericht in AmbientTalk. 59
5.2 Grafische voorstelling van het printer-voorbeeld. In de eerste fase wordt er een print bericht naar de printerActor gestuurd zodanig dat dit bericht in de outbox van de gebruikersActor terecht komt. Wanneer dan de verbinding tussen deze twee actoren verbreekt zal het print bericht in de outbox blijven zitten tot er eventueel een herstellen optreedt van de verbinding. Met de beschreven failure handling op het print bericht zal het na een timeout waarde verwijderd worden uit de outbox van de actor van oorsprong. 63

Lijst van tabellen

5.1 Een overzicht van welke MOP methoden de DueMixin en ExpiryCheckMixin overschrijven. 66
5.2 Een overzicht van welke MOP methoden de `receive_due_otherwise` taal mixin overschrijft of verfijnt ten opzichte van de twee eerder besproken `due` mixins. 78
5.3 Weergave van manier waarop berichten en handlers opgeslagen worden in de multimap. 80
5.4 Een overzicht van alle methoden die de replyBeforeMixin en futuresMixin implementeren. 87

Hoofdstuk 1

Inleiding

In een wereld van constante vooruitgang is de computer een heel belangrijke schakel geworden in het dagelijkse leven van de mensen. Hoewel de eerste computers heel groot en lomp waren zijn deze over de jaren steeds kleiner geworden in volume. De miniaturisatie van de computer heeft ertoe geleid dat de computer meer en meer mobiel is geworden. Dit mobiele gebruik van de computer heeft voor het ontstaan gezorgd van onder andere PDA's en mobiele telefoons. Het succes van gedistribueerde netwerken tussen stationaire computers heeft geleid tot de ontwikkeling van netwerken voor mobiele computers. Om deze mobiele netwerken mogelijk te maken werden technologieën zoals WiFi en Bluetooth ontwikkeld. Deze technologieën maken het mogelijk om draadloze verbindingen te maken met andere toestellen. Deze draadloze netwerken hebben andere kenmerken dan normale stationaire netwerken waardoor de resulterende gedistribueerde systemen zich fundamenteel anders zullen gaan gedragen. Deze gewijzigde hardware heeft dus ongetwijfeld een impact op de software-ontwikkeling voor zulke systemen. Voorlopig is er echter nog maar weinig onderzoek verricht naar programmeertechnologie om programmeren in een mobiele omgeving op hoog niveau gemakkelijker te maken.

1.1 Doel

Wanneer we te maken hebben met gedistribueerde systemen is het heel belangrijk goede failure handling te voorzien zodat verbroken verbindingen op een goede manier behandeld kunnen worden. Onder failure handling verstaan we in deze thesis het detecteren en behandelen van verbroken verbindingen tussen gedistribueerde computers verbonden via een netwerk. In stationaire netwerken worden verbroken verbindingen aanschouwd als erg uitzonderlijke gevallen omdat de computers in deze netwerken verbonden zijn met vaste kabels en de kans dat er iets met deze kabels gebeurd relatief klein is. Verbroken verbindingen kunnen echter ook het gevolg zijn van fouten in een van de computers die deel uitmaken van het netwerk, ook hier zijn deze fouten echter uitzonderlijk omdat vaste computers vandaag heel betrouwbaar zijn geworden op vlak van gegarandeerde werking.
Het probleem van failure handling wordt complexer wanneer we dit willen toepassen op mobiele netwerken in plaats van hun stationaire voorgangers.

Hoewel er in beide aan failure handling moet gedaan worden zal de frequentie waarin dit moet gebeuren sterk veranderen. Draadloze verbindingen komen tot stand wanneer meerdere computers dicht genoeg bij elkaar komen zodat er communicatie mogelijk wordt. Een bestaande verbinding kan echter verbreken wanneer de verbonden computers te ver uit elkaars buurt gaan. Ook de kleine mobiele toestellen die vandaag beschikbaar zijn bezitten nog niet dezelfde graad van betrouwbaardheid als vaste toestellen, vooral door het gebruik van batterijen zodat het uitvallen van toestellen ook kan leiden tot failures. Fouten zullen dus in mobiele netwerken in principe frequenter voorkomen.
Het gevolg van de gewijzigde frequentie van verbroken verbindingen op software-engineering niveau is dat het gebruik van failure handling technieken ontworpen voor stationaire netwerken zou leiden tot overmatige uitvoering van code die eigenlijk uitzonderlijk zou moeten uitgevoerd worden. Bovendien kan zulke foutafhandelingscode de broncode van de applicatie grondig vervuilen aangezien in mobiele netwerken potentieel elke communicatie een hoger risico loopt op falen. In deze scriptie zullen bestaande failure handling systemen onderzocht worden met het oog op gebruik in een mobiele omgeving. Aan de hand van dit onderzoek zullen er dan criteria opgesteld worden voor failure handling in mobiele netwerken. Het doel van deze scriptie is het voorstellen van taalconstructies om aan failure handling te doen in een mobiele omgeving en deze taalconstructies dan ook te testen via een experimentele implementatie tegenover de bovenvermelde criteria.

1.2 Methodologie

Het implementeren en testen van de taalconstructies die in deze scriptie zullen voorgesteld worden gebeurt in een zogenaamde *ambient-georiënteerde* programmeertaal (AmOP), AmbientTalk genaamd [3]. De experimenten met taalconstructies voor failure handling zullen bestaan uit reflectieve uitbreidingen van deze ambient-georiënteerde taal. Voorlopig kan men een ambient-georiënteerde taal bekijken als een gedistribueerde programmeertaal die aangepaste eigenschappen heeft om gemakkelijk om te gaan met de eigenschappen van een mobiele gedistribueerde omgeving. Het experimenteren zal bestaan uit het ontwerpen en implementeren van specifieke taalconstructies in plaats van een meer traditionele aanpak gebaseerd op software bibliotheken of zogenaamde middleware. Elk van de voorgestelde taalconstructies zal dan geëvalueerd worden tegenover de criteria voor failure handling die zullen gespecificeerd worden.

1.3 Contributies

In deze scriptie worden vooreerst de mogelijkheden tot partiële foutafhandeling van de voornaamste gedistribueerde programmeertalen bestudeerd in de context van mobiele netwerken. Elke taal wordt kort uitgelegd met code voorbeelden om vertrouwd te raken met haar syntax en semantiek. Hierna worden de failure handling constructies van deze talen in detail besproken. Ten slotte worden al deze taalconstructies voor failure handling aanwezig in deze talen geëvalueerd om te bepalen of deze al dan niet voldoende bruikbaar zijn om te gaan gebruiken

in de context van mobiele gedistribueerde omgevingen.
Naast de bespreking en evaluatie van reeds bestaande taalconstructies voor failure handling in hedendaagse talen zullen er drie nieuwe taalconstructies voor foutafhandeling voorgesteld worden. Deze taalconstructies werden ontworpen voor en geïmplementeerd in de AmOP taal AmbientTalk om aan partiële foutafhandeling te doen in de context van mobiele netwerken. Deze taalconstructies werden ontworpen als reflectieve uitbreidingen in AmbientTalk hetgeen mogelijk is omdat deze taal een heel uitbreidbaar MOP (Meta-Object Protocol) bezit.

1.4 Overzicht

In deze sectie zullen we een overzicht bieden van de verschillende hoofdstukken teneinde de lezer te helpen om de structuur van de tekst te doorgronden.
Het volgende hoofdstuk bestaat uit een bespreking van de voornaamste taalconstructies voor partiële foutafhandeling in bestaande talen of bibliotheken. Deze bespreking bestaat uit een algemeen overzicht van de belangrijkste eigenschappen van de taal zelf voordat de eigenlijke foutafhandeling zelf besproken wordt.
Na het bespreken van bestaande partiële foutafhandeling in de context van stationaire netwerken zullen de problemen besproken worden die voorkomen wanneer men wil programmeren in een mobiele omgeving.
In het derde hoofdstuk zullen we het AmOP paradigma beschrijven. Dit paradigma beschrijft de eigenschappen die een programmeertaal moet bezitten om op een robuuste manier te kunnen programmeren in de context van mobiele netwerken. De taal AmbientTalk is gebaseerd op dit AmOP paradigma en is dus heel geschikt om toepassingen voor mobiele toestellen in te programmeren. Hiernaast is AmbientTalk een zeer compacte en reflectieve taal waardoor experimenteren met taalconstructies in deze taal heel gemakkelijk wordt. Hoofdstuk 3 biedt dan ook een gedetailleerde bespreking van de taal AmbientTalk.
Na het bespreken van reeds bestaande partiële foutafhandeling in talen of bibliotheken en de omschrijving van het AmOP paradigma zullen we deze twee onderwerpen in context van elkaar gaan bekijken. De taalconstructies voor foutafhandeling in hoofdstuk 2 zullen geëvalueerd worden in de context van het in hoofdstuk 3 besproken AmOP paradigma en de daarbij horende criteria. Op basis van deze AmOP criteria zullen we zelf een reeks criteria opstellen die de bruikbaarheid van een taalconstructie voor foutafhandeling in de context van mobiele netwerken en het AmOP paradigma karakteriseren. Deze criteria zullen dan doorheen deze scriptie gebruikt worden om aan te tonen of beschouwde failure handling systemen geschikt zijn voor gebruik in de mobiele omgevingen. De beschouwing van de reeds bestaande partiële foutafhandeling in de context van mobiel gedistribueerd programmeren en de opsomming van de criteria voor dit soort taalconstructies vindt men terug in hoofdstuk 4.
Uit de bespreking uit hoofdstuk 4 zullen we kunnen besluiten dat er geen enkele taalconstructie vandaag beschikbaar is die volledig aan alle criteria voldoet die opgesomd zijn in hoofdstuk 4. Teneinde de lacunes van de bestaande taalconstructies weg te werken hebben we zelf taalconstructies ontworpen in AmbientTalk omdat zoals eerder al aangehaald het heel gemakkelijk is deze taal uit te breiden met nieuwe taalconstructies. Hoofdstuk 5 zal drie van deze

experimenten in detail beschrijven, zowel op vlak van syntax en semantiek als de implementatie zelf. Naast deze uitleg volgt er voor elke taalconstructie een evaluatie tegenover de criteria die al eerder werden opgesomd in hoofdstuk 4. De conclusies over de criteria en de experimenten vindt men terug in hoofdstuk 6. Dit houdt in dat de voor- en nadelen van elk experiment zullen besproken worden. Daarnaast zullen de tekortkomingen van de gedane experimenten worden aangehaald als elementen voor toekomstig onderzoek.

Hoofdstuk 2

Foutafhandelingsmechanismen in Gedistribueerde Programmeertalen

2.1 Inleiding

Een goed programma moet altijd rekening houden met alle mogelijke inputs zodat het altijd correct kan reageren. Men streeft er dus naar om de consistentie van programma's zo lang mogelijk te behouden door te anticiperen op alle mogelijke scenario's. Dit kan zeer moeilijk worden in een netwerkomgeving omdat verbindingen nu eenmaal op elk moment kunnen verbreken. Om op deze verbroken verbindingen te anticiperen worden er in de meeste programmeertalen concepten geïntroduceerd die de programmeur kan gebruiken om te reageren op een verbroken verbinding. Het detecteren van deze fouten is een zeer moeilijk onderdeel van het probleem dat we in deze scriptie zullen bespreken en daardoor zullen we in dit hoofdstuk verder ingaan over hoe deze detectie geïmplementeerd is in enkele talen. Een ander onderdeel van discussie zijn de verschillende soorten foutafhandeling die we moeten voorzien in een taal, we zullen verder in deze scriptie de termen foutafhandeling en failure handling als synoniemen beschouwen en ze ook op die manier gebruiken. Moeten we voor netwerk verbindingen een ander soort failure handling implementeren dan "normale"fouten zoals divide-by-zero fouten? Is het ook mogelijk om failure handling mechanismen te maken die fatale fouten kan herstellen zoals volledige crashes van een bepaalde node van een netwerk? In dit hoofdstuk zullen we verder ingaan op deze problemen.
Dit hoofdstuk zal een overzicht bieden van failure handling voorzieningen in een selectie van programmeertalen of specifieke bibliotheken van talen. Om de failure handling in een taal grondig te kunnen uitleggen wordt er eerst een korte introductie gegeven van de taal in kwestie. Hierna wordt een grondige evaluatie gemaakt van de failure handling features die de programmeertaal bevat waaronder voorbeelden en vergelijkingen met soortgelijke talen. De bedoeling van dit hoofdstuk is een duidelijk beeld te geven van de beschikbare voorzieningen voor failure handling in reeds bestaande, hedendaagse programmeertalen. Hier een

kort overzicht welke talen we zullen bespreken en waarom we ze gekozen hebben:

Java RMI [16] is een bibliotheek van de programmeertaal Java die gebaseerd is op synchrone exception handling in traditionele netwerken. De technieken in deze exception handling kunnen interessant zijn voor gebruik in failure handling in mobiele netwerken.

ProActive [15] maakt gebruik van een model met actieve objecten dat eveneens gebruikt wordt in de taal AmbientTalk die zal besproken worden in hoofdstuk 3. Failure handling in samenhang met actieve objecten is een interessant gegeven dat we verder in deze thesis nog nodig zullen hebben.

Emerald [12] [13] gebruikt eveneens actieve objecten en maakt het dus even interessant om te bespreken als ProActive.

Argus [10] [11] is een programmeertaal met een zeer uitgebreid failure handling systeem waardoor het zeer interessant is deze taal te gaan bestuderen om eventueel ideeën op te doen voor failure handling in een mobiele omgeving.

E [14] maakt het mogelijk applicaties te programmeren voor open peer-to-peer netwerken zodat deze programmeertaal een goede basis vormt voor failure handling mechanismen in mobiele netwerken omdat deze ook open en peer-to-peer zijn.

2.2 Java RMI

Java RMI [16] is een bibliotheek van de Java programmeertaal. Deze werd in de eerste plaats ontworpen om gedistribueerd programmeren in Java een stuk gemakkelijker te maken voor de programmeurs en dan meer specifiek voor gesloten netwerken of LANs. In dit type van netwerken gaan we er dus vanuit dat failures op vlak van verbindingen die verbreken eerder uitzonderlijk zijn. Om de failure handling goed te kunnen uitleggen zal er eerst een voorbeeld programma uitgelegd worden waarna we dieper ingaan op de failure handling die beschikbaar is in Java RMI.

2.2.1 Gedistribueerd programmeren in Java RMI

- De units of distribution in Java RMI zijn zoals in Java zelf *objecten* in een klasse-gebaseerde omgeving.
- Java RMI kan zoals applicaties in Java gebruik maken van *threads* om concurrent te programmeren.
- De communicatie tussen de units of distribution wordt verwezenlijkt door het gebruik van *messages*.

In Java RMI wordt het versturen van een bericht over het netwerk geïmplementeerd alsof men een "normaal" bericht zou versturen. Een andere belangrijke eigenschap van Java RMI is het feit dat de berichten synchroon zullen verstuurd worden en de verzender dus zal wachten tot het een antwoord krijgt van de ontvanger van het bericht. Versturen van bijvoorbeeld getallen in Java zullen bij het versturen over het netwerk met behulp van pass-by-value doorgegeven

worden terwijl objecten door middel van pass-by-copy, wanneer deze de `Serializable` interface implementeren. Het gebrek aan transparantie en de normale semantiek van een java RMI programma kan afgeleid worden uit de volgende voorbeeld code:

De interface:

```
public interface Howdy extends Remote
{
   String Greet (String msg) throws RemoteException;
}
```

Deze interface die een extentie is van de Remote klasse vormt een plaatsvervanger voor klassen die zich op een andere node in het netwerk bevinden. De interface zal ervoor zorgen dat de client een interface heeft voor de automatisch aangemaakt proxy van de echte object die het verbergt omdat het zich op een andere machine bevindt.

Opbouw van onze server:

```
class HowdyServer extends UnicastRemoteObject
      implements Howdy
{
   public HowdyServer () throws RemoteException {};

   // Implementatie van Greet

   public String Greet (String msg) throws RemoteException
   {
      return "Howdy Yurself";
   }

   public static void main (String [] args) throws Exception
   {
      HowdyServer server = new HowdyServer ();

      // Registreer (bind) deze service bij rmiregistry

      Naming.rebind (Howdy.class.getName (), server);
   }
}
```

Hieruit ziet men meteen dat de programmeur zijn applicatie nog altijd zal moeten modelleren naar een normaal client-server systeem, dit omdat we natuurlijk met de onderliggende Java code moeten rekening houden. De `HowdyServer` is een zogenaamd remote object, wat wil zeggen dat objecten op andere toestellen in het netwerk berichten naar dit object kunnen sturen, en moet daardoor de klasse `UniCastRemoteObject` uitbreiden. Het uitbreiden van deze klasse zorgt er op zijn beurt voor dat de programmeur zelf verantwoordelijk is voor de concurrency in zijn applicatie. Het client-server model is mogelijk in Java RMI doordat men servers kan gaan registreren in een centrale databank (name servers), namelijk `rmiregistry`.

De client ziet er zo uit:

```
class HowdyClient
{
   static void main (String [] args) throws Exception
   {
      String   hostname;

      // Om remote berichten te versturen moeten we
      // de hostname kennen, deze zullen we dus opzoeken

      if (args.length < 1)
         hostname = new String (
            (InetAddress.getLocalHost ()).getHostName ());
       else
         hostname = args [0];

      // Creatie van een remote object waarnaar
      // men berichten kan sturen

      Howdy howdy = HowdyFactory.getHowdy (hostname);
      try {
        System.out.println (howdy.Greet ("Howdy"));
      } catch(RemoteException e) {
        System.out.println ("An error occured in sending Howdy");
      }
   }
}

public class HowdyFactory
{
   public static Howdy getHowdy (String hostname) throws Exception
   {
      return (Howdy) Naming.lookup
                        ("rmi://" + hostname + "/" + Howdy.class.getName ());

   }
}
```

De client moet uit een centrale locatie de plaats van de server kunnen opvragen voordat deze client kan starten met berichten naar de server te sturen. Hieruit kan men dus afleiden dat er enkel communicatie in één richting wordt aangemaakt, met name van client naar server. Bij het succesvol opvragen van de server wordt er een proxy van het remote object aangemaakt, de client zal dan de hierboven gedefinieerde interface gebruiken om met deze proxy te communiceren.

2.2.2 Exception handling

We zullen nu dieper ingaan op de aanwezige failure handling mechanismen in Java RMI. Zoals men kan opmerken uit het code voorbeeld, maakt Java RMI ook gebruik van het exception handling systeem van Java om failures te kunnen behandelen door failures te modelleren via `RemoteException` exceptions. De generale superklasse voor de RMI exceptions is dus de `RemoteException` klasse met als meest belangrijke subklassen:

- *UnknownHostException* wordt teruggegeven wanneer bij het verbinden naar een andere machine deze niet gevonden wordt en het aanmaken van de verbinding dus mislukt.
- *(Un)MarshalException* komt voor bij objecten die voor bepaalde redenen niet kunnen verstuurd worden over het netwerk, dit kan bijvoorbeeld voorkomen wanneer dit object de `Serializable` interface niet heeft uitgebreid.
- *ConnectException* kan voorkomen wanneer een machine op een andere node in het netwerk weigert een connectie te aanvaarden om er een bericht over te verzenden naar een object dat zich op deze machine bevindt.
- *ExportException* wordt teruggegeven wanneer een poging om een remote object te exporteren faalt.

Typische failure handling in JavaRMI is van deze stijl:

```
try
   {
      result = remoObject.aMessage (arguments)
   }
   catch (RemoteException e)
   {
      // behandelen van remote exceptions als deze voorkomen
   }
```

Elke `RemoteException` die wordt teruggegeven uit de code in het eerste deel van deze try-catch zal behandeld worden door de failure handler in het `catch` deel. Informatie over specifieke subklassen van de `RemoteException` kan men opvragen uit de variabele `e` die zichtbaar is in het `catch` deel.

2.2.3 Karakterisatie

In dit deel zullen we de belangrijkste kenmerken van de exception/failure handling mechanismen in Java RMI nog even herhalen.
In Java RMI worden failures gemodelleerd als exceptions waardoor failures gelijk worden aan exceptions. De failure handling wordt verwezenlijkt door gebruik te maken van try-catch taalconstructies die we kennen vanuit Java zelf. Het gebruik van een try-catch systeem verplicht de programmeur om te kiezen voor synchrone communicatie omdat alle informatie uit het `try` deel bekend moet zijn om te kunnen oordelen of het `catch` deel al dan niet opgeroepen moet worden. Een try-catch taalconstructie laat echter wel toe blokken code te groeperen per failure handler.

Een failure in JavaRMI is een object dat het soort fout die het voorsteld in meer detail beschrijft. Doordat deze objecten aangemaakt worden door middel van klassen kunnen deze ook in hiërarchiën gemodelleerd worden en is het mogelijk gebruik te maken van inheritance relaties.
Wanneer er een fout optreedt in Java is het niet meer mogelijk om onmiddelijk de code die gefaald heeft opnieuw te proberen (retry). Het is ook niet mogelijk om aan Java te zeggen dat het bij een failure moet verderwerken aan de code vlak na de code die gefaald heeft (resume). Wanneer men dit gedrag wil programmeren moet men de oproepen manueel doen. Doordat een zogenaamde retry/resume niet aanwezig is in Java en doordat failures als exceptions worden gezien is het onmogelijk om in Java een onderscheid te maken tussen mogelijk tijdelijke fouten en meer fatale fouten.

2.3 ProActive

ProActive [15] is een bibliotheek van Java die in het leven is geroepen om het programmeren van grids gemakkelijker te maken. Omdat ProActive ook bepaalde eigenschappen bezit in het kader van mobiliteit en veiligheid, kunnen we in deze taal wel is interessante ideeën opdoen voor ons eigen failure handling systeem. Voor we echter zullen overgaan tot het bespreken van het failure handling systeem, gaan we eerst de basis eigenschappen en werking van deze bibliotheek weergeven.

2.3.1 Gedistribueerd programmeren in ProActive

Omdat ProActive gebruik maakt van het actor model om zijn objecten te modelleren, vinden we twee soorten objecten terug in de taal:

- **Actieve objecten** [8]: deze objecten hebben hun eigen message queue en hebben ook een eigen thread die alle messages in de queue één voor een zal uitvoeren.
- **Passieve objecten**: objecten zonder speciale kenmerken, te vergelijken met gewone Java objecten.

In hoofdzaak heeft ProActive de volgende eigenschappen:

- Een applicatie is verdeeld in subsystemen. Er kan maar één actief object zijn per subsysteem en alleen één subsysteem per actief object. Men heeft dus per subsysteem 1 actief object samen met een arbitrair aantal passieve objecten, mogelijk ook geen.
- Men kan geen passieve objecten delen tussen twee subsystemen.

Het versturen van berichten tussen actieve objecten gebeurt asynchroon. Dit betekent dat bij het versturen van een bericht over het netwerk de verzender niet zal wachten op een antwoord van de ontvanger zodanig dat onmiddelijk na het versturen van een asynchroon bericht de verzender gewoon verder zal doen met de volgende code. Wanneer men passieve objecten meegeeft als parameter aan deze asynchrone berichten, zullen deze doorgegeven worden door pass-by-deep-copy. Dit betekent dat alle ouders van dit object gekopiëerd en meegegeven

zullen worden over het netwerk. Actieve objecten houden referenties bij van andere actieve objecten. Omdat berichten tussen actieve objecten asychroon zijn, geven deze geen pure return waardes terug. Het versturen van een bericht geeft in ProActive in sommige gevallen een future terug die kan gebruikt worden om de waarde van het bericht later te kunnen gebruiken. De uitleg van futures in ProActive doet hier niet ter zake omdat we teveel in detail moeten treden om het gebruik ervan in ProActive uit te leggen, daardoor zullen we enkel de grote lijnen van het gebruik ervan aanhalen. Wanneer men wil dat een future resolved gaat de huidige computatie geblockt worden (door de functie `waitForReply` op te roepen) tot het andere actieve objecten het bericht, waarvan de future de voorlopige return waarde was, heeft afgehandeld (dit is zo wanneer deze future de enige future is die op dat moment aanwezig is in het actieve object, dit is heel belangrijk omdat de functie `waitForReply` wacht tot eender welke future, van berichten die verzonden werden uit dit actief object, resolved wordt). Om een duidelijker beeld te scheppen van hoe een ProActive applicatie eruit ziet, volgt hier een code voorbeeld:

Instantiatie van een actief object in ProActive wordt als volgt geïmplementeerd:

```
TinyHello tiny = (TinyHello) ProActive.newActive(
                TinyHello.class.getName(), // the class to deploy
                null // the arguments to pass to the constructor, here none
            );
```

Een asynchrone message wordt als volgt geprogrammeerd:

```
StringMutableWrapper received = tiny.sayHello();
```

2.3.2 Exception handling

Nu we de grote lijnen van de werking van ProActive besproken hebben zullen we wat dieper ingaan op de failure handling die beschikbaar is in deze bibliotheek. Zoals je kan verwachten maakt deze bibliotheek gebruik van het exception handling systeem dat al aanwezig is in Java. ProActive maakt echter een onderscheid tussen twee soorten exceptions:

- Functionele exceptions zijn fouten die het gevolg zijn van applicatiedomein-specifieke fouten.
- Niet-functionele exceptions zijn het gevolg van netwerk fouten zoals het verbreken van verbindingen.

We zullen nu verder in detail gaan hoe men beide soorten exceptions kan gaan opvangen en behandelen.

Functionele exceptions

Als men het over functionele exceptions heeft moet men dus vooral denken aan exceptions die het gevolg zijn van applicatiedomein-specifieke fouten. Het gaat hem dus vooral over gewone exceptions die je ook terugvindt in normale Java code. Door deze gelijkenis is het ook niet verwonderlijk dat deze exceptions opgevangen en behandeld kunnen worden door een try-catch systeem. Het is

echter niet mogelijk het try-catch systeem in Java compleet over te nemen in ProActive, aangezien actieve objecten asynchrone berichten versturen en men dus geen idee heeft of het bericht nu al dan niet goed ontvangen en behandeld werd. Om dit probleem op te lossen zijn er enkele natives beschikbaar in de bibliotheek om toch met het try-catch systeem uit Java te kunnen werken:

```
ProActive.tryWithCatch(SomeException.class);
try {
  Result r = someAO.someMethodCall(); // Asynchronous method
  // call that can throw an exception
  doSomethingWith(r);
  ProActive.endTryWithCatch();
} catch (SomeException se) {
  doSomethingWithMyException(se);
} finally {
  ProActive.removeTryWithCatch();
}
```

Wanneer je dus exception handling wil op asynchrone berichten tussen actieve objecten zullen alle try-catch blocks moeten verrijkt worden met de natives uit het voorbeeld. Het valt ook op te merken dat de plaatsing van deze natives zo uniform is dat er een conversie programma beschikbaar is zodat men alle try-catch blocks automatisch kan omzetten. Hier ziet men dan ook het nadeel van bibliotheken in vergelijking met specifiek gemaakte talen met een eigen syntax voor exception handling.

Niet-functionele exceptions

De niet functionele exceptions omvatten vooral de netwerk failures zoals verbroken verbindinen. We zullen eerst een stukje voorbeeld code laten zien om een idee te krijgen hoe men dit soort failures aanpakt.

```
ProActive.addNFEListenerOnAO(myAO, new NFEListener() {
    public boolean handleNFE(NonFunctionalException nfe) {
        // Do something with the exception...
        // Return true if we were able to handle it
        return true;
    }
});
```

Men kan dus zogenaamde listeners definiëren op de zogenaamde units of distribution in ProActive. Deze listeners zullen ervoor zorgen dat de juiste handlers opgeroepen worden wanneer er een zogenaamde NFE opgemerkt wordt binnen een onderdeel van de distributie waarop de listener is gedefinieerd. Een overzicht van alle onderdelen waarop men listeners kan definiëren vindt men hier:

- Actief Object
- Proxy (ofwel de client side van een actief object)
- Java Virtual Machine

- Group bestaat uit collectie van reïficeerbare objecten in ProActive. Het is dus mogelijk objecten te groeperen in ProActive en op deze groep dan specifieke handlers te definiëren.

Deze non-functionele exceptions zijn een hele goede manier om met netwerk failures om te gaan, maar de nadelen van een exception handling systeem zijn nog altijd aanwezig zoals eerder al besproken bij de Java RMI bibliotheek. Exceptions zijn er om uitzonderlijke gevallen aan te pakken terwijl wij eigenlijk een systeem willen waar netwerk failures als normaal worden beschouwd.

Fatale failure handling

Met fatale failure handling in ProActive bedoelen we het omgaan met volledige systeemcrashes van één of meer nodes in het netwerk en hoe het hele netwerk van nodes hiermee omgaat. ProActive heeft twee features die men kan gebruiken om met fatale failures om te gaan.

- **Communication Induced Checkpointing (CIC)** : elke tijdsinterval t seconden wordt een backup genomen van alle objecten in het systeem zodat na een failure deze kunnen "restored" worden naar een vorige backup. Dit systeem heeft als nadeel dat wanneer er één node in het netwerk crasht alle nodes in het netwerk een rollback moeten doen naar het laatste checkpoint en van daaruit terug verder werken.
- **Pessimistic message logging (PML)**: er wordt een backup gemaakt van alle messages die verstuurd worden in het systeem, zodat men na een failure het volledige systeem terug kan opbouwen door alle messages die al verstuurd worden terug uit te voeren. Dit systeem heeft als voordeel dat enkel het gecrashte systeem een rollback moet doen naar zijn laatste backup waarna zijn huidige status van daaruit opgebouwd wordt.

Voor beide van de bovenvermelde systemen is dus een volledige connectiviteit nodig evenals een stabiel netwerk zonder veel verbroken verbindingen.

2.3.3 Conclusie

Hoewel ProActive zeer veel tools bevat om aan failure handling te doen, is het helemaal niet geschikt om te gebruiken in een mobiel gedistribueerde omgeving. Een pluspunt is echter wel het gebruik van de niet-functionele exceptions die worden behandeld door het gebruik van listeners. Een klein nadeel is dat de granulariteit van de eenheden waarop de listeners gedefinieerd kunnen worden te groot is, men kan namelijk geen listener definiëren op één verzonden bericht.

2.4 Emerald

Emerald [12] [13] werd ontworpen om gedistribueerd programmeren gemakkelijker te maken door zeer goede taalondersteuning te bieden. Het maakt gebruik van een prototype-gebaseerd objectmodel, wat vooral betekent dat men geen gebruik maakt van klassen om objecten te beschrijven en te instantiëren zoals onder andere in Java, maar van bepaalde objecten die als prototype fungeren voor andere objecten. In dit hoofdstuk zullen we eerst een globale omschrijving geven van Emerald, waarna we de failure handling zullen bespreken.

2.4.1 Gedistribueerd programmeren in Emerald

Zoals al eerder vermeld, maakt Emerald gebruik van een prototype-gebaseerd object model [4]. Een object in Emerald bevat de volgende zaken:

- Een naam.
- Een representatie, wat hoofdzakelijk bestaat uit referenties naar andere objecten.
- Een verzameling van operaties die gedefinieerd zijn op het object.
- Een optioneel proces, dat naast de normale operaties van het object zal draaien nadat het object geinitialiseerd is. Objecten met een proces zullen we actief noemen, objecten zonder proces noemen we passief.

Het berichtensysteem van Emerald is ook verschillend van de talen die we tot nu toe besproken hebben. In Emerald is er eigenlijk geen notie van plaats. Het oproepen van operaties op een object is volledig plaats onafhankelijk, men hoeft dus niet noodzakelijk te weten waar het object zich bevindt. Het enige wat men wel nodig heeft om dit te verwezenlijken is een referentie naar het object waarnaar men een bericht wil versturen. Hierdoor maakt Emerald gebruik van call-by-object-reference en worden dus alle objecten die meegegeven worden als argumenten, doorgegeven als referentie. Om performantie redenen zijn er ook twee alternatieve manieren voor parameter passing:

- Call-by-move: hierdoor zal het argument samen mee migreren naar de bestemming en zich daar gaan nestelen.
- Call-by-visit: zoals de naam al aangeeft zal het argument samen mee migreren naar de bestemming maar zal onmiddellijk terugkomen nadat de opgeroepen operatie voltooid is.

De berichten worden steeds synchroon verwerkt, ook in het geval van een remote message send en geven dus ook na elke voltooide operatie een return waarde terug.

```
const myDirectory : Directory == object oneEntryDirectory
        export Add, Lookup, Delete
        monitor
                var name : String <-  nil
                var An : Any <-  nil
                operation Add[n : String, o : Any]
                        name <- n
                        An  <- o
                end Add
                function Lookup[n : String] ! [o : Any]
                        if n = name then
                                o <-  An
                        else
                                o <-  nil
                        end if
                end Lookup
```

```
                operation Delete[n : String]
                        if n = name then
                                name <-  nil
                                An  <- nil
                        end if
                end Delete
        end monitor
end oneEntryDirectory
```

Dit voorbeeld stelt een simpele directory implementatie voor. Het bouwt het object `myDirectory` ex-nihilo op wat betekent dat het zijn gedrag zelf beschrijft zonder gebruik te maken van klassen. Het heeft drie methoden `Add`, `Lookup` en `Delete`. Deze methoden en de instance variabelen worden gedefinieerd in een monitor om race conditions te voorkomen door mogelijk concurrente oproepen van methoden op dit object.

2.4.2 failure handling

In Emerald onderscheiden we twee grote verzamelingen van failures die kunnen voorkomen. Hier volgt een grondige bespreking van beiden.

<u>Unavailability</u>

We zeggen dat objecten unavailable zijn als geen enkele locatie een connectie kan maken met dit object en het dus mogelijk gecrasht is. Emerald voorziet handlers, die men kan definiëren op objecten, om deze failures te behandelen. Een code voorbeeld van zo een handler vind je hier.

```
when myDirectory unavailable
        declarationsAndStatements
end unavailable
```

<u>Failure</u>

Failures in Emerald zijn errors afkomstig van applicatiecode, bijvoorbeeld een deling door nul. Een voorbeeld van hoe men het gebruikt vind men hier.

```
solution <- someNumber / otherNumber
on failure
      declarationsAndStatements
end failure
```

Wanneer in het bovenliggend voorbeeld `otherNumber` nul is dan zal de failure handler `declarationsAndStatements` opgeroepen worden. Dit failure systeem heeft veel gelijkenissen met het try-catch systeem dat we terugvinden in gebruikelijke talen zoals Java. Wanneer een bepaalde code block een failure teruggeeft zal de handler die bij dat code block hoort worden aangeroepen. Wanneer een failure niet onmiddellijk door een handler wordt opgevangen, zal deze propageren in de code totdat er een handler gevonden wordt. Dit soort van failure handling is helemaal niet geprefereerd omdat eigenlijk alle catch blocks alle mogelijke failures zullen catchen zodat men helemaal geen idee heeft welke fout er nu is opgetreden. Emerald heeft met andere woorden geen exception objecten die op basis van hun type kunnen opgevangen worden.

2.4.3 Conclusie

Hoewel het objectmodel dat in Emerald wordt gehanteerd heel bruikbaar lijkt in een gedistribueerde omgeving, stelt de failure handling teleur. Het concept van de unavaileble handlers is echter wel een stap in de goede richting doordat men handlers kan uitvoeren wanneer een bepaald onderdeel van de distributie zogenaamd unavailable wordt. Een nadeel aan Emerald is het feit dat wegvallende verbindingen worden gemodelleerd als exceptions zoals we ook al eerder zagen bij de Java gebaseerde talen. Een ander nadeel vinden we terug in de semantiek van de failure handling, omdat deze gebaseerd is op een catch-all semantiek is men verplicht failure handling per message send te definiëren.

2.5 Argus

Zoals alle talen die we in dit hoofdstuk bespreken is Argus [10] [11] ontwikkeld om programmeurs van een aantal moeilijkheden te ontlasten in het schrijven van gedistribueerde programma's. Argus specialiseert zich in het consistent houden van data in een gedistribueerde omgeving. Dit merkt men doordat de elementen in processen geïmplementeerd worden als transacties. Transacties worden vooral in databases gebruikt om ervoor te zorgen dat datamanipulatie ofwel geheel goed verloopt of als er iets fout loopt geheel geannuleerd wordt. We beginnen met het bespreken van Argus in het algemeen, waarna we iets dieper zullen ingaan op de failure handling.

2.5.1 Gedistribueerd programmeren in Argus

Het object model van Argus bestaat uit gewone objecten en een speciaal soort van objecten, guardians genaamd. Guardians bevinden zich in nodes, die verschillende computers voorstellen, en elke guardian kan verschillende concurrente processen beheren. Actions zijn speciale processen die geïmplementeerd zijn als transacties. Actions groeperen operaties die ofwel volledig uitgevoerd kunnen worden, ofwel totaal niet.
Methoden in Argus worden handlers genoemd en deze zorgen dat er acties op guardians gedefinieerd kunnen worden. Handlers zijn de enige manier om met guardians te communiceren en zijn dan ook de enige manier om toegang te krijgen tot de data binnen een guardian. Een voorbeeld van zo een handler vind men hier.

```
close = handler (a: account-number) signals (no-such-acct, positive-balance)
        b: bucket := ht[hash(a.num)]
        for i: int in bucket$indexes(b) do
                if b[i].num -= a then continue end
                if b[i].acct.bal > 0 then signal positive-balance end
                b[i] := bucket$top(b) % store topmost element
                                      % in place of closed account
                bucket$remh(b) % discard topmost element
                return
                end
        signal no-such-acct
        end close
```

De handler uit het bovenstaande voorbeeld wordt gebruikt om een account bij een bank te sluiten, het geeft een fout wanneer het nummer van de accout niet bestaat (`no-such-acct`)en geeft ook een waaschuwing wannneer de nog geld op de account staat bij het sluiten ervan (`positive-balance`).
Het versturen van berichten van de ene guardian naar de andere is locatie onafhankelijk. Hiermee wordt bedoelt dat de zender de locatie niet hoeft te weten van de ontvanger om een bericht te kunnen versturen, dit wordt helemaal geregeld door het interne systeem van Argus. Argumenten die geen guardian of handler zijn worden doorgegeven volgens het pass-by-value principe, terwijl guardians en handler volgens het pass-by-reference principe worden doorgegeven.
Guardians kunnen andere guardians dynamisch aanmaken. Bij het aanmaken kan ook beslist worden waar de nieuwe guardian zich moet gaan nestelen, dit kan gebeuren in elk node van het netwerk. Deze feature kan natuurlijk gebruikt worden om betere performantie te verkrijgen door guardians op strategisch goede plaatsen te ontplooien.
In latere versies van Argus wordt er ook gebruik gemaakt van promises en zogenaamde streams. Deze streams zorgen ervoor dat in een omgeving die asynchrone berichtgeving gebruikt men een zekerheid kan invoegen in welke volgorde de berichten behandeld worden door de ontvanger. Streams zorgen er ook voor dat men een bepaalde zekerheid heeft dat berichten goed aankomen en behandeld worden door de ontvanger. Promises zijn objecten die een plaatsvervanger zijn van de return waarde van een handler. Een promise kan zich in twee toestanden bevinden:

- **Blocked**: een promise is blocked wanneer de handler call nog niet is behandeld en men de return waarde nog niet kent. Hierdoor zal elk process blocken wanneer het de waarde van de promise probeert te gebruiken. Het wachten tot een bepaalde promise resolved wordt bekomt men in Argus door gebruik te maken van de `claim` operator, die de computatie gaat blocken totdat de promise resolved wordt.

- **Ready**: wanneer de handler is afgehandeld door de ontvanger zal de promise de waarde dragen van de return waarde. Een promise kan nooit meer een andere waarde aannemen als het eens in zijn bestaan ready is geworden.

- Naast de vorige twee mogelijkheden kan een promise ook resulteren in een exception zodat dit dan ook afgehandeld kan worden door het try-catch systeem van Argus dat hieronder zal besproken worden.

2.5.2 Failure handling

Zoals in de inleiding al aangehaald werd focust Argus op het consistent houden van data in een gedistribueerde omgeving. Door het gebruik van atomaire uitvoering van actions zal een bepaald stuk code helemaal moeten uitgevoerd worden ofwel wordt het hele code block als failed beschouwd en wordt er een signal (exception) teruggegeven. Deze signal kan dan opgevangen worden door try-catch blocks in Argus waar men hier een voorbeeld van ziet.

```
% for a transfer it does the following
% find out accounts and amounts from user
% and store in local variables to, from and amt
enter topaction
        t: branch := get-branch(to)
        f: branch := get-branch(from)
        coenter
                action f.withdraw(from, amt)
                action t.deposit(to, amt)
                end except others: abort exit problem end
                % all exceptions cause abort of topaction
        end % topaction
                except when problem: % tell user that transfer failed
                        end % except
% tell user that transfer succeeded
```

Het bovenstaand voorbeeld geeft de code weer voor het afhandelen van transfers tussen twee accounts in bijvoorbeeld een bank voorbeeld. De aanroep van `enter topaction` zorgt ervoor dat deze code in een apart process zal uitgevoerd worden. Merk op dat deze `topaction` effectief zal uitgevoerd worden als alles erbinnen correct en zonder `abort` uitgevoerd werd. De code na `coenter` zal parallel uitgevoerd worden maar zal er ook voor zorgen dat wanneer er een exception opgevangen wordt uit een van de twee `action`'s ook de `topaction` zal falen. Bij een failure van de `topaction` zal de oproep `except when problem` ervoor zorgen dat de gebruiker kan ingelicht worden van het falen van de hele transfer.
Er bestaan in Argus twee standaard exceptions die worden teruggegeven bij een netwerkfout, we zullen deze twee hier kort beschrijven:

unavailable wil zeggen dat de fout van een tijdelijke aard is maar dat het niet aangeraden is onmiddelijk opnieuw dezelfde actie te proberen. Er bestaat dus een mogelijkheid dat de verbinding kan herstelt worden.

failure wil zeggen dat de fout van een permanente aard is zodat het niet aan te raden is de actie opnieuw te proberen.

Exceptions zijn wel geen objecten zoals men in bijvoorbeeld Java en dergelijke. Het zijn een soort fun-calls die men ook argumenten kan meegeven door de programmeur toegekend aan signals. Dit kan heel handig zijn om een verstaanbaar failure handling systeem op te bouwen.

Fatale failure handling

Naast het signaling systeem in Argus is er ook een mogelijkheid om te herstellen van fatale crashes. Hiervoor maakt men gebruik van twee soorten objecten:

- Stable objects worden op vaste tijdsintervallen opgeslagen op vaste geheugens zodat ze later kunnen hersteld worden.
- Volatile objects zullen verloren gaan bij een fatale crash

Nadat een crash heeft plaatsgevonden zullen de stabiele objecten terug worden ingeladen in het werkgeheugen en zal de Argus VM de code van de guardians

herstellen zoals ze werden gedefinieerd in de code en kan de normale computatie terug heropgestart worden.

2.5.3 Conclusie

Argus modelleert wegvallende verbindingen via speciale signals (exceptions). Het maakt gebruik van asynchrone berichtgeving met het gebruik van promises en streams om controle over deze berichtgeving te behouden. Deze laatste eigenschappen maken Argus een interessante taal om te bestuderen met het oog op het programmeren in een mobiel gedistribueerde omgeving.

2.6 E

Ten slotte zullen we de taal E [14] bespreken, die zoals de vorige talen, in het leven is geroepen specifiek voor het programmeren van gedistribueerde en concurrente programma's. Het objectmodel van E is object-gebaseerd waardoor het enkel gebruik maakt van objecten (geen klassen zoals in bijvoorbeeld Java). Er zal eerst een grondig overzicht gegeven worden van wat de taal E inhoudt en welke belangrijke features deze taal te bieden heeft. Hierna zullen we een uitgebreide bespreking geven van de beschikbare failure handling.

2.6.1 Gedistribueerd programmeren in E

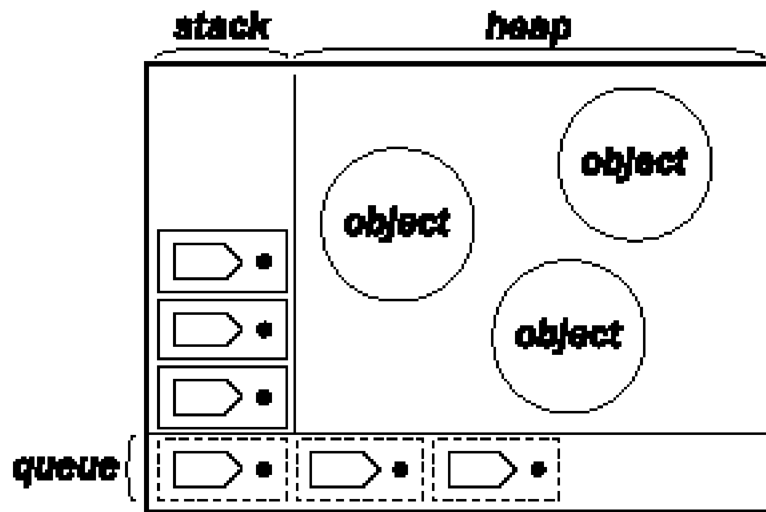
Ten eerste wordt er een onderscheidt gemaakt tussen vat's en machines. Een vat is eigenlijk een verzameling van objecten, deze objecten worden in een heap bijgehouden in elke vat. Een machine is een verzameling vat's. Een belangrijk gegeven bij vat's is het feit dat ze maar één thread hebben voor een hele verzameling objecten, samen met één stack en één queue om de control flow van de berichten goed te regelen. De communicatie hangt af van het type object-referentie tussen twee objecten welke we nu eerst zullen bespreken:

- **Near references** bestaan tussen twee objecten die in hetzelfde Vat zitten. Wanneer een object een near reference bevat van een ander object kan hij zowel near, als eventual communicatie gebruiken om berichten naar dit object te versturen.

- **Eventual references** ontstaan tussen zogenaamde cross-vat objecten, waarmee we willen zeggen dat de twee objecten zich in verschillende vat's bevinden. Een object die een eventual reference bevat van een ander object kan hiermee enkel eventual berichten versturen en zal er een exception teruggegeven worden als men toch probeert near communicatie te gebruiken.

Door het verschil in deze referenties krijgen we ook twee soorten communicatie tussen twee objecten. Het verschil zullen we hier kort beschrijven:

- **Near communicatie** is synchroon en heeft een gewone return value. Het wordt voorgesteld met de dot notatie. Wanneer men te maken heeft met near references zal er gebruik gemaakt worden van dit soort communicatie en enkel tussen deze referenties.



Figuur 2.1: Een representatie van een vat [14].

- **Eventual communicatie** is asynchroon waardoor de sender niet geblockt zal worden bij het versturen van een bericht. De return waarde van dit soort communicatie tussen objecten is een promise en kan dus omgezet worden naar een return waarde wanneer de ontvanger het bericht heeft afgehandeld (< − stelt een eventual send voor). Men kan dit soort communicatie gaan gebruiken bij eventual referenties wat betekent dat dit soort communicatie zowel voor inter-vat als intra-vat communicatie kan gebruikt worden.

Objecten worden doorgegeven via pass-by-reference als men ermee wil gaan communiceren over het netwerk. Om de syntax en semantiek van E wat duidelijker te maken zullen we hieronder een voorbeeld geven van een status holder. `makeStatusHolder` zal een `statusHolder` teruggeven die kan gebruikt worden om een bepaalde status bij te houden met de mogelijkheid om deze ook te wijzigen. Men kan door de variabele `myListener` bepaalde objecten gaan inschrijven op deze `statusHolder` om dan op de hoogte gehouden te worden bij veranderingen in de status. Dit zie je ook bij de definitie van de `setStatus` methode waarbij de `statusChanged` methode eventual verstuurd wordt naar alle listeners.

```
def makeStatusHolder(var myStatus) {
        def myListeners := [].diverge()
        def statusHolder {
                to addListener(newListener) {
                  myListeners.push(newListener)
                }
                to getStatus() { return myStatus }
                to setStatus(newStatus) {
                        myStatus := newStatus
```

```
                    for listener in myListeners {
                        listener <- statusChanged(newStatus)
                    }
                }
            }
            return statusHolder
}
```

Binnenin een Vat zal er gebruik gemaakt worden van een stack om de near communicatie af te handelen en een queue om de eventual communicatie te verzorgen. De normale orde van uitvoering zal eerst alle berekeningen op de stack vervolledigen en daarna de queue te behandelen. Men kan dus besluiten dat near communicatie binnen één vat prioriteit krijgt boven eventual communcatie binnen één vat.
Zoals eerder al eens werd aangehaald in andere talen zoals Argus krijgt men bij een eventual send een promise terug, die men later kan gebruiken als de return waarde van een bericht. Een promise kan zich in twee staten bevinden doorheen zijn levenscyclus:

- De unresolved status: in deze staat zal de promise de rol spelen van een eventual reference, terwijl de eigenlijke waarde van de promise nog niet achterhaald is. Een promise blijft in deze staat zolang het bericht niet werd afgehandeld door de ontvanger van het bericht.
- De ready status: wanneer een eventual bericht afgehandeld wordt verandert de promise naar zijn echt waarde. Als de waarde van deze promise een object blijkt te zijn dat zich in hetzelfde Vat bevindt dan de drager van de promise dan zal deze promise via de `when` constructie in E resolven naar een near reference. Omgekeerd als de promise een object voorstelt in een andere Vat dan zal het resolven naar een eventual reference.

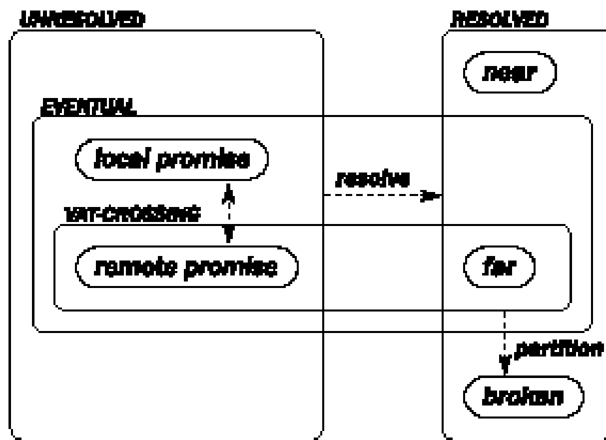
Wanneer er al berichten werden verstuurd gebruik makende van de unresolved promise zullen deze berichten opgeslagen worden in een FIFO queue. Indien de promise zijn ready status bereikt zullen deze berichten in de juiste volgorde worden verstuurd door gebruik te maken van de uiteindelijke waarde van de promise.
Het feit dat we al berichten kunnen sturen naar een unresolved promise geeft de mogelijkheid tot zogenaamd promise pipelining. Dit betekent dat het versturen van een eventual send naar een promise zelf een promise zal teruggeven waar op zijn beurt ook een eventual send kan naar verstuurd worden.

```
def r3 := x <- a() <- c(y <- b())
```

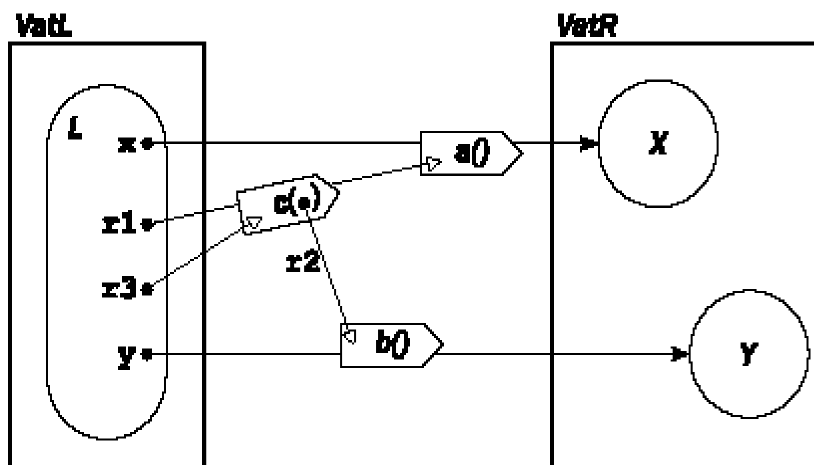
of

```
def r1 := x <- a()
def r2 := y <- b()
def r3 := r1 <- c(r2)
```

Wanneer x en y zich in een andere vat bevinden zal E deze drie requests samen versturen naar die vat. Dit heeft als voordelen dat er maar één keer iets verstuurd hoeft te worden over het netwerk. Een duidelijke representatie van deze promise pipelining vindt men in deze figuur.



Figuur 2.2: Een overzicht van welke soorten references er mogelijk zijn in verschillende omstandigheden. Zoals uit de figuur duidelijk wordt bestaan er twee soorten promises die dan ofwel resolven naar een near of far reference. Bij een failure wordt de promise echter resolved naar een broken reference. [14]



Figuur 2.3: Voorbeeld van pipelining [14]

2.6.2 Failure handling

Om aan failure handling te kunnen doen hebben we natuurlijk eerst fout detectie mechanismen nodig. De taal E maakt gebruik van zijn reference semantiek om aan failure detectie te doen. Er werd al aangegeven dat promises konden resolved worden tot een eventual (far) en near reference, naast deze soorten kan de reference ook broken worden. Een reference is broken wanneer er een netwerk partitie optreedt tijdens de communicatie tussen objecten. Het reference systeem is zo opgebouwd dat er na verloop van tijd gemeenschappelijke kennis zal bestaan van een broken reference. Dit houdt in dat wanneer één cross-vat reference broken zou worden, alle cross-vat referenties tussen de twee vats broken zullen worden. Een laatste belangrijke eigenschap van references tussen objecten is het feit dat wanneer een reference eenmaal broken is geweest hij broken blijft.

We zagen al dat failure detectie via het reference systeem in E wordt verwezenlijkt, hierdoor wordt het groeperen van exception handlers ook op basis van references gedaan. Handlers (functies) worden per reference aangemaakt zoals men in dit code voorbeeld kan aanschouwen.

```
to addListener(newListener) {
        myListeners.push(newListener)
        newListener <- statusChanged(myStatus)
        def handler() { remove(myListeners, newListener) }
        newListener <- _whenBroken(handler)
}
```

Er zijn drie belangrijke berichten die we in meer detail moeten bekijken om een inzicht te kunnen krijgen in hoe het failure handling systeem in E werkt:

- **_whenBroken**: deze message is een universeel bericht dat alle objecten in E verstaan. Wanneer men dit bericht naar een reference stuurt zal deze reference het argument dat meegegeven wordt als handler definiëren en deze dan oproepen als de reference broken wordt.
- **_whenMoreResolved**: wanneer dit bericht naar een reference wordt gestuurd zal het een handler definiëren die zal uitgevoerd worden wanneer de reference geresolved wordt.
- **_reactToLostClient**: als een cross-vat reference broken wordt dan wordt deze message verstuurd naar het object waar deze reference naar wees. Dit bericht wordt dus gebruikt om aan objecten duidelijk te kunnen maken dat bepaalde clients mogelijk geen verbinding meer hebben met het desbetreffende object.

In de eerste plaats zullen normale objecten op een _whenBroken reageren door het te negeren omdat ze niet broken zijn. Een broken reference zal een _whenBroken bericht niet negeren maar wel een eventual bericht sturen naar de handler om deze uit te voeren.

<u>When-catch taal constructie</u>

Door een when-catch constructie kan er omgegaan worden met fouten in eventual messages. Het is de syntax die men kan gebruiken zodat man de _whenBroken en _whenMoreResolved berichten niet altijd manueel zou moeten oproepen. Dus als een promise wordt resolved naar een broken reference dan kan men hiervoor een handler in een catch blok zetten. Om dit aan te tonen zullen we hier een voorbeeld geven van een implementatie van een asynchrone and die gebruik maakt van de expliciete promise structuur in E.

```
def asyncAnd(answers) {
        var countDown := answers.size()
        if (countDown == 0) f return true g
        def [result, resolver] := Ref.promise()
        for answer in answers {
                when (answer) -> {
                        if (answer) {
                                countDown -= 1
                                if (countDown == 0) {
                                        resolver.resolve(true)
                                }
                        } else {
                                resolver.resolve(false)
                        }
                } catch exception {
                        resolver.smash(exception)
                }
        }
return result
}
```

In dit code voorbeeld wordt er expliciet een promise aangemaakt met het vaste duo dat men altijd bij een promise zal terugvinden, namelijk een result en een resolver. Wanneer men naar de definitie van het _whenMoreResolved bericht kijkt, zal men merken dat dit bericht indirect wordt gebruikt in de when-catch block. Wanneer men het smash bericht naar een resolver stuurt zal de promise worden omgezet naar een broken reference.

Offline capabilities

Offline capabilities in E duiden objecten aan op basis van een persistente key, wie de key kent, kan toegang tot het object krijgen. Zoals eerder aangegeven kunnen references eens ze naar een broken status zijn overgegaan niet meer terug connecteren naar het object in de reference. Om dit op te lossen kan een object een offline capability bijhouden die het mogelijk maakt een nieuwe connectie aan te maken naar het doel van een broken reference.
Wanneer een offline capability gevraagd wordt om zulk een nieuwe reference aan te maken zal er een promise teruggeven worden die belooft een connectie te zullen aanmaken. Indien het lukt om een verbinding aan te maken zal de promise een reference worden naar het object naar waar men connecteerde, bij een gefaalde poging zal de promise broken worden met een exception.
De levensduur van een offline capability kan ook gereguleerd worden zodat men

kan bepalen hoelang de reference kan gebruikt worden voor het creëeren van nieuwe references. Deze levensduur kan op drie manieren worden vastgesteld:

- Time-to-live: men geeft een vaste levensduur mee aan de offline capability waardoor het na deze tijd niet meer bruikbaar is.
- Revocable: een offline capability is ook expliciet onbruikbaar te maken.
- Transient: een offline capability kan onbruikbaar worden wanneer de vat die haar bijhield crasht.

Herstel van fatale crashes

E heeft ook een systeem om van fatale crashes te herstellen. In dit systeem maakt E gebruik van twee soorten vat's:

- Volatile vat's worden enkel in het werkgeheugen opgeslagen en zullen dus verloren gaan bij een fatale crash.
- persistente vat's worden op vaste geheugens geplaats wanneer de vat zijn volledige stack leeg is, dit noemt men een checkpoint.

Een belangrijk feit om te onthouden bij dit systeem is de manier waarop men de objecten in de persistente va's gaat opslaan. Alle vat-crossing references zullen bijvoorbeeld opgeslagen worden als broken references zodat bij een reïncarnatie van een vat alle cross-vat references broken zullen zijn. Na een reïncarnatie is het dus aan offline capabilities in beide richtingen (herstelde vat $->$ nog werkende vat, werkende vat $->$ herstelde vat) om nieuwe verbindingen aan te maken.

2.6.3 Conclusie

E heeft een zeer uitgebreid failure handling systeem gebaseerd op references, dat zeer goed werkt in gesloten gedistribueerde omgevingen. De keuze om broken references niet de mogelijkheid te bieden opnieuw te connecteren vormt echter een heel zwaar probleem in mobiele gedistribueerde omgevingen. Het is echter wel mogelijk om verbroken verbindingen te herstellen door gebruik te maken van offline capabilities. Een nadeel dat al veel is voorgekomen in dit hoofdstuk is natuurlijk de manier waarop E failures modelleert als exceptions waardoor het moeilijk schaalt naar een mobiele omgeving. Het systeem van asynchrone berichtgeving en de daarbij horende promises en when-catch constructies maken de programmeertaal E zelf wel geschikt voor gebruik in een mobiele omgeving.

2.7 Conclusie

In dit hoofdstuk hebben we een aantal programmeertalen besproken die vandaag beschikbaar zijn om te bestuderen welke mechanismen er beschikbaar zijn om aan failure handling te doen. Het is de bedoeling dat men uit dit hoofdstuk inspiratie kunnen halen voor mechanismen om aan failure handling te kunnen doen in een mobiele omgeving. Uit de besprekingen is gebleken dat men failure handling steeds gaat beschouwen als exceptions die door een al dan niet asynchroon exception handling mechanisme moeten worden opgevangen.

Hoofdstuk 3

Ambient-georienteerd programmeren

3.1 Inleiding

Bij het schrijven van een applicatie is de keuze van de taal een belangrijk onderdeel. De complexiteit van de code zal hoofdzakelijk afhangen van de toepasbaarheid van de taal op het probleemdomein van de applicatie. In de context van deze scriptie kunnen we ons dus afvragen of er talen beschikbaar zijn die het programmeren van mobiele gedistribueerde applicaties minder complex maken door de programmeur de juiste abstracties aan te bieden. Minder complexe code zorgt er ook voor dat de code beter begrijpbaar wordt, wat gunstig is voor bijvoorbeeld het onderhouden van de code. Voor we dit echter kunnen toetsen moeten we een idee krijgen over welke eigenschappen zo een taal moet bezitten.
Mobiele netwerken zijn heel erg dynamisch omdat de verschillende toestellen die deel uit maken van het netwerk los van elkaar kunnen bewegen. Naast mobiele toestellen worden mobiele netwerken verbonden door middel van draadloze verbindingen. Deze belangrijke eigenschappen hebben tot gevolg dat men rekening moet houden met frequent verbroken verbindingen. Naast zulke verbroken verbindingen zullen er ook vaak nieuwe toestellen zich proberen aanmelden op het netwerk, net zoals er reeds verbonden toestellen zullen verdwijnen. Omdat er een constant komen en gaan optreedt van toestellen noemen we zo een netwerk ook wel een *open* netwerk. De combinatie van de gegeven eigenschappen maakt het programmeren van een applicatie in de gegeven omgeving erg moeilijk, daarom is het zeker in dit geval belangrijk een goede taal ter beschikking te hebben die rechtstreeks ondersteuning biedt om met deze fenomenen om te gaan.
In dit hoofdstuk zullen we nagaan wat we zoal nodig hebben om gemakkelijk applicaties voor mobiele netwerken te kunnen programmeren. Sectie 2 van dit hoofdstuk zal de grote problemen in het programmeren van applicaties voor mobiele netwerken beschouwen zodat we daaruit elementen kunnen afleiden om bruikbare features te kunnen creëren voor een ambient-gebaseerde taal. Nadat we een idee hebben gekregen van wat het "Ambient-Oriented Programming paradigma", kort AmOP, inhoudt zullen we een concrete AmOP

taal beschouwen, AmbientTalk genaamd. We zullen AmbientTalk dan ook evalueren als taal om mobiele applicaties mee te progammeren.

3.2 Software-ontwikkelingsproblemen in mobiele netwerken

We zagen al eerder dat programmeren in een mobiele omgeving erg veel problemen met zich meebrengt die de aandacht van de programmeur vereisen. In deze sectie zullen we de belangrijke moeilijkheden beschrijven van mobiele netwerken. Deze bespreking wordt nadien dan gebruikt om kenmerkende eigenschappen van een ambient-georiënteerde programmeertaal te identificeren. De belangrijkste problemen in het programmeren in een mobiele omgeving werden reeds onderzocht in bestaand werk [3]. De analyse wordt hier kort herhaald:

- **Zwakke verbindingen**. Mobiele apparaten communiceren met elkaar door middel van draadloze verbindingen. Draadloze verbindingen hebben echter een beperkte reikwijdte waardoor connecties verbreken indien toestellen in het netwerk te ver van elkaar verwijderd raken. De gangbare programmeertalen zien echter verbroken verbindingen als uitzonderingen waardoor ze de programmeur verplichten met elke verbroken verbinding om te gaan. Dit zou de code van een mobiele applicatie overspoelen met exception handling code waardoor de code niet meer overzichtelijk zal zijn.

- **Ambient resources**. In een traditioneel, vast netwerk is de locatie van bepaalde data vaak gefixeerd op een welbepaalde node en is de locatie ervan bekend bij elke node van het netwerk. Programma's dienen vaak de nodige resources (bronnen) zoals services en data te ontdekken in hun directe omgeving. Aangezien toestellen mobiel zijn verandert deze omgeving continue zodat het onmogelijk wordt statisch data en services te alloceren.

- **Onafhankelijkheid**. In het vorige hoofdstuk hebben we kennis gemaakt met vele talen die nog steeds gebruik maakten van het client-server model om gedistribueerde applicaties te modelleren. Dit model houdt in dat er een vaste en centrale server beschikbaar is waarmee clients kunnen connecteren en berichten naar kunnen sturen. In een mobiele omgeving is het echter wenselijk geen zulke hiërarchieën op te bouwen tussen toestellen doorheen het netwerk doordat connecties nu eenmaal zwak zijn dus verbinding met een server is onbetrouwbaar. Daarom is het een vereiste dat alle toestellen in een mobiele omgeving onafhankelijk van elkaar kunnen werken. Een peer-to-peer model schaalt dus beter in een mobiel netwerk.

- **Inherent concurrente systemen**. Bij vaste netwerken is het gebruikelijk dat de zender van een bericht over het netwerk wacht op een antwoord van de ontvanger; dit noemt men synchrone message passing. Het feit dat in mobiele netwerken verbindingen zwak zijn, maakt het zeer moeilijk om deze synchrone berichten te behouden. Synchrone berichten zorgen

ervoor dat het normale verloop van applicaties volledig opgehouden wordt omdat de verzender altijd zal wachten op een antwoord van de ontvanger vooraleer normale computatie te hervatten, zelfs wanneer de connectie opeens verbroken wordt.

- **Service discovery**. In een vast netwerk is de locatie van alle toestellen in het netwerk bekend voor alle andere toestellen. De zwakke verbindingen in een mobiel netwerk brengen hier verandering in en brengen een nieuwe nood met zich mee. Men heeft een manier nodig waarmee toestellen anderen kan opzoeken in een dynamische omgeving zonder weet te hebben van anderen hun locatie.

We hebben hier een hele opsomming van vele moeilijkheden die we moeten behandelen indien we mobiele applicaties willen programmeren. Uit [3] blijkt dan ook dat er geen enkele populaire taal van vandaag een antwoord heeft op alle moeilijkheden. Het AmOP paradigma is opgebouwd rond de gegeven opsomming en bezit dan ook de eigenschappen die nodig zijn om gemakkelijk mobiele applicaties te schrijven.

3.3 Ambient Oriented Programming

Het AmOP paradigma werd voorgesteld om bovenstaande problemen aan te kaarten door middel van toegewijde taalconstructies [3]. We herhalen hier kort de kenmerken waaraan een ambient-georiënteerde taal moet voldoen en voegen zelf een criterium toe betreffende foutafhandeling.

3.3.1 Klasseloze objectmodellen

De gangbare talen die vandaag beschikbaar zijn gebruiken klassen om objecten mee aan te maken. In dit soort talen worden de objecten beschreven en aangemaakt door middel van klassen en een object onafscheidelijk verbonden met zijn klasse. Een vast netwerk heeft weinig problemen om met klassen te werken omdat iedereen weet waar alle actoren zich bevinden en men kan dus steeds de juiste klassen opzoeken. Wanneer er updates aan klassen worden gemaakt, kunnen deze gemakkelijk doorheen het netwerk verspreid worden. De problemen vormen zich wanneer men klassen gaat gebruiken in een mobiele omgeving. In een mobiele omgeving is het niet mogelijk een centrale plek te voorzien om klassen in op te slaan. Men is dus verplicht om de klassen van de objecten die worden doorgegeven aan de andere actoren in een mobiel netwerk ook door te geven. Hierdoor gaan vele actoren in het netwerk een eigen kopie verkrijgen van een klasse. Dit voldoet echter niet aan het idee van klassen omdat er conceptueel maar *één* klasse aanwezig is. Het grote nadeel aan dit kopiëren van klassen is het feit dat objecten van dezelfde klasse een ander gedrag gaan vertonen afhankelijk van de machine waarin ze aanwezig zijn.
We zullen dus een ander object model moeten gebruiken in het AmOP paradigma. Het grootste probleem is het feit dat de objecten afhankelijk zijn van hun klasse en niet kunnen bestaan zonder deze klasse. Om dit te

vermijden zullen we een model voorstellen waarin objecten volledig onafhankelijke entiteiten zijn. Een veel gebruikt model in dit kader is het prototype-gebaseerd model [4]. In dit model worden objecten ofwel onafhankelijk (ex-nihilo) opgebouwd, ofwel met een ander object als bouwsteen via het klonen. Na de creatie leven deze objecten onafhankelijk van hun oorsprong verder. Omdat we hier geen klassen meer gebruiken hebben de meeste prototype-gebaseerde talen mogelijkheden om methoden (slots) aan te passen.

3.3.2 Niet-blokkerende communicatie primitieven

We zagen al eerder dat er inherente concurrentie opduikt in mobiele netwerken, waardoor synchrone berichtversturing tussen entiteiten in het netwerk niet langer praktisch is. Sterker nog, idealiter willen we zo weinig mogelijk synchronisatie tussen twee toestellen in een mobiel netwerk omdat zwakke verbindingen steeds de mogelijkheid bieden tot te lange locks. In een mobiele omgeving hebben we dus primitieven nodig die het verloop van een applicatie niet zullen blokkeren. Hiermee bedoelen we niet enkel het gebruik van asynchrone berichtversturing over het netwerk, maar ook bijvoorbeeld niet-blokkerende "receive"-mechanismen, i.e. "event-gebaseerd".

3.3.3 Gereïficeerde communicatiesporen

In de vorige sectie zagen we al dat synchronisatie zoveel mogelijk te vermijden valt tussen toestellen in een mobiel netwerk. Wanneer twee programma's echter concurrent werken zonder te synchroniseren en er fouten optreden in de communicatie tussen deze programma's in een inconsistente staat achtergelaten worden, hebben we een manier nodig om de actoren die werden aangetast opnieuw met elkaar te synchroniseren. Enkel op deze manier kunnen we de actoren weer tot op een punt brengen zodanig dat de normale gang van zaken in de aangetaste objecten hervat kan worden. Men heeft dus een manier nodig om de gedane communicatie voor te stellen zodat men kan proberen de betrokken partijen terug te synchroniseren met elkaar op basis van reeds verzonden of nog te verzenden berichten.

3.3.4 Ambient kennissenbeheer

Het probleem van discovery werd al in de vorige sectie aangehaald samen met een mogelijke oplossing. Vaste netwerken maken gebruik van centrale servers om communicatie met verschillende clients mogelijk te maken. Dit is mogelijk doordat clienten weet hebben van de locatie van deze servers. Zulke servers kunnen dan de rol van "name server" of "directory service" spelen om verschillende clienten met mekaar in context te brengen. In een mobiele omgeving hebben de toestellen meestal geen vaste locatie waardoor verbindingen gebaseerd op locatie niet mogelijk zijn. Een ambient kennissenbeheer (Eng: ambient acquaintance management) systeem is nodig willen we kunnen connecteren met andere toestellen in het netwerk. Het overbrengen van een vereiste dienst of het aanbieden van een bepaalde service kan bijvoorbeeld gebeuren door gebruik te maken van pattern matching. In een systeem dat pattern matching gebruikt kunnen we dus door strings te gebruiken een specifieke nood of aanbieding overmaken. Een andere en meer gesofisticeerde mogelijkheid is het gebruik

van reguliere expressies om andere actoren te zoeken. We willen dus zeker een mogelijkheid tot discovery in ons AmOP paradigma onder welke vorm dan ook. Wat we dus nodig hebben is een manier van service beschrijving in plaats van gebruik te maken van vaste URL's of addressen.

3.3.5 Time-based failure handling

Vele talen zoals Java hebben een heel uitgebreid exception handling systeem dat de meeste detecteerbare fouten kan opsporen en deze mededelen in de vorm van exceptions. We hebben al aangegeven dat we veel van deze exceptions niet willen in een mobiele omgeving omdat deze geen uitzonderingen zijn maar eerder de regel. Bij een verbroken connectie wordt het verloop van de applicatie enkel in de tijd verstoord omdat de communicatie toch wordt hervat als de connectie hersteld wordt. Het zou dus heel handig zijn om een mechanisme ter beschikking te hebben zodat we aan bepaalde berichten een soort deadline kunnen meegeven waarbinnen we willen dat het uitgevoerd wordt. Indien code niet binnen de vooropgestelde deadline werd uitgevoerd, willen we dit behandelen zoals we gewoon zijn bij een try-catch constructie in een standaard exception handling systeem. Deze time-based failure handling is dus een gevolg van de zwakke verbindingen en asynchrone communicatie die typisch zijn voor mobiele netwerken.

3.3.6 Conclusie

Uit de verzameling van problemen hebben we in deze sectie vier grote kenmerken opgesomd die zeker nodig zijn om gemakkelijk mobiele applicaties te schrijven. De afleiding van deze kenmerken vindt men in [3]. We noemen dan ook talen die deze features bezitten ambient-georiënteerde programmeer talen.
AmOP is dus een paradigma dat vier kenmerken van een programmeertaal voorstelt om om te gaan met problemen die we besproken hebben in sectie 3.2 specifiek voor mobiele netwerken. De eigenschappen vormen een goede basis maar zeggen niet hoe men bijvoorbeeld communicatie, discovery of failure handling moet gaan doen. In het volgende hoofdstuk zullen we failure handling in de context van AmOP verder bestuderen. Maar voor we dit kunnen doen zullen we eerst de AmOP taal AmbientTalk bespreken welke ook de taal is waarin onze experimenten geprogrammeerd zijn.

3.4 AmbientTalk

In deze sectie zullen we een taal bespreken die het AmOP paradigma ondersteunt, AmbientTalk genaamd [1]. AmbientTalk is een concurrente, gedistribueerde programmeertaal wiens sequentiële aspect gebaseerd is op de taal Pico. Hierdoor moeten we eerst een korte introductie geven van Pico vooraleer we AmbientTalk in kunnen uitleggen.

3.4.1 Pico

Pico [6] is initieel ontwikkeld als leeromgeving om studenten kennis te laten maken met programmeren. Later werd het meer en meer gebruikt als

onderzoeksplatform om in te experimenteren met nieuwe taal features, omdat de kern van de taal zeer compact en uitbreidbaar is. Pico werd ontwikkeld op het Programming Technology Lab, en ligt aan de basis van een hele reeks talen, waaronder AmbientTalk. Naar analogie met het objectmodel dat werd voorgesteld door Lieberman [4] ontstond Pic% [5]. Deze taal vormde een goede basis, door zijn prototype-gebaseerd objectmodel, om een mobiele gedistribueerde taal rond op te bouwen, AmbientTalk genaamd.
Wat Pico zo geschikt maakte als leeromgeving voor beginnende studenten is zijn simpele syntax, voorgesteld in figuur 3.1, en de gelijkenis met Scheme [7]. De belangrijkste gelijkenissen met scheme zijn:

- First class functies waardoor het mogelijk wordt functies als argumenten mee te geven met andere functies.
- Lexical scoping dat ervoor zorgt dat de omgeving waarin een functie opgeroepen wordt diegene is waarin de functie gedefinieerd werd in plaats van de omgeving waarin de functie aanroepen wordt.
- Dynamically typed waardoor er geen types moeten gespecifieerd worden bij elke variabele.
- Automatic memory management zodat men zich niets hoeft aan te trekken van enige garbage collection bij het programmeren zelf.

Er zijn echter ook een paar verschillen tussen Pico en scheme.

- De syntax van beide talen is heel verschillend maar vooral dan in het feit dat scheme prefix notatie hanteert terwijl Pico de infix notatie gebruikt. Hiernaast maakt scheme ook enkel gebruik van haakjes om zijn syntax in te definiëren.
- Terwijl men in scheme gebruik kan maken van lijsten moet men het Pico doen met fixed-size tabellen.
- Een laatste tevens een belangrijk verschil is het feit dat scheme gebruik maakt van special forms terwijl de natives in Pico geïmplementeerd zijn als call-by-function.

Zoals de naam al aanduidt is Pic% een object georiënteerde uitbereiding van Pico. De syntax van deze taal vindt men ook terug in figuur 3.1. Voor een grondige bespreking van Pic% is [5] aan te raden.

3.4.2 Prototype-gebaseerd objectmodel

In het AmOP paradigma worden zoals eerder al aangehaald klasse-gebaseerde objectmodellen vermeden omdat objecten dan teveel afhankelijk worden van klassen. In deze sectie zullen we een model uitleggen dat reeds in 1986 beschreven werd, namelijk het prototype-gebaseerd objectmodel. Dit object model werd voor het eerst beschreven door Lieberman [4] en zoals we verder in dit hoofdstuk zullen aantonen biedt dit model veel voordelen wanneer het in de context van het AmOP gebruikt wordt.
De informatie over objecten wordt in het prototype-gebaseerde model niet

	Table	Function	Native Type
Definition	a[REF] : EXP	a(PARS) : EXP	a : EXP
Assignment	a[REF] := EXP	a(PARS) := EXP	a := EXP
Reference	a[REF]	a(PARS)	a
Declaration	a[REF] :: EXP	a(PARS) :: EXP	a :: EXP
Message Send	REF.a[REF]	REF.a(PARS)	REF.a
Super Send	.a[REF]	.a(PARS)	.a

Figuur 3.1: Overzicht van de syntax van zowel Pico als Pic% [5].

opgeslagen in statische klassen zoals in het klasse-gebaseerde model. Men gaat eerder objecten creëren door middel van het klonen en uitbreiden van bestaande objecten. In een prototype-gebaseerd model kan men een object prototype (ex-nihilo) aan maken dat kan dienen als basis voor andere objecten via het klonen van objecten. Na het aanmaken van een object door middel van klonen of uitbereiden, kan men dynamisch slots toevoegen aan en verwijderen uit het nieuw aangemaakte object. Slots in het prototype-gebaseerde model komen overeen met instance variabelen en methods in klasse-gebaseerde modellen. Een voorbeeld van een prototype-gebaseerde taal is Self [9].
Een ander groot verschil tussen prototype- en klasse-gebaseerde modellen zit hem in het sharing mechanisme. In klasse-gebaseerde modellen maken we gebruik van overerving als sharing mechanisme tussen klassen. Sharing in een prototype-gebaseerde taal wordt verwezenlijkt door delegatie. Dit houdt in dat wanneer men een bepaalde methode aanroept op een object, en dit object bevat deze methode niet, dan zal dit object de methode delegeren naar zijn ouder (Eng: parent object). Het subtiele verschil tussen delegatie en normale message forwarding zit hem in het feit dat de self referentie bij delegatie steeds naar het oorspronkelijke delegerende object zal wijzen terwijl dit bij message forwarding niet het geval is hetgeen resulteert in de gewenste "late binding" semantiek voor `self` die ook in klasse-gebaseerde talen wordt gebruikt.

3.4.3 Tweeledig objectmodel: actieve vs. passieve objecten

Voor het sequentiële aspect van de taal AmbientTalk wordt gebruik gemaakt van zogenaamde passieve objecten. Passieve objecten kan men vergelijken met normale objecten die voorkomen in elke object georiënteerde taal. Men heeft ervoor gekozen deze objecten in AmbientTalk te introduceren omdat gewone sequentiële programma's geen concurrente machinerie vereisen en er veel makkelijker te werken valt met deze passieve objecten, bijvoorbeeldin verband met synchrone methode invocatie. Het gebruik van passieve objecten heeft echter tot gevolg dat er strenge regels moeten worden ingevoerd opdat er geen zogenaamde race-conditions zouden optreden. Race conditions komen voor wanneer een bepaalde entiteit wordt veranderd of geraadpleegd door meer dan één thread of proces tegelijkertijd. Men heeft dan ook twee basisregels ingevoerd die deze race-conditions op passieve objecten moet kunnen voorkomen [3]:

- Een passief object mag maar voorkomen binnen *één* enkel actief object zodat er geen sharing mogelijk is van passieve objecten tussen verschillende actieve objecten.
- Wanneer men passieve objecten verstuurt tussen actieve objecten door middel van argumenten in asynchrone berichten, breekt men de sharing regel. Het versturen van passieve objecten zal hierdoor leiden tot het kopiëren van deze objecten waardoor er een nieuw object zal terechtkomen bij de ontvanger. Let wel, dit kopiëren wordt enkel toegepast wanneer we spreken over passieve objecten, actieve objecten zullen doorgegeven worden door middel van hun referentie en kunnen dus wél gedeeld worden.

AmbientTalk maakt gebruikt van actoren [8] als de entiteiten in de gedistribueerde context van de taal. De communicatie tussen deze actoren gebeurt asynchroon, wat wil zeggen dat de actor die het bericht verzendt niet zal wachten op een antwoord vooraleer verder te doen met de rest van zijn computatie. De asynchrone berichten worden opgeslagen in een zogenaamde *message queue*, waar ze zullen wachten tot ze kunnen behandelt worden door de actor. Elke actor heeft één thread waarin hij de berichten in zijn queue een voor een zal behandelen. Dit sequentieel behandelen van inkomende berichten zal ervoor zorgen dat geen inconsistenties kunnen voorkomen in de interne staat van de actor op het vlak van concurrency. In de rest van de scriptie worden de termen actor en actief object als synoniemen beschouwd.
In een normale AmbientTalk applicatie zullen actieve objecten verschillende, minstens één (het gedrag van een actor is een passief object), passieve objecten bevatten. Een bepaalde node in het netwerk kan dan weer verschillende actieve objecten bevatten met mogelijke referenties naar andere actieve objecten die zich in andere nodes in het netwerk bevinden.

3.4.4 Passieve object laag

Passieve objecten in AmbientTalk volgen het prototype-gebaseerde objectmodel dat we al eerder hebben gesproken. We zullen nu een simpel counter voorbeeld introduceren om de syntax van AmbientTalk's passieve objecten uit te leggen.

```
{
  makeCounter() :: object({
    i : 0;
    inc() :: { i := i + 1 };
    decr() :: { i := i - 1 };
    cloning.cloneMe(v) :: {
      i := v
    };
  });

  makeProtectedCounter(limit) :: {
    makeCounter().extend({
      inc() :: { if( i = limit,
                     error("limit has been reached"),
                     .inc())}
```

```
    })
  }

  Counter : makeCounter();
  cloningCounter : Counter.cloneMe(4);
  protectedCounter : makeProtectedCounter(5)
}
```

Door gebruik te maken van de `object(..)` primitieve kunnen we objecten aanmaken. Als argument dient een expressie te worden meegegeven waarin de verschillende slots worden gedefinieerd volgens het prototype-gebaseerde object model. Men kan twee verschillende soorten slots tegen komen.

Value slots (variabel slot) dienen om zogenaamde instance variabelen in op te slaan zoals `i` in het bovenstaand voorbeeld.

Method slots (constant slot) worden gebruikt om de methoden in te definiëren die men zal kunnen aanroepen op het object. Een voorbeeld van method slots is bijvoorbeeld `inc()` en `cloneMe` in het voorbeeld.

De declaratie van variabele en constante slots kan men afleiden uit de syntax van Pic%, namelijk " : " voor variabelen en " :: " voor constanten. In tegenstelling tot variabelen kan men geen assignment toepassen op constanten en zullen deze constanten geshared worden bij het klonen. Zoals we ook bij de syntax van Pic% in figuur 1 zagen kunnen we in AmbientTalk berichten versturen door de dot notatie. Het creëren van nieuwe objecten wordt verwezenlijkt door bestaande objecten te klonen of uit te breiden, AmbientTalk bezit dan ook de twee primitieven extend en cloning om deze taken te vervullen. We zullen deze twee primitieven uitleggen aan de hand van het gegeven code voorbeeld. Cloning zal ervoor zorgen dat we een nieuw counter object (cloningCounter) terug krijgen waarbij de variabele `i` zal gebonden worden aan de parameter, die meegegeven werd aan de cloning methode `cloneMe`. `Extend` [3] zorgt er dan weer voor dat we een nieuw object (protectedCounter) terug krijgen waar er nieuwe slots in kunnen gedefinieerd worden. Het nieuw aangemaakte object zal echter `makeCounter` als zijn ouder zien. Dit heeft als voordeel dat we methoden kunnen overschrijven. In het voorbeeld maken we van deze eigenschap gebruik om de methode `inc` van de `protectedCounter` te voorzien van een extra if test om te kijken of de limiet nog niet overshreden is. Wanneer dit niet het geval is kunnen we de `inc` methode gewoon delegeren naar zijn ouder door de `.inc()` aanroep. Hierdoor hebben we dus extra gedrag toegevoegd aan de extentie van `makeCounter` door gebruik te maken van de `extend`.

3.4.5 Actieve objectlaag

In AmbientTalk worden actors opgebouwd in twee fasen. In een eerste fase wordt er een passief object aangemaakt met de nodig slots die het gedrag van de actor zullen bepalen (het behaviour object). Hierna wordt aan het gecreëerde passieve object een message queue en een thread toegevoegd zodat het als het ware actief wordt. Dit proces kan men ook afleiden uit de syntax bij het creëren van een actief object: `actor(o)` waarbij `o` een passief object voorstelt. Bij de creatie van een actor zal er onmiddellijk een `init()` bericht verstuurd worden

naar deze actor zodat deze zichzelf kan initialiseren. Vanaf dat moment zal `thisActor` de huidige actor aanwijzen. Men kan asynchrone berichten versturen tussen actieve objecten door gebruik te maken van de #-notatie. We hebben dus twee verschillende manieren om berichten te versturen, de ene synchroon, de andere asynchroon. Het is niet toegestaan om asynchrone berichten te versturen naar een passief object en geen synchrone berichten naar een actief object.

```
{
  makecounter() :: object({
    i : void;
    inc() :: { i := i + 1 };
    decr() :: { i := i - 1 };
    init() :: { i := 0 }
  });

  counterActor : actor(makecounter());
  counterActor#inc()
}
```

Het bovenstaande code voorbeeld toont een simpel actief object. Het eerder al vernoemde counter object kan ook als gedrag van een actor gebruikt worden zoals men hier kan zien. Bij het aanmaken van het object makecounter zal eerst de `init()` methode aanroepen worden. Hierna kunnen we asynchrone berichten naar deze counter versturen door middel van de #-notatie. Het verschil met het vorige voorbeeld is dat deze counter zijn eigen thread encapsuleert en dus zelf zijn methoden zal uitvoeren.

3.4.6 Mailboxen

In sectie 3.4.3 werden reeds zogenaamde message queues besproken. Deze message queues zullen we verder in deze scriptie ook `mailboxen` gaan noemen. De `inbox` is een van deze mailboxen die actoren in AmbientTalk standaard bezitten. Naast de `inbox` bestaan er echter meer mailboxen, elk met een eigen taak. In deze mailboxen kunnen actieve objecten allerlei zaken opslaan en opvragen. De eerste soort van mailboxen die we gaan bespreken hebben we eigenlijk al besproken wanneer we het hadden over het versturen van berichten tussen actieve objecten. De `in` mailbox vervult de rol van message queue zodat actoren al hun inkomende berichten op een sequentiële manier kunnen uitvoeren in hun eigen thread. Verzonden asynchrone berichten worden dan weer opgeslagen in de `out` mailbox van de verzender, waar deze mailbox de taak van voorlopig opslagmedium vervult totdat dit bericht over het netwerk kan verstuurd worden.
Wanneer we terugdenken aan de vier grote features die vereist zijn in het AmOP paradigma hebben we er nog twee niet behandeld in het geval van AmbientTalk. Het interessante aan mailboxen is het feit dat ze deze twee features beide mogelijk maken. De eerste feature die we zullen bespreken is de zogenaamde gereïficeerde communicatiesporen (Eng: Reified Communication Traces) die we al in sectie 3.3.3 besproken hebben. Deze feature hield in dat we een mogelijkheid willen hebben om gedane communicatie op te slaan zodat we bij inconsistenties alle betrokken partijen opnieuw kunnen synchroniseren tot op een punt waar beide partijden nog in een consistente staat verkeerden. De `rcv`

mailbox zal alle berichten opslaan die een actor toegestuurd heeft gekrege en welke reeds verwerkt werden. Equivalent met de `rcv` box zullen alle succesvol verstuurde berichten opgeslagen worden in de `sent` box. Als we nu de kennis van de `rcv`, `sent`, `in` en `out` box samen gaan gebruiken, krijgen we een duidelijk beeld van welke handelingen succesvol worden uitgevoerd en welke nog niet zijn behandeld. Hierdoor zouden we dan ook nieuwe states kunnen creëren voor inconsistente actoren en voldoen actors in AmbientTalk aan de gereïficeerde communicatiesporen (Eng: Reified Communication Traces) eigenschap van het AmOP paradigma.
Om de laatste AmOP eigenschap, Ambient kennissenbeheer (Eng: Ambient Acquaintance Management), mogelijk te maken in AmbientTalk werden er vier bijkomende mailboxen geïntroduceerd: `joinedbox`, `disjoinedbox`, `requiredbox` en de `providedbox`. Bij het bespreken van deze AmOP eigenschap was al eens aangehaald dat er een mogelijkheid moest bestaan een vereiste dienst en/of een aanbieding over te kunnen maken aan anderen in het netwerk. Het systeem in AmbientTalk maakt gebruik van pattern matching om het ontdekken van services mogelijk te maken tussen actoren:

- `providedbox`. In deze mailbox kan de actor strings over services, die hij aanbiedt aan andere actoren die met hem zouden connecteren, plaatsen.
- `requiredbox`. Wanneer bepaalde actoren nood hebben aan een bepaalde service kan hij een string die deze vereiste dienst representeert in deze mailbox plaatsen.

Het eigenlijke matchen gebeurt op het moment dat een string uit de `providedbox` van de ene actor overeenstemt met een string uit de `requiredbox` van een andere actor. Wanneer er een match plaatsvindt, zal er een *resolution* object worden toegevoegd aan de `joinedbox` van de actor die de service nodig heeft. Dit resolution object bevat de string waarop de match is gebeurd en een referentie naar de actor die de service aanbood. Indien de connectie tussen de twee actoren zou verbreken, wordt het resolutie object verplaatst naar de `disjoinedbox`. Dit systeem van vier mailboxen vormt een complete set van tools die het mogelijk maken de Ambient kennissenbeheer eigenschap uit het AmOP paradigma te verwezenlijken. We zullen hier een voorbeeld geven van een simpel printer voorbeeld om het gebruik van deze tools te tonen.

```
gebruikerActor() :: actor(object({
  init() :: {
    .init();
    required.add("printer");
    joinBox.addAddObserver(thisActor()#joined)
  };
  joined(resolution) :: {
    display("We have connected with actor ",provider(resolution), eoln)
  }
}))
printerActor() :: actor(object({
  init() :: {
    .init();
    provided.add("printer")
```

```
  }
}))
```

Het voorbeeld toont de code van twee actoren, `gebruikersActor` en `printerActor`. In de `init` methode plaatsen de beide actoren de "printer" string in de nodig mailbox zodat er pattern matching kan gebeuren op de string. In de actor die een dienst vereist kan men door een observer op de `joinbox` te plaatsen een methode triggeren wanneer er een verbinding wordt gemaakt met een printer. De `joined` methode zal dus opgeroepen worden wanneer er een verbinding wordt gemaakt met een actor die "printer" in zijn `providedbox` heeft staan en dicht genoeg in de buurt is gekomen van de `gebruikerActor`.

3.4.7 Meta-Object Protocol

Met het MOP in AmbientTalk willen we de interne staat van de interpreter kunnen opvragen en veranderen naar de wens van de programmeur. De voorstelling van deze interne staat wordt door de acht ingebouwde mailboxen en hun observers voor hun rekening gehouden. Hierdoor kunnen we deze eigenschappen al aanzien als deel van het MOP van AmbientTalk. Naast deze eigenschappen bezit het MOP van AmbientTalk nog vele andere kenmerke welke we in deze sectie in detail zullen bespreken.

First-class messages maken het mogelijk alle informatie op te vragen van een asynchroon bericht. Bij de aanroep van een asynchroon bericht, anActor#aMessage(), zal er eerst een MOP methode `createMessage(sender, dest, name, args)` aangeroepen worden om een first-class bericht object aan te maken in de actor die het bericht verstuurd. Wanneer men dan dit first-class bericht heeft aangemaakt kan men alle informatie die meegegeven werd met de `createMessage` opvragen.

Message sending en processing kan gemanipuleerd worden zodanig dat er geen standaard gedrag meer voorkomt in het zenden en verwerken van berichten. Hierboven zagen we al dat bij het versturen van een asynchroon bericht er een MOP methode `createMessage` opgeroepen werd. Hierna zal dit bericht object in de `outbox` geplaatst worden van `anActor` door middel van een MOP call van `send(msg)`, waarbij `msg` het bericht object voorstelt. Wanneer er een bericht in de `outbox` terecht komt zal de AmbientTalk interpreter het proberen te verzenden. Dit verzenden houdt in dat het bericht verplaatst wordt van de `outbox` van de verzender naar de `inbox` van de ontvanger. De thread van de ontvanger zal er dan voor zorgen dat zijn `inbox` een voor een leeggemaakt wordt. Het behandelen van een bericht gebeurd door het oproepen van de `process(msg)` method. Nadat het bericht correct werd afgehandeld zal het verhuizen naar de `rcvbox`. Dit traject wordt afgelegd door elk bericht in AmbientTalk en zoals men kan merken kan men door overschrijving en verfijning van de juiste methoden het gedrag van dit traject veranderen. Wanneer we dus het delegatie systeem gebruiken om bepaalde MOP methoden te verfijnen kunnen we een ander gedrag bekomen dan standaard mogelijk is. Om dit aan te tonen zullen we hier een voorbeeld tonen hoe men elke verzending in een actor kan gaan loggen.

```
send(msg) :: {
  display("sending ... ", msg.getName());
  .send(msg)
}
```

Deze methode verfijnt de `send` MOP methode doordat er een super send aanwezig is die ook het standaard gedrag zal oproepen van de `send`. Door de code uit dit voorbeeld kunnen we dus een display doen wanneer er een verzending gebeurd van een asynchroon bericht. Men ziet hier ook de voorheen besproken first-class berichten doordat men de naam van het bericht opvraagt in de `display`.

first-class mailboxen in AmbientTalk vormen uitstekende tools om nog meer toegang te hebben tot de interne staat van de AmbientTalk interpreter zodat ze een belangrijk onderdeel zijn van het MOP zoals eerder al werd aangehaald. Later in deze scriptie zullen we enkele voorbeelden bespreken hoe we deze eigenschap van mailboxen kunnen gebruiken om nieuwe features te implementeren. De volgende operaties op mailboxen worden ondersteund:

- add(el): voegt een nieuw element toe aan de mailbox.
- delete(el): verwijdert een element uit de mailbox.
- iterate(code(el)): met deze methode kan men over alle elementen van een mailbox itereren.
- Men kan ook listeners definiëren op mailboxen zodat er bijvoorbeeld een methode kan opgeroepen worden bij elke JOIN actie.

Om al een idee te krijgen van het gebruik van deze operaties in AmbientTalk zullen we hier een voorbeeld geven van een router actor. Deze actor zal alle inkomende berichten gaan doorsturen naar een andere actor zonder deze zelf te verwerken.

```
routerActor(anotherActor) :: actor(object({
  routeTo : void;
  init() :: {
    .init();
    routeTo : AnotherActor;
    inBox.addAddObserver(thisActor()#route)
  };
  route(msg) :: {
    inBox.delete(msg);
    msg.setTarget(anotherActor);
    outBox.add(msg)
  }
}))
```

De `routerActor` neemt de actor waarnaar moet geforward worden als parameter en wordt in de `init` fase in de `routeTo` variabele gezet. In de `init` wordt er ook een observer op de `inbox` gezet zodat de `route` methode zal opgeroepen worden telkens wanneer er een bericht in de `inbox` binnenkomt. Wanneer er dan een bericht in de `inbox` zit zal dit onmiddelijk verwijdert worden en de bestemming

van het bericht verandert naar de actor naarwaar geforward moet worden. Dit is mogelijk omdat we natuurlijk gebruik kunnen maken van first-class berichten in AmbientTalk. Na het invoegen van de nieuwe bestemming wordt het bericht dan in de `outbox` van `routeActor` gezet zodat het bericht verstuurd kan worden naar `anotherActor`.

taalmixins (language mixins) zijn de manier in AmbientTalk om taalextenties te maken die op een modulaire en herbruikbare manier kunnen neergeschreven en toegepast worden op verschillende actoren. Met deze taalmixins willen we dus specifieke blokken code gaan gebruiken om gedrag van specifieke actoren te gaan uitbreiden. Om dit mogelijk te maken gebruikt AmbientTalk natuurlijk delegatie als methode te gaan overschrijven en/of verfijnen. Delegatie houdt in dat wanneer een bepaalde methode wordt opgeroepen en het ontvangende object hier niet op kan reageren, het bericht gedelegeerd wordt naar zijn ouder. Deze delegatie zal ervoor zorgen dat we onder de standaard omgeving van een passief object andere omgevingen kunnen hangen om zo bestaande methode definities te overschrijven. Taalmixins zijn omgevingen die men onder het passieve gedrag van een actor moet hangen om het gedrag ervan te veranderen, mixins zijn modulaire uitbreidingen, geparameteriseerd met zijn super-object. Om dit systeem duidelijker uit te leggen zullen we het eerder gegeven logging voorbeeld gaan inmixen bij het counter voorbeeld zodat we in de counter actor een logging krijgen bij elke inkomende berichten.

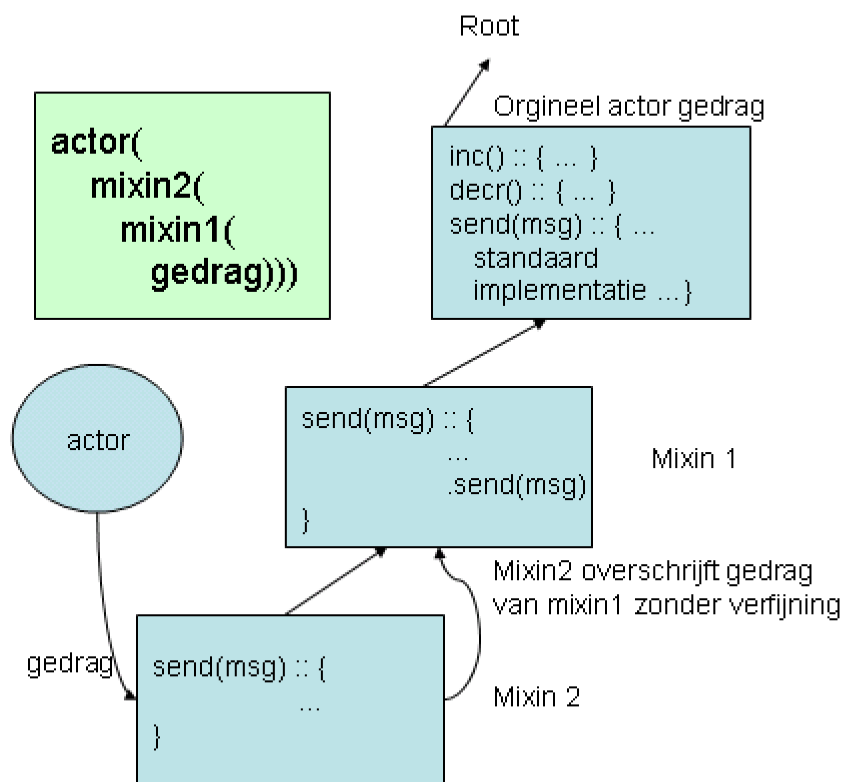
```
{
  logging() :: {
    send(msg) :: {
      display("sending ... ", msg.getName());
      .send(msg)
    };
    capture()
  };

  makecounter() :: object({
    i : void;
    inc() :: { i := i + 1 };
    decr() :: { i := i - 1 };
    init() :: { i := 0 }
  });

  counterActor : actor(makecounter());
  loggingCounterActor : actor(makecounter().logging());
  loggingCounterActor#inc()
}
```

`Logging` in het voorbeeld stelt onze taalmixin voor en bestaat dus uit een verfijnde implementatie van de `send` MOP methode zoals eerder al besproken en eindigt met de `capture()` oproep. Deze oproep is nodig om de mixin omgeving onder de omgeving te hangen van het object dat men wil uitbreiden. Door deze `logging` taalmixin kunnen we nu gemakkelijk twee verschillende soorten counters maken. De `loggingCounterActor` zal dus bij het ontvangen van

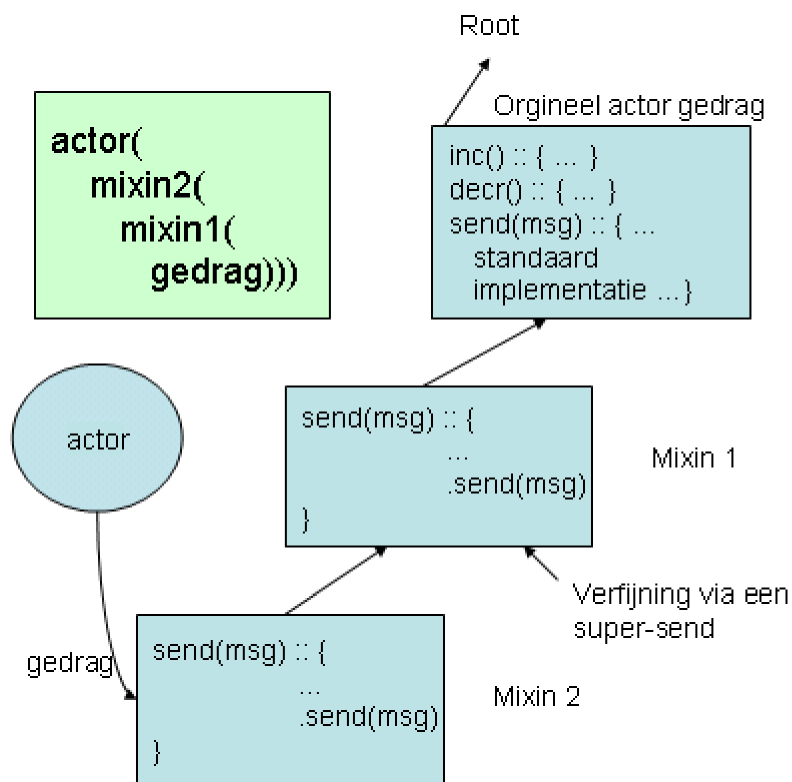
de `inc()` methode de display uitvoeren uit de mixin en daarna de standaard implementatie van de MOP methode `send` verder volgen.
Men kan wel bemerken dat we zeer voorzichtig moeten zijn bij het combineren van verschillende taalmixins op een actor zodanig dat we niet per ongeluk bepaalde methoden gaan overschaduwen die we niet willen overschrijven. Het is dus een vereiste om regels in te voeren met betrekking op naamgeving zodat verschillende van deze overschrijvende omgevingen elkaar niet gaan overschrijven. Bij het overschrijven van MOP methoden is het heel belangrijk om in de overschreven methoden expliciet een eigen delegatie te implementeren. Deze expliciete delegatie is nodig bij het gebruik van meerdere omgevingen onder de basis omgeving zodat elke overschrijving nog steeds correct zal reageren wanneer MOP methoden worden aangeroepen. Verfijning van gedrag wordt dus verwezenlijkt door gebruik te maken van deze expliciete delegatie in tegenstelling tot het overschrijven van methoden.
Mixins zijn de belangrijkste tools om nieuwe features toe te voegen aan AmbientTalk. Hoewel dit een zeer krachtige tool is moeten we zeer voorzichtig zijn wanneer we vele mixins gaan samenstellen, zoals aangegeven in de vorige bespreking.

3.5 Conclusie

In dit hoofdstuk hebben we een opsomming gegeven van de belangrijkste problemen in het programmeren in mobiele netwerken. Naast deze problemen hebben we het AmOP paradigma besproken dat oplossingen biedt voor de voorheen aangehaalde problemen met het programmeren in mobiele netwerken. We hebben dan ook kunnen concluderen dat de taal AmbientTalk helemaal voldoet aan het AmOP paradigma zodat we deze taal zullen gaan gebruiken in verder experimenten in dit paradigma. Een belangrijke eigenschap van AmbientTalk is het gebruik van een tweeledig objectmodel dat bestaat uit actieve en passieve objecten. Actieve objecten (actoren) hebben hun eigen thread en een message queue zodat asynchrone berichgeving mogelijk wordt. Het gedrag van actieve objecten wordt gedefinieerd in een passief object zodat een actor altijd zeker één passief object bevat. De mailboxen in AmbientTalk zorgen er dan weer voor dat vele AmOP eigenschappen aanwezig zijn in de taal. Het MOP van AmbientTalk werd ook beschreven, daaruit konden we concluderen dat het vele mogelijkheden biedt om de interne staat van AmbientTalk te consulteren en aan te passen.
Met de kennis van het AmOP paradigma in het achterhoofd kunnen we nu de taalconstructies die we in hoofdstuk 1 zagen nu gaan evalueren tegen dit paradigma.



Figuur 3.2: Hier kan men zien wat er gebeurd wanneer men geen expliciete delegatie gaat invoeren in een mixin. De vierkante boxen stellen objecten voor en de opwaartse pijlen kan men beschouwen als de delegatie link. Het eerste geval toont wat er kan gebeuren als een mixin geen expliciete delegatie bevat. Het grote probleem hier is het feit dat het standaard MOP geen aanroepen meer zal krijgen, wat de goede werking van bepaalde mixins kan schaden.



Figuur 3.3: Een voorbeeld van een compatibele mixin compositie.

Hoofdstuk 4

Evaluatie van foutafhandelingsmechanismen voor ambient-georiënteerd programmeren

In dit hoofdstuk zullen we bestaande failure handling evalueren in de context van het zojuist besproken AmOP paradigma [3]. De in hoofdstuk 2 besproken middelen om aan failure handling te doen in hedendaagse talen zullen gebruikt worden bij deze evaluatie. We kunnen deze hedendaagse features in enkele categoriën onderverdelen de welke we een voor een zullen evalueren ten opzichte van het AmOP paradigma.

Try-catch exception handling is die vorm van exception handling die men terugvindt in populaire hedendaagse talen zoals Java. In het `try` code block bevindt zich de code die die de dynamische scope afbakent voor het afhandelen van exceptions, daarnaast biedt het ook de mogelijkheid om handlers te definiëren voor verschillende soorten van fouten.

When-catch exception handling . Het `when` gedeelte uit dit soort van exception handling slaat op het feit dat er hier gebruik gemaakt wordt van futures. De behandeling van exceptions gebeurd in deze systemen asynchroon op het moment dat futures worden geresolved.

Observer-based exception handling . Mogelijkheid om listeners naar fouten te definiëren op entiteiten in de omgeving. Men kan handlers inschakelen om de opgevangen fouten te kunnen behandelen naargelang de fout en de entiteit in de omgeving waarop de handler is gedefiniëerd.

Fatale failure handling . Fatale fouten komen voor bij systemen die volledig crashen, dit wil zeggen dat gegevens in het RAM geheugen verloren gaan. Deze fouten worden opgevangen door middel van opslag op vaste geheugens.

Unavailable handlers . Wanneer een bepaalde entiteit in een netwerk niet meer kan gecontacteerd worden zal het beschouwd worden als *onbeschik-*

baar. Bij dit soort van failure handling kan men dus handlers definiëren die het onbeschikbaar worden van een entiteit kunnen behandelen.

In dit hoofdstuk wordt bestudeert of en in welke mate deze bestaande exception handling mechanismen kunnen toegepast worden in de context van het AmOP paradigma.

4.1 Criteria voor ambient-georiënteerde foutafhandelingsmechanismen

Voor we echter kunnen beginnen aan de bespreking over de mate waarin bestaande exception handling mechanismen kunnen toegepast worden in de context van het AmOP paradigma, zullen we eerst drie criteria definiëren die ons in deze bespreking zullen helpen. Deze criteria zijn gebaseerd op de bespreking van het AmOP paradigma uit hoofdstuk 3 en zijn meestal essentieel om aan failure handling te doen in een AmOP omgeving.

Asynchroon omgaan met exceptions (failures) is heel belangrijk in het programmeren in een mobiel gedistribueerde omgeving. Dit werd al aangetoond in hoofdstuk 3 omdat het een eigenschap is van het AmOP paradigma zelf en het kan dus niet ontbreken in criteria voor failure handling in een mobiele omgeving.

Abstractie over tijdelijke verbroken verbindingen is essentieel om aan failure handling te doen in een AmOP omgeving. Zoals we al in hoofdstuk 3 zagen bestaan mobiele netwerken hoofdzakelijk uit draadloze verbindingen welke een grotere kans hebben om te verbreken. Zulke verbroken verbindingen kunnen echter vaak hersteld worden doordat bijvoorbeeld de toestellen terug in elkaars zendbereik bewegen. Het is dus essentieel dat men in failure handling voor een mobiele omgeving de mogelijkheid heeft om een onderscheid te kunnen maken tussen de tijdelijke verbroken verbindingen en permanente failures die we wel willen gaan behandelen. Het is echter belangrijk om weten dat men nooit kan weten wanneer een verbroken verbinding slechts tijdelijk dan wel permanent verbroken is. Men kan dit onderscheid enkel kunstmatig maken.

Blokgestructureerde foutafhandeling is een nuttige eigenschap van elk foutafhandelingsmechanisme om aan failure handling te doen. Het geeft de mogelijkheid om een foutafhandelaar te specifiëren voor meer dan één verzonden bericht. Dit criterium is belangrijk in de context van software-engineering omdat het een foutafhandelingsmechanisme meer modulair maakt zodat het beter beheerbaar en aanpasbaar wordt.

We zullen nu deze criteria gaan gebruiken om de in hoofdstuk 2 besproken foutafhandelingsmechanismen te evalueren naar hun nut in een AmOP omgeving.

4.2 Try-catch exception handling

In dit deel zullen we de voor- en nadelen bekijken van het try-catch exception handling systeem met betrekking tot het AmOP paradigma. We zullen de bruikbaarheid van de syntax en de semantiek bestuderen in een ambient-georiënteerde omgeving.

4.2.1 Exceptions zijn niet langer uitzonderlijk

Wanneer we exception handling gaan gebruiken voor de behandeling van netwerkfouten zoals wegvallende verbindingen moeten we rekening houden met het feit dat hedendaagse exception handling zulke fouten bekijkt als uitzonderlijk. In het AmOP paradigma zijn fouten echter niet meer zo uitzonderlijk doordat we te maken krijgen met volatiele verbindingen die we al besproken hebben in hoofdstuk 3. Het gebruik van exception handling in het AmOP paradigma heeft als gevolg dat men de code moet "vervuilen" met extreem veel exception handling code omdat elk verzonden bericht kan falen en moet voorzien worden van foutafhandelingscode.

4.2.2 Abstractie over tijdelijke verbroken verbindingen

Zoals al uit de bespreking van de criteria voor failure handling in een AmOP omgeving bleek is het essentieel dat men abstractie kan maken over tijdelijke verbroken verbindingen. In een exception handling systeem zal men elke verbroken verbinding echter aanschouwen als een failure zonder er rekening mee te houden of het nu een tijdelijke of een permanente verbreking is. Het feit dat we te maken hebben met volatiele verbindingen (hoofdstuk 3) maakt het dus onmogelijk om een exception handling systeem te gebruiken. Het try-catch systeem dat we bijvoorbeeld bij ProActive zagen voldoet dus al zeker *niet* aan dit criterium.

4.2.3 Niet blokkerende communicatie

Een belangrijke eigenschap die we nodig hebben om gemakkelijk ambient-georiënteerd te kunnen programmeren is het feit dat we niet-blokkerende communicatie nodig hebben, zoals bepaald is in het eerste criterium uit 4.1. Wanneer we het onderliggend code voorbeeld in ProActive [15] bekijken zien we dat er extra taal primitieven nodig zijn om een try-catch systeem te integreren in een systeem dat gebruik maakt van asynchrone berichtgeving. De moeilijkheid om het try-catch systeem samen met asynchrone berichten te gebruiken ligt hem in het feit dat we aan het einde van een `try` moeten wachten op alle asynchroon verstuurde berichten om er zeker van te zijn dat er zich geen exceptions voordeden. Dit betekend dus dat we blokkerende primitieven nodig hebben voordat we exceptions überhaupt kunnen opvangen. Aangezien een belangrijke eigenschap om aan failure handling te doen in het AmOP paradigma luidt dat men asynchroon moet omgaan met exceptions, is de try-catch exception handling helemaal niet geschikt voor gebruik in een AmOP taal.

```
ProActive.tryWithCatch(SomeException.class);
  try {
    Result r = someAO.someMethodCall(); // asynchroon bericht
```

```
    doSomethingWith(r);
    ProActive.endTryWithCatch(); // wachten op r
  } catch (SomeException se) {
    doSomethingWithMyException(se);
  } finally {
    ProActive.removeTryWithCatch();
  }
```

In het voorbeeld zien we het probleem van blokkerende communicatie duidelijker. Voor de `catch` enigszins zou kunnen opgeroepen worden moet het bericht `someMethodCall` afgehandeld worden door het remote object en er moet dus een return waarde teruggegeven worden. Deze return waarde kan op zijn beurt dan een exception zijn of een object van type Result.

4.2.4 Try-catch als taalconstructie

Hoewel exceptions niet de aangewezen techniek zijn om aan failure handling te doen bezit de try-catch taalconstructie enkele bruikbare elementen die gebruikt kunnen worden in de ontwikkeling van failure handling mechanismen. Belangrijke eigenschappen zijn bijvoorbeeld:

- Codespecifieke foutenbehandeling. In een try-catch constructie kan men fouten gaan opvangen in de dynamische scope van een blok code, alle code die de dynamische trace van dit blok code uitgevoerd werd zal dus op een zelfde manier afgehandeld worden met het ook op exception handling. De mogelijkheid om fouten te behandelen in zeer specifieke gevallen heeft als voordeel heel expressief te zijn, wat het programmeren van fouten behandeling gemakkelijker maakt. Try-catch voldoet dus aan het criterium voor blokgestructureerde foutafhandeling.

- Handler delegatie. Bij elk code block is het in een try-catch constructie ook mogelijk meer handlers te definiëren voor verschillende soorten exceptions. Wanneer er echter een bepaalde exception niet wordt opgevangen bij het code block zelf, dan zal deze exception worden doorgegeven aan bovenliggende try-catch constructies. De mogelijkheden die deze delegatie biedt, samen met het feit dat men meerdere handlers per try-catch block kan definiëren, maken try-catch taalconstructies heel expressief.

4.3 When-catch exception handling

Hier volgen de voor- en nadelen van een when-catch exception handling systeem. Voorbeelden van talen waar men zo een vorm van fouten afhandeling in terugvindt zijn E en Argus. Omdat de systemen op bepaalde vlakken verschillen in beide talen zullen ze per taal besproken worden.

4.3.1 When-catch in E

Zoals te zien is in het code voorbeeld moet men in de taalconstructie aanwezig in E [14] een promise als eerste argument meegeven in het "when" deel. Na het when deel kan men een code block definiëren dat zal uitgevoerd worden wanneer

de promise geresolved wordt. In hoofdstuk 2 zagen we al dat E met referenties werkt en dat deze na een verbreking van de verbinding naar een "broken" status wordt herleid. De promise die men in het when deel moet invullen is zodoende ook een referentie, wanneer deze geresolved wordt naar een "broken" status via een exception wordt het catch block aangeroepen. Aangezien er geen blokkeringen optreden in dit systeem voldoet het aan het eerste criterium en zou het kunnen gebruikt worden in een AmOP omgeving. Het grote probleem is het feit dat in E een netwerkfout fataal is voor een bepaalde referentie. Hierdoor kan men geen abstractie maken over tijdelijke verbroken verbindingen en voldoet het dus niet aan dit criterium voor foutafhandelingsmechanismen in een AmOP omgeving.

```
def promise := somePromise <- someMessage
when(promise) -> {
  //code uitgevoerd wanneer de promise resolved
} catch(e) {
  //wordt uitgevoerd wanneer de promise naar een
  //exception resolved
}
```

In dit voorbeeld zien we de syntax voor het when-catch systeem in E.

4.3.2 Except-clauses in Argus

Argus [10] [11] heeft ook een systeem waarin men gebruik kan maken van promises om aan failure handling te doen. Om de syntax en semantiek gemakkelijk duidelijk te maken, zullen we een klein code voorbeeld uitleggen.

```
pt = promise returns (real) signals (foo)
aPromise := someGuardian.someHandler()
y: resolvedValue := pt$claim(aPromise)
          except when foo: ...
               when unavailable(s: string): ...
               when failure(s: string): ...
          end
```

Omdat fouten in Argus na code blokken moeten geplaatst worden en deze dan fouten gaan opvangen die in deze blokken code worden gedetecteerd krijgen we een heel andere taal constructie dan die in E. De `claim` operatie zal een `promise` proberen resolven in een blokkerende vorm, dit wil zeggen dat de eerste regel zal geblokkeerd worden zolang de promise niet geresolved is. `pt` voor de `claim` operatie wil zeggen dat de promise van die soort is en waardoor de promise in het goede geval `real` teruggeeft en `foo` in het geval van een fout. Nadat de promise resolved is kan de rest van de code uitgevoerd worden. Doordat het resolven van een promise in een blokkerende vorm moet gebeuren is dit soort van fouten behandeling in strijd met het criterium voor niet-blokkerende behandeling van failures uit sectie 4.1. Let wel dat in Argus dit eigenlijk te wijten is aan het feit dat men geen specifieke taal constructie bezit voor niet-blokkerende behandeling van promises. Dit wil zeggen dat we in Argus niet kunnen specifiëren:"Wanneer de promise geresolved wordt voer dan deze code uit" (Er bestaat wel een ready primitieve die een boolean teruggeeft zodat dit

wel kan maar terug in een sequentile en blokkerende manier). Zoals men kan zien uit het voorbeeld zijn er twee soorten van exceptions mogelijk in het versturen van berichten over het netwerk in Argus.

- `unavailable` wil zeggen dat de fout van een tijdelijke aard is maar dat het niet aangeraden is onmiddelijk opnieuw dezelfde actie te proberen. Er bestaat dus een mogelijkheid dat de verbinding kan herstelt worden.
- `failure` wil zeggen dat de fout van een permanente aard is zodat het niet aan te raden is de actie opnieuw te proberen.

Hoewel men in Argus een onderscheid kan maken tussen tijdelijke en permanent verbroken verbindingen is het nog steeds zo dat in beide gevallen er een exception zal teruggegeven worden en is het dus aan de programmeur om deze exceptions naar behoren te behandelen. Hoewel deze eigenschap van Argus een goede stap is in de richting van het voldoen aan het criterium van abstractie over tijdelijk verbroken verbindingen, zorgt het gebruik van exceptions steeds voor problemen in de context van mobiele netwerken.

4.4 Observer-based exception handling

Het idee achter dit soort van exception handling is het registreren van failure handling procedures bij een distributie-eenheid. Deze failure handlers zullen dan opgeroepen worden indien er een failure werd gedetecteerd in de geviseerde distributie-eenheid. Dit systeem is event-driven zodat het voldoet aan het eerste criterium dat voorziet in het asynchroon omgaan met failures. Het voldoet ook in zekere zin aan het derde criterium in die zin dat de code die de exceptions afhandelt niet noodzakelijk gerelateerd is aan de code waarin de exceptions voorkomen. We hebben in hoofdstuk 2 twee voorbeelden van dit soort exception handling beschouwd en we zullen deze twee hier dan ook apart gaan evalueren.

4.4.1 ProActive

In ProActive kan men uit verschillende distributie eenheden kiezen om de failure handlers op te definiëren, met name:

- AO (Actief Object)
- Proxy (ofwel de client side van een actief object)
- JVM (Java Virtual Machine)
- Group

Het nadeel van al deze eenheden is het feit dat er heel weinig granulariteit mogelijk is omdat de eenheden nogal groot gekozen zijn. Hier volgt een code voorbeeld van het gebruik van de observer-based aanpak in ProActive. Deze observer zal op een actief object gedefinieerd zijn en zal triggeren wanneer er een niet functionele exception wordt opgemerkt. Niet functionele exceptions zijn vooral fouten met betrekking tot het gedistribueerde aspect (verbroken verbindingen).

```
ProActive.addNFEListenerOnAO(myAO, new NFEListener() {
    public boolean handleNFE(NonFunctionalException nfe) {
        // Do something with the exception...
        // Return true if we were able to handle it
        return true;
    }
});
```

4.4.2 E

In E zijn de referenties de distributie-eenheid waarop men de observers kan definiëren. Dit heeft in tegenstelling tot ProActive het voordeel van een fijne granulariteit te hebben. We zullen ook hier een voorbeeld tonen van het gebruik van de observer-based aanpak in E.

```
to addListener(newListener) {
        myListeners.push(newListener)
        newListener <- statusChanged(myStatus)
        def handler() { remove(myListeners, newListener) }
        newListener <- _whenBroken(handler)
}
```

In de code registreert men een listener op een referentie die ervoor zal zorgen dat wanneer de referentie naar een "broken" status overgaat de handler direct uitgevoerd wordt. In plaats van de _whenBroken kan men ook de _whenClientDisconnected om failure handling ook ten opzichte van de andere zijde van de connectie toe te laten.

4.5 Fatale failure handling

In hoofdstuk 2 hebben we drie soorten fatale failure handling besproken die vandaag beschikbaar zijn. Twee systemen hiervan vindt men terug in ProActive, het andere in de taal E. Zoals toen al werd uitgelegd zorgt deze fatale failure handling ervoor dat men van systeem crashes kan herstellen zodat hervatting van normale operatie na een crash mogelijk wordt gemaakt. Hoe bruikbaar deze methoden zijn in een AmOP omgeving zal hier besproken worden.

4.5.1 Fatale failure handling in ProActive

De twee fatale failure handling systemen in ProActive hebben beide een nood aan globale snapshots van het hele systeem om te kunnen functioneren. Dit is echter ondenkbaar in een AmOP omgeving omdat toestellen constant in beweging zijn en meestal niet in de buurt van anderen zullen zijn wanneer er een snapshot dient gemaakt te worden.

4.5.2 Fatale failure handling in E

Het systeem dat aanwezig is in E [14] om fatale failures aan te pakken is een goed voorbeeld van het individueel herstellen na een crash, i.e. toestellen dienen niet te synchroniseren om een backup te maken van hun toestand. Een

groot probleem met dit systeem is het feit dat er zeer moeilijk herstelling van connecties mogelijk is omdat E, zoals vele talen, ervan uitgaat dat connecties zeer zelden zullen verbreken. De enige mogelijkheid tot het herstellen van verbindingen na een volledige crash is door gebruik te maken van de zogenaamde offline capabilities.

4.6 Unavailable handlers

Unavailable handlers, cfr. Emerald, geven de programmeur de mogelijkheid om acties te ondernemen wanneer het systeem de locatie van een bepaald object als unavailable beschouwd. We zagen al eerder dat het gebruik van exceptions niet aan te raden is in een AmOP omgeving. Het is erg belangrijk om in een mobiel gedistribueerde omgeving de tijdelijke foutjes te kunnen negeren terwijl we fouten die een blijvend of significante invloed hebben op de applicatie wel te kunnen detecteren en behandelen. De mogelijkheid om een bepaald onderdeel van de omgeving als onbeschikbaar te verklaren en hierop handlers te kunnen uitvoeren is dus zeer handig in een AmOP omgeving. Een voorbeeld van dit soort afhandeling van fouten vinden we terug in Emerald [12] [13]. De regels die de implementatie gebruikt om een bepaald deel van de omgeving onbeschikbaar te verklaren zal de bruikbaarheid grondig beïnvloeden. Wanneer dit gebeurd direct na elke verbroken verbinding kan men het gaan vergelijken met exception handling en is de bruikbaarheid in AmOP gering. Het unavailable systeem wordt pas echt bruikbaar wanneer men een deel van de omgeving onbeschikbaar verklaart na een opeenvolging van fouten. In Emerald zal een bepaalde node in de omgeving unavailable beschouwd worden na een exception in de constructie ervan evenals wanneer alle andere nodes in het netwerk even geen connectie meer hebben met een bepaalde node. Het systeem in Emerald is dus niet bruikbaar in het AmOP paradigma.

4.7 Conclusie

In dit hoofdstuk hebben we de failure handling systemen geëvalueerd in de context van het AmOP paradigma. Een korte herhaling van de belangrijkste voor- en nadelen van deze systemen vindt men hier.

try-catch voldoet aan het criterium voor blokgestructureerde foutafhandeling maar faalt bij de twee andere criteria. Het is synchroon en heeft geen mogelijkheden om abstracties te maken over tijdelijk verbroken verbindingen.

when-catch voldeed aan het criterium dat stelt dat men asynchroon moet omgaan met failures, maar faalt bij de twee andere criteria. Men kan maar een when-catch definiëren voor één promise en geeft niet de mogelijkheid om abstracties te maken over tijdelijk verbroken verbindingen.

observer-based voldoet ook enkel aan het eerste criterium waardoor het wel asynchroon omgaat met failures. Naast deze eigenschap is het ook zeer modulair. Bij ProActive vonden we dan ook het nadeel dat we geen fijne granulariteit konden bekomen omdat de distributie-eenheden zo uitgebreid zijn.

De meeste van de besproken systemen zijn dus niet geschikt om in een mobiel gedistribueerde omgeving te gebruiken. We zullen hierdoor in het volgende hoofdstuk experimenten voorstellen die de hierboven aangehaalde voordelen van bestaande foutafhandelingsmechanismen zullen proberen te combineren om zo failure handling mechanismen te bekomen die wel nuttig zijn in een AmOP omgeving.

Hoofdstuk 5

Experimenten met taalconstructies voor foutenafhandeling in mobiele netwerken

De evaluatie van beschikbare failure handling heeft aangetoond dat er geen taalconstructies aanwezig zijn in de voorheen beschreven talen die voldoen aan alle criteria voor foutenafhandeling in mobiele netwerken die we hebben besproken in hoofdstuk 3. In dit hoofdstuk zullen we enkele nieuwe taalconstructies voorstellen die wel voldoen aan de criteria voor foutenafhandeling in het AmOP paradigma [3]. Deze taalconstructies werden geïmplementeerd in de AmOP taal AmbientTalk die we in hoofdstuk 2 al besproken hebben.

Due zal zorgen voor foutenafhandeling voor berichten die men wil versturen naar andere actoren.

Receive_due_otherwise kan gebruikt worden voor foutenafhandeling bij het ontvangen van berichten van andere actoren.

Reply_before zal men gebruiken om fouten af te handelen bij het versturen, verwerken en antwoorden op verstuurde berichten door de actor van bestemming.

In dit hoofdstuk worden elk van deze taalconstructies uitgelegd op vlak van syntax en semantiek. Naast de uitleg van de syntax en semantiek wordt ook het design en implementatie als taalmixin in AmbientTalk in detail uitgelegd. Voor we overgaan tot het gedetailleerd bespreken van de verschillende experimenten zullen ze eerst kort ingeleid worden door ze te gaan positioneren in de levenscyclus van een bericht in AmbientTalk.

5.1 Analyse van asynchrone berichtenversturing in AmbientTalk

Wanneer we naar de levenscyclus van een bericht in AmbientTalk kijken kunnen we de volgende stappen onderscheiden die een bericht aflegt via het MOP dat al besproken werd in hoofdstuk 3.

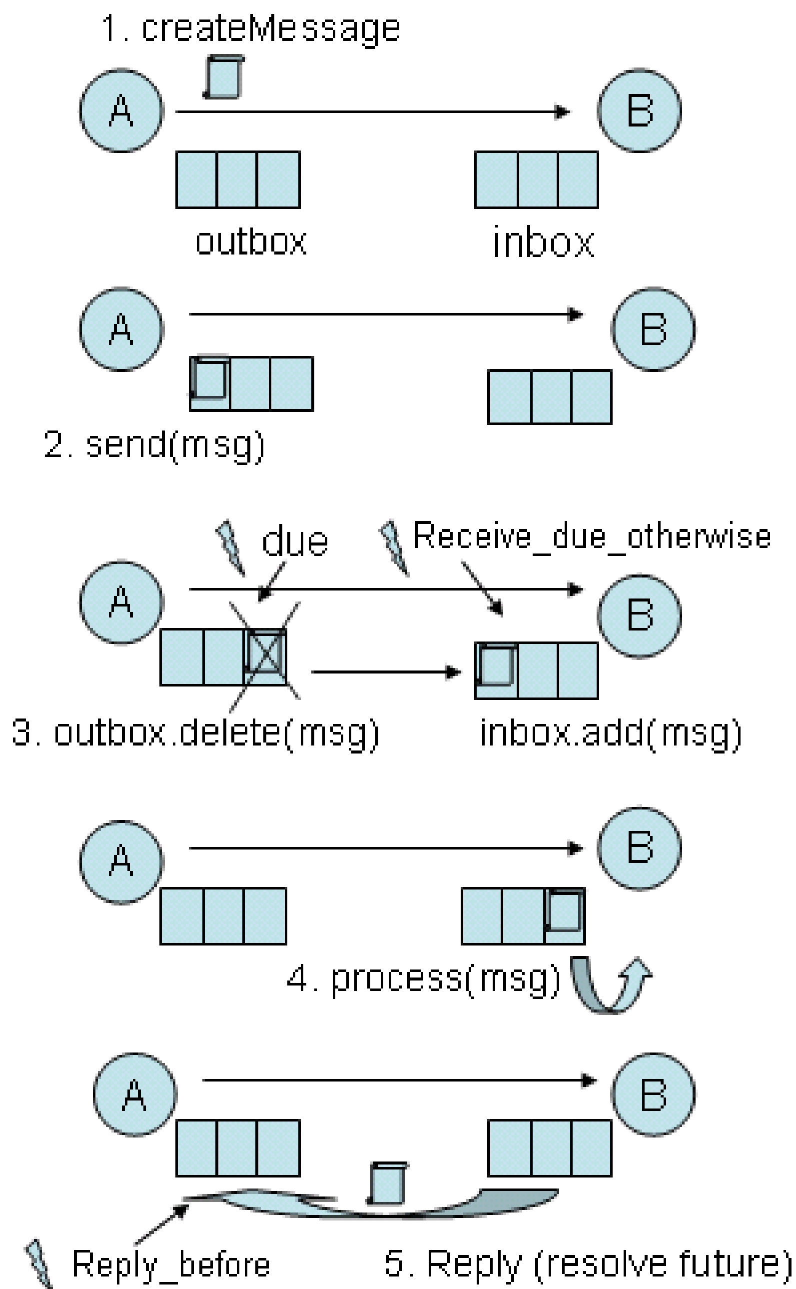
Creatie. De creatie fase van een bericht is een lokaal proces waarbij men enkel een MOP methode (createMessage) hoeft aan te roepen. Fouten kunnen dus in deze fase enkel het gevolg zijn van functionele fouten, wat niets te maken heeft met failure handling.

Zenden. Deze fase bestaat erin het aangemaakte bericht in de outbox van de actor te plaatsen zodat de VM van AmbientTalk het bericht kan gaan versturen over het netwerk. Omdat dit proces ook enkel uit een lokale bewerking bestaat hebben we geen failure handling nodig om fouten op te sporen en te behandelen.

Verzenden. Wanneer een bericht zich in de outbox van een actor bevindt zal AmbientTalk dit bericht proberen te versturen over het netwerk naar de actor van bestemming. Zoals we al in hoofstuk 2 hebben besproken verbreken verbindingen mogelijk zeer frequent in een AmOP taal, hiervoor hebben we een nieuwe taalconstructie gemaakt, `due` genaamd en wordt besproken in sectie 5.2. Deze failure handling zal er voor zorgen dat men in de taalconstructie kan bepalen wanneer men kan beginnen spreken over een failure en wanneer er dus een failure handler aangeroepen moet worden. Zoals al aangehaald in het vorige hoofdstuk zal het gebruik van een tijdslimiet een belangrijke rol spelen in het bepalen en identificeren van fouten. Omdat we hier enkel spreken over de "Versturen" fase zullen er enkel failures worden gedetecteerd op uitgaande berichten.

Ontvangen. Het ontvangen van berichten bestaat erin dat deze berichten opgevangen worden in de inbox van de actor van bestemming. Dit proces wordt ook beïnvloed door het netwerk zodat deze fase ook in aanmerking komt voor een failure handling taalconstructie. De `receive_due_otherwise` taalconstructie zal voor dit soort foutenafhandeling zorgen en wordt besproken in sectie 5.3. In tegenstelling tot de "Versturen" fase hebben we hier failure handling op toekomende berichten en dit is een heel nieuw gegeven in deze scriptie. Failure handling op binnenkomende berichten zullen we proberen te identificeren door het niet op tijd ontvangen van verwachte berichten. Het baseren op tijd vinden we hier dus al terug omdat we geen gebruik kunnen maken van exceptions.

Verwerken. Een bericht in de inbox zal door de actor van bestemming behandeld worden door gebruik te maken van de MOP methode `process`. Op zich is dit natuurlijk ook een lokaal iets maar wanneer men in de context van een mobiel netwerk gaat bekijken kan men ook hier failure handling op toepassen. Dit verwerken van een bericht is eigenlijk een oproep uit de actor van vertrek zodanig dat deze actor ook failure handling voor het verwerken van zijn berichten wil. De `reply_before` taalconstructie wordt gebruikt om de foutenafhandeling mogelijk te maken



Figuur 5.1: Levencyclus van een bericht in AmbientTalk.

in deze context en wordt besproken in sectie 5.4. Het behandelen van fouten in het verwerken van berichten kan bekomen worden door een tijdslimiet in te stellen waarbinnen een bepaald bericht moet verwerkt worden door de actor van bestemming. Merk ook op dat dit soort van failure handling veel gelijkenissen vertoond met zogenaamde futures. Futures zorgen ervoor dat men toch een return waarde krijgt bij een asynchroon bericht en vergelijkbaar met deze futures willen we in dit soort failure handling deze return waarde binnen een bepaalde tijdslimiet terugkrijgen.

Deze drie verschillende soorten van behandeling van fouten die hier kort werden beschreven zullen in dit hoofdstuk in detail besproken worden doormiddel van voorbeelden, code en mogelijke uitbreidingen in detail te bespreken.

5.2 De `due` taal constructie

De `due` taal constructie stelt de failure handling van de "Versturen" fase voor. De basis semantiek die gebruikt wordt in deze constructie bestaat uit het detecteren van fouten op uitgaande berichten door op basis van een bepaalde conditie te beslissen wanneer de verzending faalt, in dit geval wordt een limiet op de tijd die nodig is om een bericht te versturen over het netwerk gebruikt. Om deze semantiek iets concreter te beschouwen wordt deze uitgelegd met een klein voorbeeld. Naast een voorbeeld wordt het design en de implementatie in AmbientTalk in groot detail besproken, evenals verdere variaties die mogelijk zijn in het gebruik van deze taal constructie.

5.2.1 Syntax en semantiek

Om de syntax en de semantiek van de `due` taalconstructie uit te leggen zal gebruik gemaakt worden van een voorbeeld, namelijk het printer-gebruiker voorbeeld. De omgeving waarin we zullen werken bestaat uit twee actoren: een printeractor en een gebruikersactor. De printeractor biedt een print service aan andere actoren aan die zich in het zendgebied bevinden van de printer zelf, dit wordt in AmbientTalk geïmplementeerd door een "printer" string te plaatsen in de providedbox van de printeractor. De gebruikersactor in het voorbeeld zal aan zijn kant een "printer" string in zijn requiredbox plaatsen zodat andere actoren weten dat deze gebruiker op zoek is naar een printer. Wanneer deze twee nu in elkaars buurt komen zal er een connectie ontstaan en zal de gebruiker een zogenaamde netwerkreferentie (Eng: remote reference) krijgen naar de printer waarmee hij berichten kan versturen naar de printer. Voor we echter een heel voorbeeld kunnen uitleggen zal eerst de basis syntax beschreven worden door gebruit te maken van de volgende abstracte code.

```
due( < vervalconditie >,
     { < code > },
     expired( < bericht filter >, [ < handler >, ... ]),
     ...
     )
```

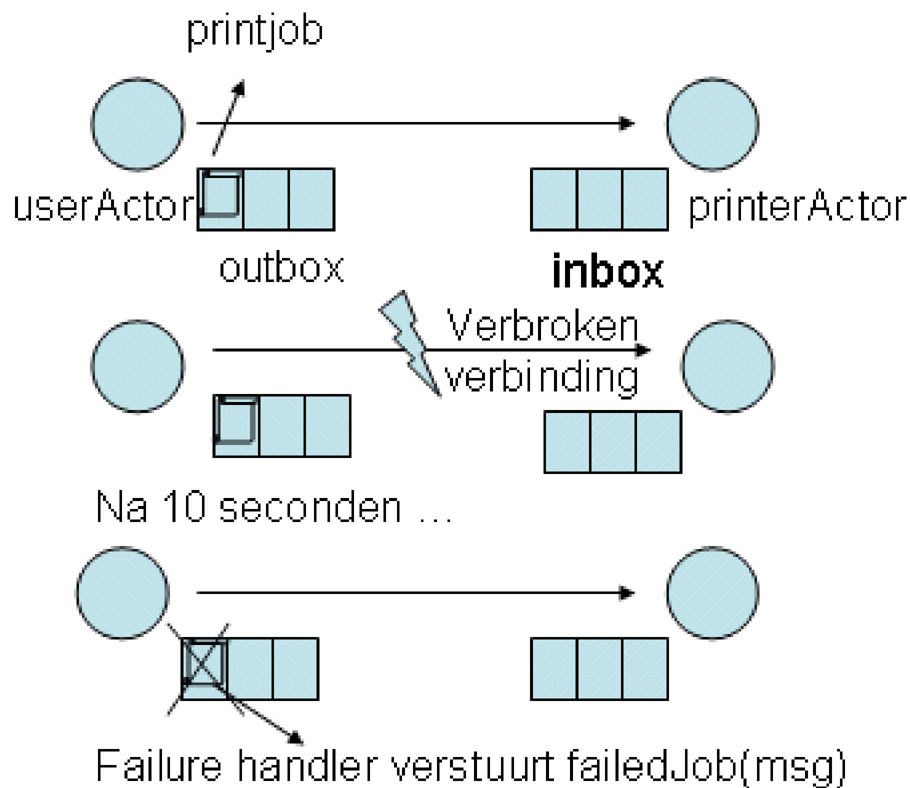
De eerste parameter wordt gebruikt om de vervalconditie aan te duiden waarmee de verzonden berichten moeten uitgerust worden om het mogelijk te maken

de gewenste fouten op te sporen. Alle berichten die men wil uitrusten met de vervalcondities vermeldt in het eerste parameter slot plaatst men in de dynamische context van een blok code in het tweede parameter slot. Na deze twee parameters kan men een onbeperkt aantal `expired` functies specifiëren, deze bestaan uit een bericht filter en een lijst van handlers. De bericht filter moet een boolean teruggeven die zal bepalen of de handlers al dan niet uitgevoerd moeten worden. De handler bestaan uit paren van actoren en berichten die moeten verzonden worden naar deze actoren indien de bericht filter true terug geeft.
Nu de globale syntax beschreven werd kunnen we beginnen aan de bespreking van een voorbeeld in AmbientTalk dat al eerder werd ingeleid, namelijk het printer voorbeeld.

```
gebruikersActor :: actor(object({
  printer : void;
  init() :: {
    .init();
    required.add("printer");
    joinBox.addAddObserver(thisActor()#joined)
  };
  joined(printerReferentie) :: {
    printer : provider(printerReferentie);
  };
  print(lijstVanTaken) :: {
    if(!is_void(printer),
       due(timeout(10*1000),
           { printTaken(lijstVanTaken) },
           expired(and(msg.getName() = "print", msg.getArgs()[1].dringend()),
             [thisActor()#dringendBerichtGefaald]),
           expired(msg.getName() = "print", [thisActor()#berichtGefaald])))
  };
  printTaken(lijstVanTaken) :: {
    for(i:1, i < size(lijstVanTaken), i := i+1,
        { printer#print(lijstVanTaken[i]) })
  };
  dringendBerichtGefaald(msg) :: {
    'Vertel de gebruiker dat zijn printtaak gefaald is'
  };
  berichtGefaald(msg) :: {
    'Plaats de printtaak die is gefaald in de print wachtrij
     tot er een andere printer is gevonden'
  }
}

printerActor :: actor(object({
  init() :: {
    provided.add("printer")
  };
  print(PrintTaak) :: {
    'Druk de PrintTaak af'
```

```
  }
}
```

De `gebruikersActor` maakt gebruikt van een variabele `printer` om te kunnen bepalen of er een printer in de buurt is of niet, indien dit het geval is zal de referentie naar de printer opgeslagen zitten in deze variabele. In de `init` methode zet men een "printer" string in de requiredBox zoals al eerder aangegeven in de inleiding van deze sectie. Er wordt ook een observer geplaatst op de joinBox zodat de methode `joined` wordt opgeroepen wanneer er contact gelegd wordt met een printer. Bij een join wordt dan de `printer` variabele gezet op de referentie naar de juist gecontacteerde printer. De `print` methode zorgt ervoor dat er taken kunnen verstuurt worden naar de printer wanneer er een geconnecteerd is met de `gebruikersActor`. Nu de hulpcode uitgelegd is kunnen we overgaan tot het bespreken van de `due` taalconstructie die aanwezig is in de `print` methode.
De eerste parameter van de `due`, `timeout(10*1000)`, stelt de regel voor die we gaan hanteren om te beslissen wanneer het verzenden van een bericht faalt. Een failure in dit voorbeeld zal gedetecteerd worden wanneer een bepaald bericht niet binnen tien seconden is verstuurd naar zijn bestemming (timeout verwacht de tijdslimiet in milliseconden). De berichten die aan deze tijdslimiet moeten voldoen zijn alle asynchrone berichten die worden aangemaakt tijdens de uitvoering van het blok code in het tweede parameter slot. Het feit dat er gebruik gemaakt wordt van een functie om de timeout weer te geven biedt mogelijkheden om andere regels te implementeren om een bepaald bericht als "vervallen" te kunnen beoordelen. Hoewel we hier enkel zullen spreken over de tijdsgebaseerde aanpak zal er later in dit hoofdstuk nog een bespreking volgen over andere mogelijkheden om berichten te laten falen zonder gebruik te maken van tijd.
In het geval van het printer voorbeeld zullen de `print` berichten die gespecifieerd zijn in de hulpfunctie `printTaken` binnen de tien seconden moeten verzonden worden, anders zal er een gepaste expired handler uitgevoerd worden die men in de volgende parameter slots terugvindt. Het is niet nodig de asynchrone berichten expliciet in het tweede parameter slot te plaatsen, alle verzonden berichten in de dynamische omgeving van het blok code zullen berichten zijn waarbij de verzending kan falen. Naast het oproepen van een handler zal er ook een verwijdering uit de outbox van de actor plaatsvinden van het bericht waarbij de verzending gefaald is. Het verwijderen van gefaalde berichten kan beschouwd worden als een manier om aan garbage collecting te doen in de outbox van een actor zodat het niet meer mogelijk wordt dat berichten oneindig lang in een actor zijn outbox gaan blijven zitten. Er zijn twee failure handlers gedefinieerd in het code voorbeeld, dit houdt in dat we een systeem nodig hebben om te bepalen welke handler nu eigenlijk uitgevoerd moet worden. De eerste parameter van de `expired` functie is een stuk code die een boolean terug moet geven die zal bepalen of deze handler wel of niet zal uitgevoerd worden mocht het bericht `msg` (een call-by-function parameter van de expired functie) vervallen. In het voorbeeld maakt men een onderscheid tussen de behandeling van print berichten die dringend zijn en gewone print berichten. De handlers die kunnen opgeroepen worden bij elk van deze gevallen hebben ook elk een `msg` parameter, deze zal altijd gezet zijn op het bericht waarbij de verzending gefaald is. De `msg` variabele



Figuur 5.2: Grafische voorstelling van het printer-voorbeeld. In de eerste fase wordt er een print bericht naar de printerActor gestuurd zodanig dat dit bericht in de outbox van de gebruikersActor terecht komt. Wanneer dan de verbinding tussen deze twee actoren verbreekt zal het print bericht in de outbox blijven zitten tot er eventueel een herstellen optreedt van de verbinding. Met de beschreven failure handling op het print bericht zal het na een timeout waarde verwijderd worden uit de outbox van de actor van oorsprong.

die men kan gebruiken in de `expired` functie en in de failure handlers geeft ook de mogelijkheid alle informatie op te vragen die het bericht bevat waaronder de oorsprong (de actor die het bericht verzond), bestemming (actor naar wie het bericht wordt verstuurd), naam en parameters. In het voorbeeld maakt men dan ook gebruikt van deze informatie om een onderscheid te kunnen maken tussen de foutenafhandeling van de verschillende soorten printtaken. Wanneer men echter enkel een true boolean in de bericht filter zou plaatsen zou men als het ware een "catch-all" verkrijgen, wat misschien in sommige gevallen handig kan zijn. Uit dit laatste blijkt dat deze taal constructie veel gelijkenissen vertoond met de try-catch die we al eerder in hoofdstukken 2 en 3 ten gronde hebben bestudeerd. De tweede parameter in de `expired` functie kan men gebruiken om een handler op te roepen. Het oproepen van een handler moet gebeuren door asynchrone berichten te sturen naar bepaalde actoren, deze berichten zullen dan de rol van failure handlers op zich nemen en uitgevoerd worden. Het behandelen van failures moet niet noodzakelijk in de actor waarbinnen de `due` code werd uitgevoerd gebeuren.

Nesten van due-blocks Nu we de simpele semantiek hebben besproken zal een volgend voorbeeld de meer geavanceerde semantiek duidelijk maken. Deze meer diepgaande semantiek slaat op het feit dat men in veel gevallen de `due` zal nesten waardoor we hier de vergelijking met een try-catch systeem terug gaan kunnen gebruiken. We zullen als voorbeeld de `due` uit het vorige voorbeeld uitbreiden met een geneste `due` om zo de nieuwe semantiek uit te leggen.

```
due(timeout(10*1000),
    { due(timeout(20*1000),
          { printTaken(lijstVanTaken) },
          expired(and(msg.getName() = "print", msg.getArgs()[1].dringend()),
            [thisActor()#dringendBerichtGefaald]))
     },
     expired(msg.getName() = "print", [thisActor()#berichtGefaald]))
```

Bij het nesten van due-blocks is het heel belangrijk te weten dat het toekennen van foutenafhandelaars en vervalcondities gebeurd voor de eigenlijke verzending van het bericht. Wanneer een printtaak zich dus in de outbox van een de `gebruikersActor` bevindt zal deze al uitgerust zijn met een specifieke failure handler een daarbij horende vervalconditie. Het selecteren van de vervalconditie hangt dus helemaal af van de bericht filter in het eerste parameter slot van de `expired` functies. Dit heeft zware gevolgen voor de semantiek bij het nesten van due-blocks, om dit aan te tonen zullen we het voorbeeld erbij halen. Wanneer een dringend bericht verzonden wordt uit de functie `printTaken` zal deze uitgerust worden met een deadline van 20 seconden omdat de failure hander uit die due-block de geschikte is. Wanneer er echter een gewoon `print` bericht verzonden wordt uit de `printTaken` functie in de diepste due-block dan zal het zoeken naar de juiste handler een nesting hoger moeten consulteren om de juiste te vinden. Omdat het de failure handler gebruikt uit de hogere nesting zal de vervalconditie van het normale bericht 10 seconden zijn in plaats van 20. Zoals de blokken code in de `due` wordt de nesting van due-blocks dynamisch bepaald, wat ook duidelijk wordt uit het gegeven voorbeeld. Al deze eigenschappen kan men ook al terugvinden in de bekende try-catch constructies uit populaire talen

als Java. Een belangrijk verschil echter is het feit dat de zoektocht naar een geschikte handler stopt bij de hoogste nesting. Het falen van een verzending zal dus genegeerd worden wanneer er in geen enkele nesting een geschikte handler gevonden wordt.
Nu de syntax en de semantiek van de due taal constructie is uitgelegd zullen we verdergaan met het bespreken van het design ervan in het MOP van AmbientTalk. Dit houdt in dat we vanaf nu zullen spreken over de mixins omdat we gebruik gaan maken van mixins om deze andere taal constructies te implementeren.

5.2.2 Het design

Om de `due` mogelijk te maken wordt er gebruik gemaakt van mixins in AmbientTalk omdat deze een modulair gebruik van nieuwe taal constructies mogelijk maken binnenin AmbientTalk. Omdat het echter mogelijk is dat een actor waarin er asynchrone verzendingen worden uitgevoerd niet diegene is die de berichten werkelijk zal verzenden over het netwerk, werd de due opgesplitst in twee taal mixins. Een voorbeeld van dit gebruik vinden we terug bij ambient references en futures [3]. Hierbij maakt men gebruik van een soort proxy actor waarnaar men alle asynchrone verzendingen doorstuurt waarna deze proxy de berichten effectief zal verzenden naar een gedistribueerde actor. Elke mixin heeft zijn eigen taak en ze zullen dan ook correct toegepast moeten worden om `due` te kunnen gebruiken.

- De **DueMixin** definieert de taalconstructie die men kan gebruiken in code om de failure handling mogelijk te maken. Deze mixin zal dus elk asynchroon bericht dat verzonden wordt vanuit een blok code binnen een `due` voorzien van de nodige informatie om foutendetectie op deze berichten mogelijk te maken.
- De **ExpiryCheckMixin** zorgt ervoor dat de outbox van de actor waarop deze mixin is toegepast continue afgelopen wordt om te kijken of er ten eerste berichten inzitten die enigszins kunnen falen en ten tweede of deze dan wel degelijk nog niet gefaald hebben.

Een belangrijk aspect van het designen van een mixin in AmbientTalk is het identificeren van de verschillende MOP methoden die zullen overschreven of verfijnd worden door de mixin. Om geen inconsistenties te veroorzaken in het gebruik van meerdere mixins zullen we hier een overzicht geven van de berichten die de twee mixins overschrijven. Door deze twee mixins samen te gebruiken kan men er dus voor zorgen dat fouten kunnen gedetecteerd en afgehandeld worden in uitgaande berichtgeving zonder gebruik te hoeven maken van exceptions. Indien een actor indirect wil communiceren met gedistribueerde actoren via bijvoorbeeld ambient reference proxies, dan dient de DueMixin op deze actor toegepast te worden en de ExpiryCheckMixin op de ambient reference proxy.

5.2.3 Implementatie

In deze sectie zal de implementatie van de twee taal mixins in detail uitgelegd worden. De uitleg zal ingedeeld worden in drie aparte secties: DueMixin, ExpiryCheckMixin en speciale toevoegingen aan berichten.

MOP methode	DueMixin	ExpiryCheckMixin
init		verfijnt
createMessage	verfijnt	
send	verfijnt	
process		

Tabel 5.1: Een overzicht van welke MOP methoden de DueMixin en ExpiryCheckMixin overschrijven.

DueMixin

De DueMixin code omvat de `due` taal constructie zelf, samen met de opslag van alle handlers die kunnen gebruikt worden in de failure handling. Alle onderdelen van de DueMixin zullen hier aan bod komen, waarbij alle hulpstukken uit de doeken gedaan zullen worden, waaronder de belangrijke `timeout` en `expired` functies.
Voor we beginnen aan de gedetailleerde beschrijving van de implementatie zullen we eerst een globaal overzicht geven van de code in AmbientTalk. Van elke deel zullen we dan zijn globale functie specifiëren.

```
DueMixin(gedrag) :: {
  ‘hulp object‘
  myStack :: object({ ... });

  ‘variabelen‘
  handlerStack: myStack.new();
  inDueContext : false;

  ‘publieke interface voor de taalconstructie‘
  timeout(t)::{ ... };
  expired(cond(msg), [foutenafhandelendeBerichten]) :: { ... };
  due@args() :: { ... };

  ‘Verfijnde methoden van het MOP in AmbientTalk‘
  createMessage@args :: { ... };
  send(msg) :: { ... };

  capture()
}
```

Het hulp object `myStack` wordt gebruikt om de delegatie van de handlers evenals de opslag ervan mogelijk te maken. De twee instance variabelen die in de DueMixin worden gebruikt zijn de stack die we hierjuist al hebben besproken en een boolean variabele. Deze boolean variabele wordt gebruikt in de overschreven MOP methoden om een onderscheid te kunnen maken tussen berichten die al dan niet in de dynamische context van een due-block werden aangemaakt. Het gebruik van deze boolean zal duidelijker worden bij de bespreking van de MOP methode die in deze mixin overschreven worden. Door middel van de `timeout`

functie zal man aan berichten een vervalconditie kunnen meegeven die zegt dat als een bericht niet binnen tijd `t` verstuurt is een fout is opgetreden. De `expired` functie op zijn beurt zal dan de foutenafhandelaars definiëren die moeten aanroepen worden indien er een fout optreedt. De taalconstructie zelf wordt dan weer gedefinieerd in de `due` functie met zijn arbitrair aantal argumenten. Hierna worden er nog twee MOP methoden verfijnd om het gedrag van het verzenden van berichten te kunnen manipuleren wanneer men in de dynamische scope werkt van een due-block. De twee MOP methoden `createMessage` en `send` die ervoor instaan dat berichten goed verstuurd worden zullen dus verfijnd worden. Om te beginnen zullen we de belangrijkste eigenschappen van het stack object bespreken.
Een vereenvoudiging van de code van de stack die we zullen gebruiken in de opslag van de handlers en vervalcondities bij het uitvoeren van due-blocks.

```
myStack::object({
    top: void;
    stack: void;
    cloning.new()::{ ... };
    reset()::{ ... };
    topOfStack()::{ ... };
    isEmpty()::{ ... };
    push(element)::{ ... };
    pop()::{ ... };
   getNext()::{ ... }
});
```

Het `Stack` object bestaat uit de standaard `pop` en `push` operaties. Een belangrijke operatie is de `getNext` omdat we deze zullen gebruiken om dieper in de stack te zoeken wanneer er niet direct een geschikte handler gevonden wordt. Bij elke nieuwe nesting van een `due` functie zal er een nieuw object worden aangemaakt dat op de stack geplaatst wordt. In elk object op de stack worden alle handlers opgeslagen die op dat niveau in de nesting gedefinieerd werden, elk niveau bestaat uit een tabel van enerzijds alle handlers en anderzijds de bijbehorende vervalconditie functie. Bij het versturen van een `due` bericht zal er een opzoeking gebeuren vanaf de `top` van de stack helemaal tot onderaan tot er een goede failure handler gevonden wordt. We zullen nu de verschillende functies uit de publieke interface in detail bespreken.
De `timeout` functie is verantwoordelijk voor het toevoegen van een deadline aan een bericht.

```
timeout(t) :: {
      lambda(msg) :: {
        msg.addAttachment("deadline", t + time());
        msg.addExpiryCond(time() > deadline)
      }
    };
```

De timeout functie geeft allereerst een functie terug geparameteriseerd met een bericht. Deze functie stuurt twee berichten naar de `msg` parameter, welke een bericht voorstelt.

- `addAttachment` alloceert een nieuwe variabele in de omgeving van het bericht. De eerste parameter geeft de naam van de variabele en de

tweede parameter stelt de inhoud voor van de variabele. In het geval van timeout zal `addAttachement` ervoor zorgen dat de tijd in een bericht wordt opgeslaan dat zal gebruikt worden om het moment te berekenen waarop het bericht zal vervallen.

- `addExpiryCond` stelt de conditie in die moet beoordelen of een bericht al dan niet als vervallen moet aanzien worden. Deze conditie moet altijd een boolean teruggeven, in dit voorbeeld mag de huidige tijd `time()` niet groter zijn dan de deadline variabele die we hier boven al hebben beschreven. Het is dus mogelijk om in deze conditie variabelen te gebruiken die door middel van het `addAttachment` bericht eerder aan het bericht werden toegevoegd.

In het voorbeeld werd al aangegeven dat deze timeout functie gebruikt wordt in de eerste parameter van de due taal constructie. De lambda functie zal gebruikt worden om extra informatie in de due-berichten te stoppen zodat de ExpiryCheckMixin kan beslissen wanneer een bericht vervallen is.
De taak van de `expired` functie is het groeperen van handler filters en de corresponderende handlers. Een handler is een paar bestaande uit een failure handler actor en een bericht, ook wel de ”complaint”genaamd.

```
expired(cond(msg), handlerMessages)::{
      if(not(handlerMessages = []),
        { i : 0;
          complaints[size(handlerMessages)]: {i:=i+1;
                                                [handlerMessages[i].getTarget(),
                                                 handlerMessages[i].getName()] };
          [cond,complaints] },
        [cond, handlerMessages])
    };
```

Terwijl de `timeout` functie bedoelt is om informatie aan berichten toe te voegen wordt de `expired` functie eerder gebruikt als syntactische suiker om gemakkelijk handlers toe te voegen. De eerste parameter van de `expired` functie is een functionale parameter die zelf geparameteriseerd is met een bericht dat aangemaakt werd binnen een due-block. Omdat het een functionele parameter is zal berichten filter niet onmiddelijk geëvalueerd worden. De tweede parameter bestaat uit een tabel met allemaal actor#bericht koppels in die moeten opgeroepen worden indien deze handler een failure moet behandelen. Wanneer men in AmbientTalk actor#bericht als een stuk code gaat evalueren bekomt men het bericht zelf als return waarde, waarvan we hier gebruik gaan maken om een gebruiksvriendelijke syntax te bekomen. Uit deze berichten wordt dan het doel en de naam van het bericht gehaald aangezien dit de enige twee nuttige parameters voor failure handling zullen zijn.
`Due` stelt de due taalconstructie zelf voor en activeert het due-gedrag voor berichten aangemaakt in het code block van `due`.

```
due@args()::{
      i : 0;
      expirationCondition:args[1]();
      msgHandlers[size(args) - 2]: { i:=i+1; args[2+i]() };
      handlerStack.push([msgHandlers,expirationCondition]);
```

```
      oldInDueContext : inDueContext;
      inDueContext := true;
      args[2]();
      handlerStack.pop();
      inDueContext := oldInDueContext
    };
```

De `args` parameter bestaat uit een tabel van alle parameters die meegegeven werden met de `due` taalconstructie, deze werden al uitvoering besproken in sectie 5.2.1. De `@` notatie tussen `due` en `args` zorgt ervoor dat er een arbitrair aantal handlers kan meegegeven worden bij de `due`. Een ander belangrijk feit om hier op te merken is de `args()` notatie, deze zorgt ervoor dat geen enkel element uit de `args` tabel geëvalueerd zal worden. Dit is nodig omdat we niet willen dat de code meegegeven in het tweede parameter slot onmiddelijk uitgevoerd zou worden. We zullen deze bespreking nu verder zetten met een top-down bespreking van de code van `due`. Het eerste wat ondernomen wordt bij het uitvoeren van de `due` is het uitvoeren van het eerste parameter slot, deze zal de vervalconditie opleveren en opgeslagen worden in de `expirationCondition` variabele. Hierna worden alle handlers die gedefinieerd zijn vanaf het derde paramater slot uitgevoerd en opgeslagen in de `msgHandlers` variabele. De handlers en de vervalconditie worden dan samen op de stack gepusht. De boolean die aangeeft of er een `due` blok code uitgevoerd wordt of niet wordt op `true` gezet waarna met de regel `args[2]()` de code wordt uitgevoerd die tegen failures beschermd moet worden. Na het uitvoeren van de code worden de handlers van het huidige due-block van de stack gepopt. `oldInDueContext` zal ervoor zorgen dat de `inDueContext` goed zal gezet worden wanneer we te doen krijgen met het nesten van `due` blocks.
Nu we de functies uit de publieke interface hebben besproken kunnen we overgaan tot het beschrijven van de MOP berichten die moeten verfijnd worden om de implementatie van `due` mogelijk te maken. Zoals we al zagen in het design overschrijft de DueMixin de `send` en de `createMessage` MOP methoden.
Wanneer er een bericht aangemaakt wordt in de context van een due-block dan dient dit bericht voorzien te worden van de juiste handler en vervalconditie, hiervoor wordt de `createMessage` verfijnd.

```
createMessage@args :: {
   origmsg: .createMessage@args;
   if(inDueContext,
      { result : lookuphandler(origmsg);
        if(not(result.found),
          origmsg,
          { origmsg.addAttachment("handlers", result.handlers);
            origmsg.addAttachment("expirationCondition", result.expirationCondition);
            origmsg.extend({
              expConditions : vector.new();
              isExpired() :: {
                b : false;
                expConditions.iterate(b := (el() | b) );
                b
              };
              dynamic_scope(bool) :: { bool[1] };
              addExpiryCond(bool()) :: { expConditions.add(dynamic_scope(bool) };
```

```
              });
              origmsg
            })
        });
    origmsg
};
```

De `createMessage` wordt als eerste opgeroepen wanneer AmbientTalk een stuk code tegenkomt waarin een actor een bericht naar een andere actor wil versturen. Omdat we nieuw gedrag nodig hebben voor de berichten verzonden in de context van een due-block, is het nodig om in de fase van het creëren van berichten code in te voegen om dit nieuw gedrag in te voegen. De eerste `if` maakt gebruik van de eerder besproken boolean `inDueContext` om te bepalen of er due-gedrag moet toegevoegd worden aan het bericht of het bericht op een normale manier moet afgehandeld worden. De top van de stack zal op het moment van een bericht verzonden in de context van een due-block bestaan uit de handlers die gedefinieerd werden samen de lambda functie die de vervalconditie voorstelt. De opzoeking naar de juiste handler om aan een bericht te koppelen wordt verwezenlijkt door de hulpfunctie `lookuphandler`. De functie `lookuphandler` neemt het bericht die een handler zoekt als argument en geeft een `result` object terug met drie slots: `found` (of er überhaupt een handler gevonden werd), `handlers` (indien er een filter van een reeks handlers overeenkwam met het bericht kan men deze handlers uit dit slot halen)en `expirationCondition` (gebruikt om een vervalconditie toe te voegen aan een bericht). Let dus op dat de toewijzing van een handler <u>voor</u> de eigenlijke verzending zal plaats vinden. De handlers, evenals de vervalconditie, worden aan het bericht toegevoegd door gebruik te maken van het `addAttachment` bericht. De `addAttachment` methode voegt een constant slot toe aan een bericht.
Een zogenaamd due-bericht heeft een opslagplaats voor alle failure-condities die zijn aangebracht aan het bericht nodig. Hierdoor wordt het bericht verrijkt met gedrag om aan dit soort failure handling te kunnen doen. Wanneer de `isExpired` methode wordt aangeroepen zullen deze condities allemaal getest worden met de logische or tussen de verschillende boolean return waardes. Dit wil zeggen dat één vervalconditie al genoeg is om een bericht te doen vervallen. De `addExpiryCond` methode die we al eerder zagen bij de `timeout` functie wordt gebruikt om condities toe te voegen aan het bericht. `bool[1]` geeft enkel de functie weer van de closure die men binnenkrijgt als parameter bij de `addExpiryCond` methode. Dit is nodig omdat berichten gekopiëerd worden tussen actoren en op dat moment in principe niet meer aan de omgeving van hun originele actor kunnen. Dit loskoppelen is noodzakelijk zodat de conditie functies goed kunnen uitgevoerd worden door het bericht zelf in de actor waarop de ExpiryCheckMixin geïnstalleerd is.
Het tweede `if` statement zorgt ervoor dat wanneer er geen geschikte handler gevonden wordt voor een due-bericht het zal verstuurd worden als een gewoon bericht. Dit voorkomt overtollige informatie in berichten die helemaal niet nodig zijn. Wanneer er wel een handler wordt gevonden zal de `origmsg` verrijkt worden met due-gedrag en wordt het bericht als return waarde teruggegeven.
De `send` zorgt ervoor dat voor elk bericht aangemaakt in de context van een due-block de vervalconditie van het bericht geactiveerd zal worden. Concreet betekent dit dat het berekenen van de timeout van het bericht.

```
send(msg) ::{
      if(msg.containsBehaviour("expirationCondition"),
          { msg.expirationCondition(msg) });
      .send(msg)
    };
```

Met het huidige MOP systeem van AmbientTalk is het mogelijk om berichten aan te maken, deze op te slaan en pas later te gaan verzenden naar de uiteindelijke bestemming. Hierdoor moeten we de `send` gaan overschrijven zodat we bij het eigenlijke verzenden de lambda uit de timeout functie op het juiste moment kunnen uitvoeren. Dit is nodig omdat de timeout gespecifieerd bij een due-block relatief is ten opzichte van de tijd dat een bericht *verzonden* wordt, niet ten opzichte van de tijd dat het bericht *aangemaakt* wordt.
Wanneer men een bericht binnen een `due` aanmaakt wil men natuurlijk dat het bericht wel degelijk een due-bericht is. De eerste `if` in de `send` is dan ook een test of het bericht dat wordt verstuurd al dan niet een due-bericht is. Het testen of een bericht een due-bericht is gebeurd door gebruik te maken van de `containsBehaviour` methode die een boolean teruggeeft wanneer de meegegeven naam van een bericht overeenstemt met een gedefinieerd slot in het geviseerde object. Wanneer men zeker weet dat dit bericht due-gedrag bevat zal de lambda functie die werd gevonden in de `createMessage` uitgevoerd zodanig dat de vervalconditie geactiveerd zal worden in het bericht. Hiervoor voeren we gewoon het `expirationCondition` slot uit van het bericht.

ExpiryCheckMixin

Deze mixin wordt gebruikt om vervallen berichten in de outbox op te sporen en af te handelen. De implementatie van de ExpiryCheckMixin bestaat uit een `notify` methode en maakt ook gebruik van een speciaal soort actor die in deze sectie zal besproken worden.
De delay die nodig is bij het aanroepen van deze mixin zal gebruikt worden door een zogenaamde `ticker` actor. Eens een bepaalde actor zich geregistreerd heeft bij een `ticker` actor dan zal deze actor continu een `notify` bericht krijgen met de afgesproken delay waarde als interval periode. De actor dient de `notify` methode te implementeren met de acties die telkens na elke delay uitgevoerd moeten worden. In het geval van de ExpiryCheckMixin zal de outbox na elke delay afgelopen worden op zoek naar berichten die "expired" zijn.

```
expiryCheckMixin(aDelay)::{
    myTicker : ticker(aDelay);

    ‘invoked by myTicker every aDelay millisec‘
    notify()::{
      outbox.asVector().iterate(
        if(el.containsBehaviour("isExpired"),
           { if(el.isExpired(),
                { outbox.delete(el);
                  i : 1;
                  while(i <= size(el.handlers),
                    { send(  createMessage(thisActor(),
                             el.handlers[i][1],
```

```
                                el.handlers[i][2],
                                [el]));
                          i := i +1 })
                    })
              }))
        };

        init() :: {
          .init();
          myTicker#subscribe(thisActor())
        };

        capture()
      }
```

De `init` MOP methode wordt hier overschreven om het mogelijk te maken de actor waarop deze mixin wordt geïnstalleerd in te schrijven bij een `Notifier` zodanig dat de `notify` operatie na elke delay in milliseconden uitgevoerd wordt. Het overlopen van de outbox wordt mogelijk gemaakt door de outbox als een vector te beschouwen en daarna over al zijn elementen te itereren. Elk bericht `el` in de outbox wordt dan onderzocht of het een due-bericht is, en wanneer dit zo is zal er een check plaatsvinden om te testen of het bericht al dan niet vervallen is. Dit gebeurd door de `isExpired` methode op het bericht op te roepen. Wanneer een gefaald bericht ontdekt wordt zullen de failure handling berichten verstuurd worden en wordt dit bericht uit de outbox verwijderd. We zagen al eerder dat failure handlers uit een tabel van paren van een actor en een naam van een bericht bestaan.

5.2.4 Niet-tijdsgebaseerde vervalcondities

In de vorige secties zagen we al dat het gebruik van generische vervalcondities zoals de `timeout` functie het mogelijk maakt verschillende condities aan berichten toe te voegen. In deze sectie zullen verschillende alternatieven voorgesteld worden op de `timeout` functie dewelke ook aantonen dat `due` kan gebruikt worden om failure handling te doen op basis van condities die niet op tijd zijn gebaseerd. Een interessant gegeven is het feit dat men enkel de functie zelf hoeft aan te passen om een heel ander gedrag te bekomen voor de `due` taal constructie. De verschillende alternatieven die we in dit hoofdstuk zullen bespreken zijn deze.

Replace zal ervoor zorgen dat het niet mogelijk is meerdere berichten met dezelfde naam tegelijk in de outbox te hebben, het nieuwste bericht zal telkens behouden blijven terwijl de oudere verwijderd worden.

OnMessage geeft het bericht een naam mee van een mogelijk ander bericht dat zich in de outbox kan bevinden. De aanwezigheid van een bericht met die specifieke naam heeft tot gevolg dat het bericht zichzelf zal beschouwen als vervallen.

Combine biedt de mogelijk aan om eerder geziene vervalconditie functies zoals `timeout` en `replace` samen te gaan gebruiken in een `due` code blok.

Replace

Een `due` met de `replace` functie kan mogelijk gebruikt worden bij actoren die op regelmatige tijdstippen berichten naar een andere actor moeten sturen en waarbij men geen opstapeling van meerdere berichten wil wanneer de connectie tussen de actoren voor een bepaalde periode onderbroken is. Dit zullen we aantonen door gebruik te maken van een klein voorbeeld van het gebruikt van dit soort vervalcondities. Het voorbeeld stelt een systeem voor om naar een bepaald server object continu update berichtjes te sturen zodat de gebruiker zijn status ten alle tijde bekend is bij de server. Om het voorbeeld zo simpel mogelijk te houden zullen we enkel de `due` aanroep bespreken.

```
due( replace(),
     { ‘code die de status van een gebruiker bepaald‘
       serverActor#update(status);
       ‘herstart de bovenstaande code in een loop‘
     })
```

Omdat het niet nodig is om een handler te definiëren zullen we het hier ook niet doen. Het gedrag dat men wil bekomen zit al in de `replace` functie zelf. Hier zie je ook al dat het niet verplicht is een handler te definiëren. Wanneer de connectie tussen de server en de gebruiker een korte tijd zou verbroken zijn, dan zal deze `due` constructie ervoor zorgen dat enkel de recenste `update` met de laatste nieuwe `status` variabele enkel zal klaarstaan wanneer de verbinding terug herstelt wordt.

```
replace() :: {
  lambda(msg) :: {
    msg.addAttachment("timestamp", time());
    msg.addExpiryCond(
       { reply: false;
         outbox.asVector().iterate(if(el.containsBehaviour("timestamp"),
         { if((name = el.getName()) & (timestamp < el.timestamp),
              reply:=true) } ));
         reply})
      }
    };
```

Zoals in de `timeout` functie wordt er een attachment toegevoegd aan het bericht, hier `timestamp` genaamd. De conditie die aan het bericht gegeven wordt is wel iets ingewikkelder dan die uit de `timeout` functie. Omdat de functionele parameter die meegegeven wordt met de `addExpiryCond` methode volledig uit context getrokken wordt, wat we gezien hebben bij de uitleg van de mixin van berichten, zal de outbox verwijzen naar de outbox van de actor waar de ExpiryCheckMixin op geïnstalleerd is. Dit bericht zal de hele outbox dus aflopen op zoek naar soortgelijke berichten die dezelfde naam hebben en controleren of deze recenter werden aangemaakt. Wanneer er een bericht met dezelfde naam ontdekt wordt die ook nieuwer is dan het huidige bericht, dan zal dit bericht zichzelf zien als vervallen. Dit vervallen heeft dan tot gevolg dat het bericht verwijderd wordt uit de outbox, zodat worden enkel de nieuwste berichten in de outbox bewaard blijven.

OnMessage

De `onMessage` vervalconditie laat toe om bepaalde berichten te laten vervallen door het verzenden van een ander bericht. Wanneer we berichten in het printer-voorbeeld gaan uitrusten met een onMessage foutendetectie systeem dan zouden we bijvoorbeeld normale `print` berichten kunnen laten vervallen wanneer er een `cancel` bericht wordt verstuurd naar de printer. Dit houdt in dat wanneer een `print` bericht in de outbox, waar hij zich in bevindt, een `cancel` bericht ontdekt deze positief zal reageren op de `isExpired` methode.

```
due( onMessage("cancel"),
     { printActor#print(text) },
     expired(true, [thisActor()#handleFailure]))
```

Hierboven ziet men een klein en simpel code voorbeeld van het voorheen besproken gebruik van `onMessage` en zal dan ook de semantiek bezitten die hierboven werd uitgelegd.

```
onMessage(msgName) :: {
  lambda(msg) :: {
    msg.addAttachment("msgName", msgName);
    msg.addExpiryCond(
       { reply: false;
         outbox.asVector().iterate(
          if((msgName = el.getName()),
                   reply:=true)
         );
         reply})
      }
    };
```

Ook hier wordt er een specifieke variabele aangemaakt in het bericht met de naam van een bericht, dat ervoor kan zorgen dat het "cancel" bericht zelf zal falen.

Combine

Bij de mixin die gebruikt wordt om van een bericht een due-bericht te maken werd al aangehaald dat het mogelijk is meerdere condities op te slaan die een bericht kunnen laten falen. De `combine` functie is de manier om dit op een propere manier te verwezenlijken. `Combine` laat toe verschillende van bovengenoemde functies te combineren met elkaar en zo een bericht uit te rusten met meer dan één mogelijke manier van falen. Om de werking van `combine` wat duidelijker te maken, zullen we een klein voorbeeld bespreken.

```
due( combine(timeout(10*1000), onMessage("cancel")),
     { printActor#print(text) },
     expired(true, [thisActor()#handleFailure]))
```

Wanneer we deze code in de context bekijken van het printer-voorbeeld, kan men stellen dat het `print` bericht dat verzonden wordt in de code nu op meerdere

manieren tegelijk kan vervallen door gebruik te maken van eenzelfde failure handler. Het gebruik van de `combine` kan dus heel nuttig zijn in de context van software engineering en toont tegelijkertijd de expressiviteit aan van de vervalconditie functies.

```
combine@args :: {
      lambda(msg) ::{
        for(i:1, i<=size(args), i:=i+1, {
          args[i](msg)
        })
      }
    };
```

`Combine` geeft een functie terug die bij toepassing op een `msg` alle meegegeven expiration condition functies toepast op het bericht.
Bij het combineren van deze functies is het wel nodig heel erg op te letten met naamgevingen omdat het mogelijk is dat men verschillende variabelen overschrijft zodat er inconsistent gedrag kan voorkomen. Het feit dat we de `replace` functie enkel gingen testen bij soortgelijke berichten heeft te maken met de nood aan consistent gebruik bij het combineren van de functies. Wanneer we bijvoorbeeld een bericht gaan testen op zijn timestamp variabele en deze heeft totaal geen timestamp variabele dan zal er een variabele lookup fout optreden in de uitvoering van de code. Hierdoor moet men zeer goed opletten met het gebruik van deze `combine` functie.

5.2.5 Evaluatie

In deze sectie zullen we de `due` taalconstructie gaan evalueren ten opzichte van de criteria voor taalconstructies in een AmOP omgeving die we in hoofdstuk 4 hebben besproken. We zullen hier elk van de criteria in detail bespreken.

Asynchroon omgaan met failures. Het opsporen en behandelen van failures in de `due` taalconstructie wordt geheel onafhankelijk van de normale control flow binnen de actor uitgevoerd. Failures worden bijvoorbeeld asynchroon gesignaleerd aan de desbetreffende actor via een asynchroon bericht. Daardoor gaat de `due` taalconstructie asynchroon om met failures.

Abstractie over tijdelijk verbroken verbindingen. Het gebruik van een tijdslimiet om een failure te identificeren biedt vele voordelen in een AmOP omgeving omdat het toelaat abstractie te maken over kortstondige verbroken verbindingen maar toch foutenafhandeling toelaat voor langdurig verbroken verbindingen. Maar niet enkel een tijdsgebaseerde aanpak is mogelijk in een `due` taalconstructie, het feit dat er meerdere vervalmogelijkheden zijn maakt deze taalconstructie nog beter geschikt om de voorheen aangegeven abstracties te kunnen maken. Hierdoor voldoet `due` aan dit criterium en maakt het deze taal constructie heel geschikt voor gebruik in het AmOP paradigma.

Blokgestructureerde foutafhandeling. Doordat men in de `due` taalconstructie een blok code kan meegeven waarin alle aangemaakte berichten onderworpen worden aan `due` vervalcondities is het zeer aantrekkelijk met het oog op software engineering. Men moet dus geen afzonderlijke failure

handling definiëren voor elke asynchrone verzending. Tevens worden alle berichten die dynamisch aanwezig zijn in het blok code voorzien van failure handling en moeten de asynchrone verzendingen dus niet expliciet in het code blok aanwezig zijn. Ook het feit dat de vervalcondities zeer gemakkelijk toe te voegen zijn en het systeem van handler delegatie en bericht filters zeer expressief is maakt dat op gebied van software engineering dit een heel aantrekkelijke taalconstructie is. De `due` voldoet dus ook aan dit criterium voor bruikbaarheid in een AmOP omgeving.

5.2.6 Conclusie

De `due` taal constructie geeft de programmeur de mogelijkheid om aan failure handling te doen op uitgaande berichten en dit door gebruik te maken van een taal constructie die heel sterke gelijkenissen vertoond met de try-catch, vooral op vlak van handler delegatie. Bij de `due` ligt de klemtoon eerder op het meegeven van condities waarbij de programmeur zelf specifiëert wanneer een bepaald bericht moet falen. In dit opzicht heeft de `due` constructie een heel open implementatie waarbij het toevoegen van condities om berichten te laten vervallen erg gemakkelijk is.

5.3 De `receive_due_otherwise` taal constructie

Zoals de naam van de taal constructie al doet vermoeden zal de `receive_due_otherwise` voor failure handling zorgen in de "Ontvangen" fase, die we al besproken hebben in sectie 5.1. In dit soort failure handling zullen we failures trachten te ontdekken door na te gaan of een verwacht bericht niet op tijd wordt ontvangen door de actor. In dit geval zal er enkel gebruik gemaakt worden van tijdsgebaseerde failure handling om het verschil tussen tijdelijke en permanente fouten te kunnen bepalen.

5.3.1 Syntax en semantiek

Om de uitleg van de syntax en de semantiek duidelijk te maken zullen we gebruik maken van een code voorbeeld. Het voorbeeld dat we gaan gebruiken zullen we het checker-voorbeeld noemen. De omgeving in het checker voorbeeld bestaat uit twee verschillende actoren, een gebruikeractor en een checkeractor. We zullen het checker-voorbeeld toepassen in de context van een beveiligingsfirma, om alles wat duidelijker te maken. De bewaker van de beveiligingsfirma moet gedurende welbepaalde periodes in de buurt komen van het checker-station om er zeker van te zijn dat er niets fout is gegaan met de bewaker. Men zal gebruik maken van de `receive_due_otherwise` in de checker actor om er zeker van te zijn dat binnen een bepaalde deadline de checker een bericht van de bewaker zeker heeft ontvangen, als dit niet het geval is zal er een failure handler opgeroepen worden om bijvoorbeeld een alarm te laten klinken. Hier volgt de code van de `receive_due_otherwise` die men moet gebruiken in de checker actor.

```
receive_due_otherwise( gebruikersActor -> checkIn,
                       60*1000,
                       { display("Gebruiker niet op tijd ingecheckt!")})
```

Het eerste parameter slot van de `receive_due_otherwise` bestaat uit een duo van een actor en een naam van een bericht gescheiden door ->. Dit betekent dat de taal constructie een failure zal aangeven wanneer een bericht `checkIn` van actor `gebruikersActor` niet op tijd wordt ontvangen. Omdat het gebruik van specifieke identificatie van het bericht de programmeur dwingt om de taal constructie enkel in specifieke gevallen te gebruiken werd er nog een andere eigenschap aan deze taal constructie toegevoegd. Om een algemeen geval te beschrijven kan men gebruik maken van een wildcard. De wildcard _ kan zowel gebruikt worden in het actor slot als in het naam slot. Wanneer we dus eender welk bericht van een bepaalde actor `anActor` verwachten kunnen we dit op deze manier in de taal constructie coderen: `anActor` − > _.
In het tweede parameter slot moet een tijdswaarde worden bepaald om aan te geven hoe lang er moet gewacht worden op het bericht dat gespecifieerd werd in het eerste parameter slot. Merk op dat we hier enkel een periode in milliseconden kunnen ingeven en er dus geen mogelijkheid bestaat de semantiek van failure detectie te veranderen.
De handlers in de `receive_due_otherwise` taal constructie worden gedefinieerd in het derde parameter slot. De handler die kan gedefinieerd worden in het derde parameter slot is ook geparameteriseerd met twee parameters, namelijk `requiredMsg` en `requiredActor`. In de handler kan dus gebruik gemaakt worden van het duo dat men kan definiëren in het eerste parameter slot. In tegenstelling tot de `due` is het hier niet mogelijk meerdere handlers te definiëren in een `receive_due_otherwise` constructie. Het is echter wel mogelijk meerdere `receive_due_otherwise` constructies op hetzelfde bericht te definiëren met bijvoorbeeld een andere timeout waarde en een andere handler. Wanneer men echter twee soortgelijke `receive_due_otherwise` constructies na elkaar gaat definiëren die enkel in handlers verschillen dan zal de eerste definitie genegeerd worden en overschreven worden door de tweede. Met andere woorden, het actor-bericht duo samen met de timeout waarde bepalen uniek een failure handler.
Nu we de basis syntax en de bijbehorende semantiek besproken hebben kunnen we de meer geavanceerde semantiek toelichten. Aan de hand van een nieuw voorbeeld zal uitgelgd worden hoe de `receive_due_otherwise` zich gedraagt wanneer er meerdere constructies worden gedefinieerd op dezelfde binnenkomende berichten.

```
receive_due_otherwise(
        anActor -> normalMessage,
        15*1000,
        { display("message not received in time 1: ", requiredMsg, eoln) });
receive_due_otherwise(
        anActor -> normalMessage,
        10*1000,
        { display("message not received in time  2: ", requiredMsg, eoln) });
receive_due_otherwise(
        anActor -> normalMessage,
        13*1000,
        { display("message not received in time  3: ", requiredMsg, eoln) });
```

Het voorbeeld bestaat uit drie `receive_due_otherwise` constructies die telkens op hetzelfde actor/bericht koppel gedefinieerd zijn. Elk van deze constructies bezit een eigen timeout waarde en een verschillende handler. Er zijn nu twee

MOP methode	DueMixin	ExpiryCheckMixin	ReceiveDueOtherwiseMixin
init		verfijnt	
createMessage	verfijnt		
send	verfijnt		
process			
in			verfijnt

Tabel 5.2: Een overzicht van welke MOP methoden de `receive_due_otherwise` taal mixin overschrijft of verfijnt ten opzichte van de twee eerder besproken `due` mixins.

gebeurtenissen die we moeten beschrijven om de semantiek volledig duidelijk te maken, wanneer het bericht niet op tijd wordt ontvangen door de actor waarin al deze constructies werden gedefinieerd of wanneer het bericht wel op tijd ontvangen wordt.

Bericht op tijd ontvangen

Wanneer het bericht wordt ontvangen voor de vroegste deadline, hier `10*1000`, is verstreken dan zal er geheel niets gebeuren en worden alle `receive_due_otherwise` aanroepen genegeerd die erop gericht waren failure handling te plaatsen op het ontvangen bericht. Moest het bericht nu ontvangen worden na 12 seconden dan zal de tweede handler al aangeroepen zijn, maar omdat het bericht is ontvangen na deze 12 seconden zullen de andere handlers niet meer uitgevoerd worden. Deze regels vormen dus de semantiek wanneer een bericht op tijd ontvangen wordt.

Bericht niet op tijd ontvangen

Het niet op tijd ontvangen zorgt ervoor dat eerst de handler met de laagste deadline waarde uitgevoerd zal worden, gevolgd door de tweede laagste en zo verder. Alle handlers zullen dus uitgevoerd worden na de gespecifieerde tijd. Zoals in het vorige deel al werd aangehaald zullen deze handlers enkel worden uitgevoerd wanneer het bericht nog niet is ontvangen; eens ontvangen worden alle nog niet uitgevoerde handlers van toepassing op het bericht genegeerd. Dit negeren zal ook van toepassing zijn op de wildcards die men kan gebruiken. Bij een wildcard houdt dit in dat de handler voorgoed kan genegeerd worden bij ontvangst van minstens één bericht dat aan de wildcard voldoet.

5.3.2 Het design

Om de `receive_due_otherwise` taal constructie te implementeren werd opnieuw gebruik gemaakt van mixins in AmbientTalk, die eerder al in hoofdstuk 3 werden besproken, evenals in de uitleg van het design van `due`. We zagen ook al in hoofdstuk 3 dat men heel erg moet oppassen wanneer men verschillende mixins gaat combineren, hierdoor zullen we alle tot nu toe geziene mixins samenvoegen in een tabel om te vergelijken wie welke MOP methode overschijft.

Uit de tabel kunnen we afleiden dat alle mixins compatibel zijn en dus op éénzelfde actor toegepast kunnen worden.

5.3.3 Implementatie

Deze sectie zal een gedetailleerde bespreking geven van de implementatie van de `receiveDueOtherwiseMixin` in AmbientTalk. De implementatie bestaat uit drie grote methoden in de mixin zelf en een aangepaste `Notifier` actor die we ook kort zullen beschrijven.
We zullen de bespreking beginnen met een verzicht te geven van de implementatie van de taalmixin.

```
ReceiveDueotherwiseMixin(periode) :: {
  ‘instance variabelen‘
  monitoredMessages : smallmap.multimapMixin().new();
  notificationActor : notifier(periode);

  ‘publieke interface‘
  a -> b() :: ... ;
  receive_due_otherwise(pair,
                        timeout,
                        handler(requiredActor, requiredMsg)) :: { ... };

  ‘hulp methoden‘
  notify(msgPair, timeout) :: { ... };

  ‘verfijnde MOP methoden‘
  in(msg) :: { ... };

  capture()
}
```

Om gebruikt te kunnen maken van de `receive_due_otherwise` moet men deze mixin toepassen op een actor waarin men hem wil gebruiken. De `periode` die men meegeeft zal gebruikt worden om de intensiteit aan te tonen waarin de deadlines worden getest, hoe kleiner deze waarde hoe nauwkeuriger het testen van de deadline.
De globale werking van deze mixin bestaat erin om bij een aanroep van de `receive_due_otherwise` een handler te registreren in de `monitoredMessages` map. De aanroep heeft ook tot gevolg dat er een timeout periode start waarbinnen een bepaald bericht ontvangen moet worden. De `in` methode zorgt ervoor dat alle binnenkomende berichten onderschept worden en wanneer dit bericht aanweizg was in de `monitoredMessages` map zal deze eruit verwijdert worden. Na het verstrijken van de timeout van een bepaald bericht wordt de handler opgezocht in de map en kunnen er zich twee verschillende gevallen voordoen.

- Wanneer de handler gevonden wordt in de map, dan werd het bericht niet op tijd ontvangen en zal de handler uitgevoerd moeten worden.
- Indien de handler niet wordt gevonden in de map is er geen probleem en mag dit geval genegeerd worden.

We zullen nu wat dieper ingaan op elk element uit de implementatie en deze verder toelichten beginnende met de datastructuur die we zullen gebruiken.

normalMessage	< [anActor, 15*1000, handler1], [anActor, 10*1000, handler2], [anActorr, 13*1000, handler3] >
...	...

Tabel 5.3: Weergave van manier waarop berichten en handlers opgeslagen worden in de multimap.

```
monitoredMessages : smallmap.multimapMixin().new();
```

Deze multimap zal gebruikt worden om alle berichten in op te slaan die op dat moment verwacht worden door de actor. Een multimap bestaat uit een key die toegang geeft tot een vector waarin data kan opgeslaan worden, een key geeft dus toegang tot een vector van data. De namen van de berichten zullen dienen als key met alle relevante informatie als data slot per key. Deze informatie omvat de variabelen `pair[1]` (of beter gezegd de sender), `timeout` en `handlerBlock`. Om deze structuur te verduidelijken zullen we het voorbeeld uit sectie 5.3.1 weergeven in volgende tabel. Bij de uitleg van de syntax en semantiek werd de pijlnotatie al uitgelegd, deze notatie is echter niet standaard in AmbientTalk en de mixin implementeert deze dus zelf. Dit kan doordat men in Pico, waarop AmbientTalk gebaseerd is, zelf operatoren kan definiëren.

```
a -> b() :: [ a, closure_name(b) ];
```

De pijl notatie zal er voor zorgen dat zowel de actor als de naam van het bericht in een tabel worden opgeslagen in een formaat dat handig is in de verdere implementatie van de mixin. Om niet met strings hoeven te werken om de naam van het bericht te identificeren zal het tijdelijk als een functie gezien worden en kan er dus door de functie `closure_name` te gebruiken toegang geboden worden tot meta data van de functie die een string bevat met de naam van de functie erin, welke de string is die we nodig hebben om het bericht te identificeren.
Nu we de kleine stukken code besproken hebben zullen we de implementatie van de grotere methoden gaan beschrijven. We zullen beginnen met de code te bespreken van de `receive_due_otherwise` taal constructie zelf.

```
receive_due_otherwise(    pair ,
                          timeout,
                          handlerBlock(   requiredActor,
                                          requiredMsg) ) :: {
      msgName: pair[2];
      monitoredMessages.put(msgName, [ pair[1], timeout, handlerBlock ] );
      notificationActor#notifyMeIn(timeout, pair);
      void
    };
```

Bij het aanroepen van de taal constructie wordt alles opgeslagen in de multimap met de naam van het bericht als key en wordt een slot van de vector gevuld met een tabel met alle informatie erin. Hierna zal de informatie ook naar de `notificationActor` gestuurd worden die voorheen al was geset op de speciale `Notifier` actor die reeds werd aangehaald in de inleiding van deze sectie. Om

beter te begrijpen hoe dit in zijn werk gaat zullen we er de pseudocode van deze actor ook bijhalen.

```
notificationMixin() :: {
      ‘Verzendt een notify bericht naar de actor die het notifyMeIn
      bericht verstuurde na timeout milliseconden en geeft daarin de
      args en timeout variabele mee.‘

      notifyMeIn(timeout, args) :: { ... }
}
```

Deze `notifier` krijgt `notifyMeIn` berichten van de actoren die deze actor willen gaan gebruiken. Bij de aanroep van de `notifyMeIn` berichten zal een deadline opgesteld worden die aangeeft wanneer er een `notify` bericht moet teruggestuurd worden naar de actor die voorheen het `notifyMeIn` bericht verstuurde. Wanneer de deadline verstreken is zal een `notify` bericht worden teruggestuurd met de informatie die voorheen werd meegegeven. Het is de bedoeling dat actoren die deze `notifier` willen gebruiken de `notify` methode gaan implementeren.
Wanneer we de implementatie van `receive_due_otherwise` er weer bijhalen zien we dat de actor een bericht `notify` wil terugkrijgen binnen `timeout` milliseconden met als informatie het sender/selector `pair`.
De `notify` methode wordt dus opgeroepen door de `notificationActor` nadat de timeout is verstreken en wanneer het bericht zou moeten ontvangen zijn.

```
notify(msgPair, timeout) :: {
      msgName : msgPair[2];
      msgActor : msgPair[1];
      tobeDeleted : vector.new();
      if(monitoredMessages.containsKey(msgName), {
         handlers: monitoredMessages.get(msgName);
         handlers.iterate(
           { storedActor : el[1];
             storedTimeout : el[2];
             storedHandler : el[3];
             if(and(storedActor = msgActor,
                      storedTimeout = timeout),
                { handler : storedHandler;
                  handler(msgActor, msgName);
                  tobeDeleted.add([msgName, el]) })
           })
      });
   tobeDeleted.iterate(
     { msgName : el[1];
       tobeDeletedHandlers : el[2];
       monitoredMessages.deleteItem(msgName, tobeDeletedHandlers) })
};
```

Wanneer de `notify` methode uitgevoerd wordt is de deadline voor een bepaald bericht afgelopen en als het voor deze tijd nog niet werd ontvangen zullen handlers moeten opgeroepen worden. Het achterhalen of het bericht al werd

ontvangen of niet gebeurd door in de multimap te gaan kijken of de handlers van het bericht nog aanwezig zijn. Wanneer de naam van het bericht gevonden wordt in de multimap zal de vector met alle handlers afgelopen worden op zoek naar de handler die dezelfde timeout en oorsprong bezit. Nadat de juiste handler is gevonden zal deze uitgevoerd worden. Na de uitvoering zal deze handler verwijderd worden uit de vector met als key de naam van het bericht. Het is dus mogelijk dat er nog meerder handlers overblijven na de eerste uitvoering en verwijdering van een handler.
Nu we de behandeling hebben besproken van de berichten die niet worden ontvangen zullen we nu de code bespreken van de behandeling van de berichten die wel op tijd worden ontvangen. Om dit mogelijk te maken zal de `in` methode worden overschreven, zoals al in de design sectie en in hoofdstuk 3 (inbox observer) duidelijk werd gemaakt is dit ook een MOP methode. De `in` methode wordt opgeroepen wanneer er een bericht in de inbox van een actor wordt geplaatst, het is de ideale manier om het gedrag om op tijd berichten te ontvangen te implementeren.

```
in(msg) :: { .in(msg);
   tobeDeleted : vector.new();
   if(monitoredMessages.containsKey(msg.getName()),
     { handlers: monitoredMessages.get(msg.getName());
       handlers.iterate(
         { storedActor : el[1][1];
           if(or(msg.getSource() = storedActor,
                 is_void(storedActor)),
              { tobeDeleted.add([msg.getName(),el]) })
         }) });
   if(monitoredMessages.containsKey("_"),
      { handlers: monitoredMessages.get("_");
        handlers.iterate(
          { storedActor : el[1][1];
            if(or(msg.getSource() = storedActor,
                  is_void(storedActor)),
               { tobeDeleted.add(["_", el]) })
          })
      });
   tobeDeleted.iterate(
     { msgName : el[1];
       tobeDeletedHandlers : el[2];
       monitoredMessages.deleteItem(msgName, tobeDeletedHandlers) })
};
```

Een inkomend bericht van actoren a_{in} genaamd m_{in} matcht met de handler a_{hdl} -> m_{hdl} als en slechts als aan minstens een van de volgende condities voldaan wordt.

- Wanneer m_{in} gelijk is aan m_{hdl} en ofwel a_{in} overeenkomt met a_{hdl} ofwel a_{hdl} gelijk is aan de wildcard _, dan zal het inkomende bericht matchen.
- Wanneer m_{hdl} gelijk is aan de wildcard _ en ofwel a_{in} gelijk is aan a_{hdl} ofwel a_{hdl} overeenkomt met de wildcard, dan zal het inkomende bericht matchen.

De handlers die hier verwijderd worden kunnen dus niet meer opgeroepen worden indien er na de ontvangst van een bepaald bericht een notify voor dit bericht wordt uitgevoerd. de `is_void` test bij het checken van de actoren geeft een goed resultaat omdat we in de receiveDueOtherwiseMixin de _ variabele definiëren als `void`.

```
_ :: void;
```

5.3.4 Evaluatie

In deze sectie zullen we de `receive_due_otherwise` taalconstructie gaan evalueren ten opzichte van de criteria voor taalconstructies in een AmOP omgeving die we in hoofdstuk 4 hebben besproken. We zullen hier elk van de criteria in detail bespreken.

Asynchroon omgaan met failures. Na het registreren van de handler en het bepalen van de deadline wordt alles op zulk een manier afgehandeld dat het normale verloop van de actor niet geblokkeerd wordt. Hierdoor is het detecteren en behandelen van failures asynchroon in de `receive_due_otherwise` taalconstructie en voldoet het aan het voorgeschreven criterium om bruikbaar te zijn in een AmOP omgeving.

Abstractie over tijdelijk verbroken verbindingen. Net zoals bij `due` maakt deze taalconstructie gebruik van tijdsgebaseerde failure detectie waardoor het mogelijk wordt te abstraheren over kortstondig verbroken verbindingen en de langdurig verbroken verbindingen te kunnen behandelen met handlers. Hierdoor voldoet `receive_due_otherwise` ook aan dit criterium.

Blokgestructureerde foutafhandeling. Doordat het niet mogelijk is meerdere berichten van dezelfde failure handler te voorzien door enkel gebruik te maken van één `receive_due_otherwise` voldoet het niet aan dit criterium. Het is echter wel mogelijk om met zogenaamde wildcards een breder bereik te creëren van berichten waarop de `receive_due_otherwise` zal triggeren. Het is natuurlijk niet uitgesloten dat verdere uitbreidingen aan deze taalconstructie dit wel mogelijk maken.

5.3.5 Conclusie

De `receive_due_otherwise` maakt het mogelijk om op een nietblokkerende wijze de ontvangst van bepaalde berichten af te wachten en niet-ontvangen berichten naar wens van de programmeur te behandelen. Een nadeel dat kan aangehaald worden is het feit dat men de handlers per verwacht bericht moet definiëren. Het gebruik van een tijdswaarde geeft ons de mogelijkheid om een onderscheid te kunnen maken tussen tijdelijke en permanente fouten. Doordat we slagen voor het grootste deel van de criteria kunnen we stellen dat de `receive_due_otherwise` een geschikte taalconstructie is om aan foutenafhandeling te doen in ambient-georiënteerde programmas.

5.4 `reply_before` taal constructie

De laatste taalconstructie die we gaan bepreken heet `reply_before`. Deze constructie wordt gebruikt om failure handling toe te kunnen passen op de

"Verwerken" fase die eerder werd besproken in de inleiding van dit hoofdstuk. Eerst worden futures in AmbientTalk [3] uitgelegd, waarna de syntax en semantiek van `reply_before` besproken wordt. Na de syntax en semantiek wordt de implementatie in groot detail uitgelegd. Ten slotte zullen we de invloed van promise pipelining, zoals besproken in sectie 2.6.1. in de taal E, op deze taalconstructie bespreken.

5.4.1 Futures in AmbientTalk

Wanneer we de `reply_before` willen gaan gebruiken hebben we de futuresMixin [3] nodig omdat de eerste paramater uit de `reply_before` taal constructie een future moet zijn. De enige manier om een future terug te krijgen uit een asynchrone verzending bekomt men door de futuresMixin toe te passen op de actor waarbinnen de verzending plaatsvindt. Een future wordt teruggegeven bij het verzenden van een asynchroon bericht en is een tijdelijke plaatsvervanger van de return waarde van de verzending. Met een `when` constructie kan men de werkelijke waarde van de future, die men bekomt door een asynchroon bericht te versturen, bekomen en deze dan in code gaan gebruiken. Deze `when` constructie zal dus een stuk code uitvoeren op het moment dat de future zijn werkelijke waarde heeft gekregen. Het gebruik van de `when` zal aangetoond worden door gebruik te maken van het instant messenger voorbeeld uit [3]. In het voorbeeld kan een instant messenger als volgt de nickname van een buddy bekomen:

```
when(buddy#getNickname(),
     display("Buddy online: ",content, eoln))
```

Het voorbeeld geeft weer hoe men de `when` constructie moet gaan gebruiken. Het asynchrone bericht in de eerste parameter geeft een future terug. Wanneer de future wordt geresolved, zal de code in het tweede parameter slot uitgevoerd worden. De code die uitgevoerd wordt wanneer de future geresolved wordt is geparameteriseerd met de `content` variabele die de werkelijke waarde van de future bevat.
Een nadeel aan het gebruik van `when` is het feit dat de code ervan niet zal uitgevoerd worden als de future nooit geresolved wordt. Dit kan voorvallen wanneer de communicatieverbinding tussen twee actoren verbroken wordt en niet meer wordt hersteld. Dit probleem zal worden behandeld door gebruik te maken van de `reply_before` in samenhang met `when`.

5.4.2 Syntax en semantiek

Om de syntax en semantiek beter uit te kunnen leggen zullen we ook hier gebruik maken van het instant messenger voorbeeld besproken in [3]. Als voorbeeld zullen we de situatie bekijken waarin we twee gebruikers hebben die berichten naar elkaar willen versturen door gebruik te maken van het instant messenger systeem. Wanneer een bericht goed verstuurd werd willen we de gebruiker daarvan op de hoogt houden. Dit kan men door een `when` constructie zodat de gebruiker op de hoogte wordt gebracht wanneer het versturen, ontvangen en verwerken werd volbracht. Voor de gevallen waarbij een failure opduikt en er geen tijdig antwoord van de tegenpartij komt zullen we gebruik maken van een `reply_before` constructie. Met deze taalconstructie kunnen we een behandeling voorzien, hier een bericht naar de gebruiker, wanneer er een failure opduikt in

het verwerken van een bericht. Men moet wel opletten dat een `reply_before` failure handler niet kan garanderen dat het bericht niet ontvangen werd, dit in tegenstelling tot `due` die besproken werd in sectie 5.2. Het kan enkel garanderen dat de reply binnen een bepaald aantal milliseconden nog niet ontvangen werd. Het is dus nog altijd mogelijk dat de future waarop de `reply_before` werd gedefinieerd na de deadline nog resolved wordt. De code voor dit voorbeeld vindt men hier.

```
future: buddy#receiveTextMessage(text);
when( future,
      { display("message sent", eoln) });
reply_before(  future,
               60*1000,
               display("could not send message", eoln))
```

Door gebruik te maken van de bespreking van de futuresMixin en het voorheen gegeven voorbeeld zullen we nu de syntax en semantiek van `reply_before` in detail beschrijven. Voor we aan de beschrijving kunnen beginnen zullen we eerst een korte algemene omschrijving geven van de twee te bespreken taalconstructies:

`when` is een onderdeel van de futures mixin/taalconstructie dat gebruikt kan worden om een blok code uit te voeren wanneer er een future geresolved wordt. Deze code kan men ook parameteriseren met de eigenlijke waarde van de geresolvede future.

`reply_before` kan gebruikt worden om indien een future niet binnen een bepaalde timeout waarde geresolved is een failure handler op te roepen. Deze handler kan trouwens ook geparameteriseerd worden met een recover functie die ervoor zorgt mogelijke when code toch kan uitgevoerd worden indien er een failure optreedt.

In het bovenstaande voorbeeld geeft de `buddy#receiveTextMessage(text)` aanroep een future teruggeven. Het tweede parameter slot uit de `reply_before` wordt gebruikt om een timeout in te specifiëren. Deze timeout wordt gebruikt om een limiet in te stellen hoe lang `buddy` er mag over doen om het bericht `receiveTextMessage` te verwerken en het resultaat terug te sturen. In het geval dat de verwerking van het bericht binnen de gegeven timeout periode werd afgehandeld is er niets aan de hand en zijn verdere acties overbodig. Wanneer het echter langer duurt zal de code uit het derde parameter slot uitgevoerd worden. Voor de code uit het derde parameter slot wordt uitgevoerd zullen alle `when` constructies gedefinieerd op dezelfde future verwijderd worden zodat deze door mogelijk latere resolving van de future niet meer worden uitgevoerd. In het voorbeeld zal de gebruiker dus nooit het bericht "message sent" zien eens hij het bericht "could not send message" heeft gezien. Een uitgebreide bespreking van de interactie tussen `when` en `reply_before` volgt in de volgende sectie.

Recovery semantiek

Nu we de simpele syntax en semantiek besproken hebben zullen we ook de meer geavanceerde semantiek gaan bespreken. Onder deze geavanceerde semantiek verstaan we het gebruiken van de recover semantiek die beschikbaar is in de

reply_before. Om een meer concrete uitleg te kunnen geven zullen we een voorbeeld van een berekeningsactor beschrijven zodat we zo de geavanceerde semantiek kunnen bespreken.

```
future: berekenActor#zwareBerekening(berekenParameters);
when(  future,
       thisActor()#resultaatGebruiken(content));
reply_before(  future,
               60*1000, {
               when( andereBerekenActor#zwareBerekening(berekenParameters),
                     recover(content)) });
```

In dit voorbeeld zullen we werken met de future van de asynchrone verzending berekenActor#zwareBerekening(berekenParameters). Deze future gaan we dan gebruiken in zowel een when als een reply_before constructie. De when zal ervoor zorgen dat de methode resultaatGebruiken opgeroepen wordt indien future geresolved wordt naar zijn eigenlijke waarde. De reply_before constructie die gedefinieerd werd op de future wordt gebruikt indien de berekening niet binnen de 60 seconden afgehandeld werd, waardoor we veronderstellen dat er een failure is opgetreden. We behandelen deze failure door een nieuw zwareBerekening bericht te versturen naar een andere actor (andereBerekenActor) en de future die we terugkrijgen van deze verzending zullen we dan in een when constructie plaatsen.
In het geval dat de handler van reply_before opgeroepen wordt en de tweede when gedefinieerd wordt op een nieuwe future dan biedt de recover functie de mogelijkheid om de when constructies, die gedefinieerd werden op de future, die gebruikt wordt in dezelfde reply_before, alsnog uit te voeren met de parameter van de recover functie als content waarde. Wanneer dus de future van de verzending andereBerekenActor#zwareBerekening(berekenParameters) resolved dan zal alsnog de handler van de eerste when constructie uitgevoerd worden, wat betekent dat de uitkomst die men terugkrijgt van andereBerekenActor zal gebruikt worden in de resultaatGebruiken(content) aanroep. Het is wel heel belangrijk om op te merken dat future niet zal resolven wanneer de recover uit de code van de tweede when uitgevoerd wordt. Hoewel de when constructies die actief waren op future uitgevoerd worden, wil dit niet zeggen dat future zelf geresolved wordt. Dit houdt in dat wanneer er andere actoren zouden aanwezig zijn in het geheel die dezelfde future zouden gebruiken niet aangetast zouden worden met een artificiële resolve waarde uit de recover functie. Er is dus een groot verschil in het gebruik tussen de recover functie in de handler van reply_before en het expliciet resolven van de gebruikte future in de handler van reply_before. In het eerste geval zullen enkel lokale when constructies op dezelfde future aangetast worden terwijl bij het expliciet resolven zullen alle actoren die gebruik maakten van de future aangetast worden. Een belangrijk aspect van het gelijk gebruik van reply_before en when is het *niet-determinisme* wat betreft het verwerken van het bericht. Met niet-determinisme wordt hier bedoelt dat het onmogelijk te voorspellen is welk event er eerst zal gebeuren:

1. De future wordt resolved.
2. De timeout periode verstrijkt.

Methode	replyBeforeMixin	futuresMixin
interface		
when reply_before	 implementeert	implementeert
MOP		
process createMessage send		overschrijft verfijnt verfijnt
implementatie		
invokeHandlers notify invokeWhen	implementeert implementeert 	 implementeert

Tabel 5.4: Een overzicht van alle methoden die de replyBeforeMixin en futuresMixin implementeren.

Wanneer eerst 1 en dan 2 voorkomt dan wordt 2 gewoon genegeerd, in het omgekeerde geval wordt 1 genegeerd tenzij er een expliciete `recover` functie opgeroepen wordt in de handler van `reply_before`, de future wordt echter niet resolved maar het gevolg ervan wel, met name het uitvoeren van de `when` code. De programmeur moet zich van dit probleem dus bewust zijn en zijn code hiernaar aanpassen.

5.4.3 Het design

Uit de vorige sectie werd al duidelijk dat men bij het gebruik van de replyBeforeMixin moet gebruik maken van een andere mixin, futuresMixin genaamd. De `reply_before` taalconstructie werd opnieuw als een mixin geïmplementeren. Dit houdt in dat men het gedrag van de `reply_before` modulair kan toevoegen aan elke actor. Maar bij het gebruik van mixins moet men altijd oppassen bij het overschijven van methoden omdat men het gedrag van andere mixins kan schaden. Omdat we altijd gebruik moeten maken van de futuresMixin zullen we de methoden in deze mixin naast de methoden uit de replyBeforeMixin plaatsen zodat we deze kunnen gaan vergelijken. We kunnen stellen dat de replyBeforeMixin een modulaire uitbreiding is van de futuresMixin.

Uit de tabel kan men vrij duidelijk opmaken dat er geen overlappingen zijn, daarom zijn deze twee mixins samen perfect toepasbaar op eenzelfde actor.

5.4.4 Implementatie

In deze sectie zullen we de implementatie van de replyBeforeMixin in detail uitleggen. Dit houdt in dat we de code van de mixin stuk voor stuk zullen bespreken. Om een deel van de uitleg beter te begrijpen heeft men kennis nodig van de notificatie actor, deze actor staat beschreven in sectie 5.3.3. Niet alleen moet men gebruik maken van de futuresMixin om gebruik te kunnen maken van de replyBeforeMixin, de implementatie komt ook in veel gevallen overeen met deze van de `when` constructie. Deze gelijkenis zullen we in deze sectie dan ook

verder gaan toelichten.
Om te beginnen zullen we een overzicht geven van de code in zijn geheel vooraleer we dieper zullen ingaan op specifieke onderdelen ervan. De volgende code stelt de implementatie van de replyBeforeMixin voor.

```
{
   ‘hulpfunctie buiten de mixin zelf‘
   replyBeforeListener: root.object({ ... });
   ‘de mixin zelf‘
   replyBeforeMixin(aDelay)::{
    ‘instance variabelen‘
    handlerBlocks: smallmap.multimapMixin().new();
    notificationActor : notifier(aDelay);
    ‘hulpfuncties‘
    removeHandlers(aFuture)::{ ... };
    notify(aFuture, timeout) :: { ... };
    ‘publieke interface voor de taal constructie‘
    reply_before(aFuture, timeout, handlerCode(future))::{ ... };

    capture()
  }
};
```

We zullen deze bespreking starten met het bekijken van de datastructuur die we zullen gebruiken in de implementatie van de `reply_before`.

```
handlerBlocks: smallmap.multimapMixin().new();
```

Een bespreking van de multimap kan men terugvinden in sectie 5.3.3. In de `handlerBlocks` zullen de verschillende `reply_before` handlers opgeslagen worden met de future als key in de map. Een multimap geeft de mogelijkheid om meerdere handlers te kunnen definiëren per future. Naast deze multimap zal deze implementatie ook gebruik maken van een notification actor waarvan men een bespreking in sectie 5.3.3 terugvindt.

```
notificationActor : notifier(aDelay);
```

We zullen nu eerst de centrale methode `reply_before` bespreken en vanuit deze beschrijving zullen we dan de hulpmethoden bespreken. De `reply_before` methode bevat het gedrag van de taal constructie zelf en moet dus aanroepen worden om failure handling toe te gaan passen op een future.

```
reply_before(aFuture, timeout, handlerCode(recover))::{
      handlerBlocks.put(aFuture, [handlerCode, timeout]);
      aFuture#subscribe(actor(replyBeforeListener.new(aFuture, thisActor())));
      notificationActor#notifyMeIn(timeout, aFuture);
      void
};
```

Wanneer een `reply_before` wordt aangeroepen dan zal het systeem eerst de handler, die meegegeven werd in het derde parameter slot, in de voorheen besproken multimap `handlerBlocks` opslaan met als key de future uit het eerste parameter slot. Ook moet men opmerken dat men zowel de code van de handler

als de timeout zal opslaan, dit gebeurd op deze manier zodat we de handlers op basis van de timeout kunnen gaan selecteren en daarna dan ook uitvoeren.
Een heel belangrijk deel van de implementatie bestaat eruit een listener `replyBeforeListener` te gaan definiëren op een bestaande future door gebruik te maken van de `subscribe` methode. Het versturen van het `subscribe` bericht naar een future zal tot gevolg hebben dat de actor die als parameter wordt meegegeven bij het `subscribe` bericht een `notify` bericht zal toegestuurd krijgen wanneer deze future geresolved wordt. Hierdoor kan men dus gedrag implementeren dat zal uitgevoerd worden wanneer een future geresolved wordt. De `replyBeforeListener` wacht totdat de future waarmee hij geïnitialiseerd werd geresolved wordt zodat hij een bericht kan versturen naar de actor waarin de `reply_before` handlers voor deze future gedefinieerd werden

```
replyBeforeListener: root.object({
      future : void;
      reference: void;
      cloning.new(aFuture, aReference)::{ future:=aFuture;
                                          reference:=aReference };
      notify(content)::reference#removeHandlers(future)
});
```

Bij het aanmaken van deze listener wordt er een future en een referentie opgeslagen, de future is diegene waarbij deze listener geregistreerd is en de referentie `reference` specificeert de locatie van de actor die wil weten wanneer de future geresolved zal worden. Deze listener implementeert de `notify` methode die de future zal sturen wanneer hij resolved is. De `notify` methode wordt dan op zijn beurt gebruikt om een andere actor ervan op de hoogte te brengen dat alle `reply_before` handlers op deze future mogen verwijderd worden.
Na het inschrijven van een listener bij de future wordt er een bericht gestuurd naar de `notificationActor`. De `notificationActor` zal dan `thisActor()`, hetgeen de actor voorstelt waarop de replyBeforeMixin is toegepast, ervan op de hoogte stellen als de timeout van `timeout` milliseconden verstreken is. Om te identificeren welke handler er moet opgeroepen worden zal de future meegegeven worden naar de `notificationActor`. Wanneer de timeout verstreken is zal de `notificationActor` een `notify` methode sturen naar de `thisActor()` met de nodige informatie. Hierdoor moeten we dus een `notify` callback-methode implementeren in de replyBeforeMixin. De `notify` methode zorgt ervoor dat de `reply_before` handler voor een bepaalde future en timeout uitgevoerd wordt indien deze nog aanwezig is in de `handlerBlocks` map. Als dit niet het geval is dan betekent dit dat de future op tijd werd geresolved.

```
 notify(aFuture, timeout) :: {
      tobeDeleted : vector.new();
      if(handlerBlocks.containsKey(aFuture), {
           handlers: handlerBlocks.get(aFuture);
           handlers.iterate(
           { handlerCode : el[1];
             handlerTimeout : el[2];
             if(handlerTimeout = timeout,
                { if(whenBlocks.containsKey(aFuture), {
```

```
                    whenCode: whenBlocks.get(aFuture);
                    whenBlocks.delete(aFuture);
                    recover(value) :: { whenCode.iterate(el@[value]) };
                    handlerCode(recover) }, {
                    recover(value) :: { void };
                    handlerCode(recover) });
                  tobeDeleted.add(el) })
              })
       });
       tobeDeleted.iterate(
         { handlerBlocks.deleteItem(aFuture, el) })
};
```

Wanneer er een handler wordt gevonden die overeenkomt met de `timeout` variabele zijn er twee mogelijkheden, ofwel zijn er `when` constructies gedefinieerd op de future waarop de `reply_before` toegepast is, ofwel zijn er geen `when` gedefinieerd op deze future. In het eerste geval zullen alle `when` code blokken opgehaald worden en in de `whenCode` variabele gestoken worden. Hierna worden deze code blokken voorgoed verwijdert uit de futuresMixin zodat deze `when` code niet meer kan uitgevoerd worden wanneer de future, waarop deze `when` code gedefinieerd werd, toch nog geresolved zou worden. De `when` code kan echter wel nog opgeroepen worden door gebruik te maken van de `recover` functie waarvan men de definitie in de code terugvindt. Zoals men kan zien wordt de future dus helemaal niet resolved en zal de `reply_before` handler dus geen effect hebben op het gebruik van diezelfde future in andere actoren en kan deze daar dus zonder probleem wel nog door de "echte" waarde resolved worden. Wanneer er echter geen `when` constructies gedefinieerd zijn op de future is het niet nodig deze te deleten en zal ook de `recover` functie gelijk zijn aan `void`.
We zagen eerder bij de `replyBeforeListener` dat deze de `removeHandlers` methode oproept wanneer de future geresolved wordt. Dit vormt dan ook de laatste schakel uit de implementatie van de replyBeforeMixin. De `removeHandlers` methode wordt gebruikt om handlers uit de datastructuur te verwijderen indien de future op tijd geresolved wordt.

```
removeHandlers(aFuture, content)::{
       if(handlerBlocks.containsKey(aFuture), {
          handlerBlocks.delete(aFuture)
       })
};
```

Net zoals bij de `notify` methode zal men de future gebruiken om na te gaan of deze aanwezig is in de multimap van de handlers. Omdat deze methode aangeroepen wordt wanneer de future geresolved wordt is het hierna niet meer nodig eender welke handler van deze future op te roepen. Daarom zullen alle handlers van de future verwijderd worden uit de map.

5.4.5 Promise pipelining

In hoofdstuk 2 zagen we al dat de taal E gebruik maakt van zogenaamde promise pipelining bij promises, ook wel futures genaamd. In AmbientTalk heeft men een zelfde mechanisme waarmee men naar futures al asynchrone berichten kan

sturen voordat deze geresolved werden. Promise pipelining slaat dan op het feit dat wannneer men naar een future een bericht stuurt en naar de future die men dan bekomt ook een bericht stuurt dit een kettingreactie oplevert wanneer de eerste future geresolved wordt. Door gebruik te maken van de `reply_before` kan men ervoor zorgen dat een kettingreactie niet stopgezet kan worden door een of andere failure in het systeem. We kunnnen immers "niet-op-tijd-geresolvede" futures zelf gaan resolven in de failure handlers van de `reply_before`. De `reply_before` taal constructie kan dus gebruikt worden voor gegarandeerde promise pipelining.

5.4.6 Evaluatie

In deze sectie zal de `reply_before` taalconstructie geëvalueerd worden aan de hand van de criteria die we in hoofdstuk 4 hebben opgesteld. Herinner dat deze criteria de geschiktheid van een taalconstructie in het AmOP paradigma bepalen. Een bespreking van deze drie criteria ten opzichte van de `reply_before` vindt men hieronder.

Asynchroon omgaan met failures. Uit de bespreking van de `reply_before` kan men afleiden dat wanneer de taalconstructie ergens aangeroepen wordt de huidige computatie niet wordt stilgelegd totdat de future uiteindelijk geresolved wordt. Om niet te moeten wachten bij elke aanroep van `reply_before` maakt men gebruik van een listener die event-gebaseerd zal handelen bij het resolven van een future. Deze niet-blokkerende eigenschap van de `reply_before` voldoet dus aan de niet-blokkerende communicatie vereisten van AmOP.

Abstractie over tijdelijk verbroken verbindingen. `reply_before` biedt de programmeur de mogelijkheid om een timeout waarde te specifiëren die gebruikt kan worden om failures te gaan detecteren. Het detecteren zal gebeuren door te wachten tot een bepaalde deadline verstreken werd en dit verstrijken dan te bekijken als een gedetecteerde failure. Deze tijdsgebaseerde aanpak heeft als voordeel dat men bepaalde verbroken verbindingen kan gaan negeren en enkel de langdurige disconnecties eruit gaan halen die we wel willen behandelen, niet elke verbroken verbinding zal dus een `reply_before` handler triggeren. Dit komt dus overeen met het vooropgestelde criterium dat er abstractie mogelijk moet zijn over volatiele verbroken verbindingen.

Blokgestructureerde foutafhandeling. Om aan dit criterium te voldoen is de `reply_before` nog niet uitgebreid genoeg. Het is enkel mogelijk om per future een failure handler te definiëren met een bepaalde timeout waarde. Men zou anderzijds wel kunnen voldoen aan dit criterium wanneer de `reply_before` op dezelfde wijze als de `due` zou worden geimplementeerd. Dan zou men bijvoorbeeld een heel stuk code kunnen onderhevig maken aan een bepaalde failure handler. Alle futures die aangemaakt werden in dit stuk code zouden dan onderhevig zijn aan dezelfde timeout en handler. Om nog meer expressiviteit te creëren zou men dan ook meerdere handlers kunnen toelaten met hun eigen selector zoals we al zagen bij de `due` taal constructie. Dus hoewel de huidige implementatie niet aan dit criterium voldoet is er ruimte om in toekomstig werk dit wel mogelijk te maken.

5.4.7 Conclusie

De `reply_before` taalconstructie kan gebruikt worden om aan failure handling te doen in de "Verwerken" fase in de levenscyclus van een bericht. De failure handling kan gebruikt worden om andere future-gebaseerde taaleigenschappen te beinvloeden, bijvoorbeeld wat er besproken werd in de secties over promise pipelining en de interactie met `when`. De evaluatie wees erop dat `reply_before` geschikt is voor gebruik in een AmOP omgeving. Er is echter wel mogelijkheid tot verbetering met het oog op modulaire software ontwikkeling omdat de huidige syntax en semantiek niet expressief genoeg is om aan blokgestructureerde foutafhandeling te doen.

5.5 Conclusie

We hebben in dit hoofdstuk drie verschillende taalconstructies besproken die kunnen gebruikt worden om aan failure handling te doen in de taal AmbientTalk.

`due` wordt gebruikt om aan failure handling te doen op uitgaande berichtgeving. Het geeft de mogelijkheid vervalcondities mee te geven aan asynchrone berichten die eens vervallen dan op een gepaste manier kunnen behandeld worden.

`receive_due_otherwise` zorgt voor failure handling op inkomende berichten. Wanneer bepaalde berichten niet op tijd worden ontvangen wordt er verondersteld een failure gebeurd te zijn; een failure zal dan behandeld worden naar gelang het bericht.

`reply_before` maakt het mogelijk failures te behandelen die kunnen optreden vanaf de verzending tot de ontvangst van antwoord op het bericht. Deze mixin maakt gebruik van de futuresMixin en moeten dus altijd in combinatie met elkaar gebruikt worden.

Na de bespreking van elk van deze taalconstructies werd er een evaluatie gemaakt van de constructies in vergelijking met de criteria die we vooropgesteld hebben in het vorige hoofdstuk.

Hoofdstuk 6

Conclusie

In de afgelopen jaren zijn de mobiele, draadloos met elkaar verbonden toestellen steeds goedkoper en krachtiger geworden waardoor ze de deur hebben opengezet naar een heel nieuw soort van applicaties. Het programmeren van toepassingen voor deze mobiele netwerken vraagt echter een heel ander soort programmeertaal dan het geval is voor stationaire netwerken (oa. het internet). Doordat we te maken hebben met mobiele toestellen die gebruik maken van draadloze verbindingen zullen deze verbindingen frequenter verbroken worden dan in de traditionele stationaire netwerken. Op software-niveau moet kunnen omgegaan worden met zulke verbroken verbindingen. Programmeertalen ontworpen voor stationaire netwerken beschouwen het verbreken van een verbinding vaak als "exceptioneel" en gebruiken hiervoor dus het exception handling mechanisme van de taal om deze fouten op te vangen. We hebben echter aangetoond dat zulk een mechanisme niet meer schaalt in de context van mobiele netwerken. Het is dan ook zo dat de partiële foutafhandeling die aanwezig is in hedendaagse talen niet voldoende werd gevonden om te gebruiken in een AmOP omgeving. Hierdoor is het de bedoeling van deze scriptie taalconstructies voor te stellen die het mogelijk maken partiële foutafhandeling op een gepaste manier te behandelen in de context van een mobiel netwerk. Deze taalconstructies maken het mogelijk een onderscheid te maken tussen verbindingen die slechts tijdelijk verbroken worden en fouten die vermoedelijk van een permanente aard zijn zodat het hier echt nodig is om aan foutafhandeling te doen. In deze scriptie hebben we criteria voorgesteld die het mogelijk maken om taalconstructies te gaan evalueren met betrekking tot hun doeltereffendheid om aan foutafhandeling te doen in mobiele netwerken. Naast het voorstellen van deze criteria worden er ook taalconstructies vooropgesteld die men kan gebruiken om aan foutafhandeling te doen in een mobiele context.

6.1 Contributies

In deze scriptie werden een aantal criteria naar voren gebracht om de doetreffendheid van een taalconstructie voor foutafhandeling in mobiele netwerken te evalueren. We zullen deze criteria voor foutafhandeling in een AmOP omgeving, dewelke bestudeerd werden in hoofdstuk 4, hier kort herhalen.

Asynchroon omgaan met failures is een heel belangrijk criterium voor

foutafhandeling in een mobiele omgeving omdat het heel belangrijk is in het programmeren van toepassingen voor mobiele netwerken dat er zo weinig mogelijk gebruik gemaakt wordt van blokkerende primitieven. Blokkerende foutafhandeling houdt in dat het programma expliciet wacht tot het zeker weet of een bericht al dan niet succesvol werd afgeleverd. Dit houdt een verhoogd risico op zogenaamde deadlocks in en leidt al snel tot applicaties die niet meer kunnen reageren op nieuwe gebeurtenissen omdat ze geblokkeerd zijn. Het is dus heel belangrijk om de niet blokkerende eigenschap van het AmOP paradigma zoals besproken in sectie 3.3.2 ook bij failure handling te gebruiken zodat het normale verloop van de applicatie niet verstoord kan worden. Doordat dit criterium zo belangrijk is wordt het aanzien als onmisbaar in elke taalconstructie die aan foutafhandeling wil doen in een AmOP omgeving.

Abstractie over tijdelijk verbroken verbindingen is essentieel in foutafhandeling voor applicaties die actief zijn in een mobiel netwerk. Eerder werd al aangehaald dat het grote verschil tussen stationaire en mobiele netwerken de frequentie is van het voorkomen van verbroken verbindigen. Verbroken verbindingen komen meer voor in een mobiel netwerk door het gebruik van draadloze verbindingen met een beperkt bereik. Het is echter wel zo dat verbroken verbindingen zichzelf gemakkelijk kunnen herstellen doordat toestellen in een mobiel netwerk terug in elkaars buurt komen zodat de mogelijke failure handling, die dan eventueel al werd uitgevoerd, niet meer nodig was en communicatie tussen deze toestellen terug zijn gewone gang kan gaan. Hierdoor hebben we een manier nodig om deze vluchtig verbroken verbinding te filteren zodat deze niet meer worden aanzien als echte failures en zullen genegeerd worden door de foutafhandelingscode. Abstracties over tijdelijk verbroken verbindingen zijn hierdoor essentieel in taalconstructies voor failure handling in een mobiele omgeving.

Blokgestructureerde foutafhandelingsstrategieën is een minder essentieel criterium dat vooral betrekking heeft op het software-engineering aspect van de taalconstructies. Met dit criterium willen we bekomen dat we bepaalde failure handlers kunnen definiëren op meerdere asynchrone berichten waardoor we de foutafhandeling kunnen structureren zoals bijvoorbeeld in de try-catch uit Java. Dit criterium verandert weinig aan de essentie van een taalconstructie, maar is vooral belangrijk om de taalconstructie te laten schalen naar applicaties waarin meer algemene foutafhandelingsstrategieën van toepassing zijn op grotere stukken code.

Naast deze criteria werden er in deze scriptie ook drie taalconstructies voorgesteld om aan failure handling te kunnen doen in een ambient-georiënteerde omgeving. Deze taalconstructies zijn ontworpen voor en geïmplementeerd in de programmeertaal AmbientTalk, welke in detail besproken werd in hoofdstuk 3. We zullen deze taalconstructies, dewelke in detail werden bestudeerd in hoofdstuk 5, hier nog even kort herhalen samen met een bespreking van het nut van deze constructies aan de hand van de voorheen beschreven criteria.

`due` wordt gebruikt om aan failure handling te doen op de verzonden berichten van een actor. Dit wordt mogelijk gemaakt door aan berichten een bepaalde vervalconditie mee te geven die aangeeft wanneer een bericht

vervalt en moet behandeld worden omwille van een failure. De berichten onderhevig aan een `due` taalconstructie worden geannoteerd met de nodige foutafhandelingsstrategieën waarna mogelijke fouten asynchroon worden afgehandeld. Dit houdt in dat een `due`-blok de voortgang van een actor niet verstoort. De vervalcondities die men kan toevoegen aan de berichten zorgen ervoor dat we abstractie kunnen maken over failures die we willen negeren, zoals bijvoorbeeld kortstondig verbroken verbindingen. De `due` taalconstructie voldoet dus aan de eerste twee criteria hetgeen deze geschikt maakt voor het afhandelen van fouten in een mobiele omgeving. Het is ook mogelijk om een bepaald soort vervalcondities toe te passen op een volledig blok code en dit op een dynamische manier zodat failure handlers op een dynamische manier in plaats van een statische, lexicale manier aan berichten kunnen worden toegekend.

`receive_due_otherwise` zorgt er voor dat we failure handling kunnen doen op inkomende berichten. Wanneer een bepaalde actor niet op tijd een bepaald bericht ontvangt zal het daaruit concluderen dat er een failure is opgetreden en deze afhandelen met de handlers die in de taalconstructies werden gespecificeerd. Ook de `receive_die_otherwise` handelt mogelijke fouten op een asynchrone manier af, zodat de normale computatie in de actor nooit hoeft opgehouden te worden. Dit betekent dat het wachten op een binnenkomend bericht en het eventuele afhandelen van een failure op dit bericht geheel onafhankelijk wordt gedaan van de computatie binnenin de actor. Doordat men de mogelijk heeft om een timeout in milliseconden mee te geven die aanduidt hoe lang er moet gewacht worden op een bepaald bericht, kan de programmeur aanduiden wat moet aanzien worden als een kortstondige, tolereerbare verbroken verbinding en wat als een mogelijk permanente verbroken verbinding. Het is dus duidelijk dat ook deze taalconstructie voldoet aan de twee belangrijkste criteria en daardoor een geschikte taalconstructie is om aan failure handling te doen in een AmOP omgeving. Het is echter niet zo dat de `receive_due_otherwise` voldoet aan het laatste criterium en men dus geen failure handling kan gaan definiëren op code blokken. De taalconstructie biedt echter wildcards aan die toelaten om dezelfde foutafhandelingsstrategie toe te passen op meerdere gelijkaardige berichten.

`reply_before` wordt gebruikt om failures op te vangen bij het verwerken van een bepaald bericht door de actor van ontvangst. Meer concreet zal `reply_before` een deadline vooropstellen voor het antwoord op een asynchroon bericht. Indien de actor van ontvangst dus niet tijdig kan antwoorden dan wordt dit aanzien als een failure. Voor dit reply systeem werd er gebruik gemaakt van zogenaamde futures en men kan zelfs stellen dat de `reply_before` een uitbreiding vormt op de futuresMixin in AmbientTalk. De werking van de `reply_before` is geheel onafhankelijk van de normale computatie van de actor waarbinnen hij opereert. De taalconstructie gaat dus asynchroon om met failures en voldoet al aan het eerste criterium. In de taalconstructie moet men ook een deadline specificeren die aangeeft hoelang er moet gewacht worden op een reply van de actor van bestemming. Deze deadline geeft de mogelijkheid om kortstondige verbroken verbindingen te negeren zodat enkel vermoedelijk permanente failures kunnen behandeld worden door de failure handlers.

De aard van de failure is bij deze taalconstructie heel vaag omdat men als programmeur niet kan weten welk deel van de verzendcyclus gefaald heeft: de verzending van het bericht, het verwerken van het bericht of het verzenden van het resultaat. Hierdoor is het zelfs mogelijk dat er een failure handler opgeroepen wordt wanneer het bericht succesvol verwerkt werd door de actor van bestemming, een failure in het versturen van de reply kan voor dit soort gedrag zorgen. `reply_before` is geen blokgestructureerde foutafhandelingsconstructie. De taalconstructie werkt op het niveau van één asynchroon bericht. Maar doordat de eerste twee criteria wel voldaan waren kunnen we zeggen dat de `reply_before` kan gebruikt worden voor foutafhandeling in een mobiele omgeving.

6.2 Methodologie

Om de experimenten in verband met failure handling te kunnen doen hebben we een taal-gebaseerde aanpak gebruikt. Dit houdt in dat we de foutafhandeling hebben geïmplementeerd als taalconstructies in plaats van een integratie via een bibliotheek of zogenaamde middleware. AmbientTalk biedt hiervoor het geschikte experimentatieplatform aangezien die taal via haar reflectieve eigenschappen toelaat om makkelijk nieuwe taalconstructies te definiëren. In hoofdstukken 3 en 5 zagen we dan ook dat de hierboven beschreven taalconstructies geïmplementeerd zijn als afzonderlijke "taalmixins", dewelke een modulaire reflectieve aanpassing van een actor uitdrukken. Dit maakt het mogelijk specifieke stukken in een programma te beïnvloeden met deze failure handling zonder dat men zich zorgen hoeft te maken voor enige neveneffecten op andere delen van de code. Een voorbeeld hiervan is het overschrijven van de MOP `send` methode in het logging voorbeeld in sectie 3.4.7 waardoor het gedrag van een specifieke actor veranderd werd zonder mogelijk andere actoren aan te tasten.
Een ander voorbeeld van het gebruik van zulke reflectieve taalconstructies is dat deze toelaten om het gedrag van een actor op een lokale manier aan te passen zonder de broncode van de actor aan te moeten tasten. Dit kan door het overschrijven van de meta-methoden zoals `send`. Indien de foutafhandelingsstrategieën zouden zijn aangeboden als een software-bibliotheek, dan zou de broncode manueel moeten aangepast worden opdat de bibliotheek wordt gewaarschuwd over de acties uitgevoerd door de applicatie. Om bijvoorbeeld een bericht te loggen zou men bij een software-bibliotheek manueel bij elk asynchroon bericht de log procedure moeten oproepen. Dit werd al besproken in hoofdstuk 2 bij de taal ProActive, hier moest men oproepen naar de bibliotheek doen in een standaard try-catch expressie om het asynchrone gedrag van ProActive te vereenzelvigen met het synchrone gedrag van een try-catch blok. Het gebruik van de zogenaamde taalmixins in de implementatie van de taalconstructies voor failure handling is dan ook geheel geslaagd als experiment doordat de actoren waarop ze zijn toegepast enkel gebruik hoeven te maken van de taalconstructies. Hun gedrag wordt op een transparante manier aangepast en vereist geen manuele aanpassingen in de broncode buiten het gebruik van de taalconstructies zelf.

6.3 Beperkingen

In deze sectie zullen we de nog aanwezige beperkingen aanhalen in de gegeven taalconstructies, zowel op implementatieniveau als op conceptueel vlak. Naast het aanwijzen van deze beperkingen zullen ook mogelijke verbeteringen aangehaald worden.
In de implementatie van de `due` zitten nog enkele inefficiënt geïmplementeerde delen die verbeterd kunnen worden. Het eerste probleem bevindt zich in de code van die DueMixin die de `createMessage` MOP methode verfijnt, om dit probleem te identificeren zullen we hier de code nog even herhalen.

```
createMessage@args :: {
   origmsg: .createMessage@args;
   if(inDueContext,
      { ...
      });
   ...
};
```

Zoals men kan zien bestaat de `createMessage` uit een if-test die kijkt of het bericht dat aangemaakt moet worden wel of niet binnenin een due code blok zit. Wanneer de DueMixin is toegepast op een bepaalde actor is het dus zo dat elk bericht dat aangemaakt moet worden deze if-test zal uitvoeren. Dit is natuurlijk niet efficiënt en moet verbeterd worden. Dit kan men doen door bijvoorbeeld de `createMessage` binnen een due context te assignen naar de code binnen de if en erbuiten de MOP methode terug te zetten naar de standaard implementatie.
Uit het design van `due` weten we dat deze taalconstructie bestaat uit het gelijke gebruik van twee verschillende mixins. We hebben hierboven al een probleem aangehaald in de DueMixin, maar ook in de ExpiryCheckMixin zit nog een probleem dat de efficiëntie aantast van `due`. De ExpiryCheckMixin maakt momenteel nog gebruik van polling om na te gaan of berichten in de `outbox` van de actor, waarop deze mixin toegepast werd, al dan niet vervallen zijn. Dit pollen gebeurt dan ook op berichten die in de eerste plaats al geen `due` berichten zijn, wat natuurlijk heel inefficiënt is. Dit probleem is op te lossen door gebruik te maken van de `notifier`, die ook gebruikt wordt bij de andere twee taalconstructies. Deze actor zou, met het gebruik van de deadline uit de `due` constructie, het mogelijk maken dat er per `due` bericht één notificatie bericht moet verstuurd worden, wat de zoektocht naar vervallen berichten specifieker en efficiënter maakt.
Het grote nadeel van de `receive_due_otherwise` en de `reply_before` is het feit dat men slechts op één enkel bericht een bepaalde failure handler kan toepassen zodat deze taalconstructies niet kunnen gebruikt worden om foutafhandelingsstrategieën toe te passen op een afgebakend stuk code. Om dit op te lossen zullen we in deze taalconstructies gelijkaardige aanpassingen moeten maken volgens het `due` voorbeeld zodat men via een gelijkaardige syntax een blok code kan onderhevig maken aan een bepaalde strategie.

6.4 Toekomstperspectieven

Hoewel de voorgestelde taalconstructies een degelijke failure handling verzorgen voor de specifieke fases in de transmissiecyclus van een asynchroon bericht, kan men niet zeggen dat deze drie taalconstructies de enige failure handling zijn die men nodig heeft in het programmeren van applicaties in de context van een mobiel netwerk. Denk bijvoorbeeld aan de foutafhandeling die we besproken hebben bij de taal ProActive in sectie 2.3.2. Deze taal bezit twee systemen om te kunnen herstellen van volledige systeemcrashes. Hoewel er aangetoond werd dat deze systemen niet schalen naar het AmOP paradigma is er wel degelijk een systeem nodig dat beschermt tegen dit soort van failures in een mobiele omgeving.

Er zal ook moeten nagegaan worden of de voorgestelde taalconstructies wel degelijk nuttig zijn in het programmeren van echte applicaties in een mobiele omgeving. Zo dient er nagegaan te worden of de voorgestelde taalconstructies makkelijk combineerbaar zijn in een applicatie. Ook is het momenteel onduidelijk of het mechanisme om een onderscheid te maken tussen vluchtige en permanente verbroken verbindigen aan de hand van een timeout periode, voldoende is. Het zou bijvoorbeeld meer zinvol kunnen zijn om een systeem zelf dit onderscheid te laten maken op basis van vergaarde statistieken zoals een gemiddelde verzend- en ontvangsttijd van berichten. Ook zijn er abstracties nodig om na het detecteren van een fout de verzender en ontvanger van het gefaalde bericht terug in een consistente toestand te brengen: weet de ontvanger dat de verzender het bericht als gefaald aanschouwd en vice versa? De hier voorgestelde taalconstructies laten toe te reageren op verbroken verbindingen, maar bieden de programmeur weinig hulp in het correct afhandelen en herstellen van fouten.

6.5 Conclusie

We hebben in deze scriptie kunnen concluderen dat de beschikbare vormen van partiële foutafhandeling in hedendaagse programmeertalen niet schalen naar een AmOP omgeving waardoor het nodig wordt onderzoek te doen naar nieuwe taalconstructies specifiek voor gebruik in mobiele netwerken. In deze scriptie werden vooreerst een aantal criteria vooropgesteld, voortbouwend op het AmOP paradigma, waaraan foutafhandelingsconstructies moeten voldoen om doeltreffend te zijn in een mobiel netwerk. Vervolgens werden in de taal AmbientTalk drie taalconstructies ontworpen en geïmplementeerd die het mogelijk maken te reageren op langdurige verbroken verbindingen. En hoewel het de drie voorgestelde taalconstructies nog ontbreekt aan essentiële eigenschappen (zoals blokstructurering, foutherstel en meer geavanceerde vervalcondities) zijn ze een eerste noodzakelijke stap om te komen tot een sluitend foutafhandelingssysteem voor ambient-georiënteerde software.

Bibliografie

[1] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Wolfgang De Meuter, Theo D'Hondt *AmbientTalk: A Small Reflective Kernel for Programming Mobile Network Applications* Technical Report, VUB-PROG-TR-05-06, Vrije Universiteit Brussel, 2005.

[2] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, W. De Meuter *Ambient-Oriented Programming* In Companion of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. 2005.

[3] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, W. De Meuter *Ambient-Oriented Programming in AmbientTalk* In "Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP), Dave Thomas ed., Lecture Notes in Computer Science, Springer, to appear.", 2006.

[4] Lieberman, H. *Using prototypical objects to implement shared behavior in object-oriented systems* In Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (Portland, Oregon, United States, September 29 - October 02, 1986). N. Meyrowitz, Ed. OOPLSA '86. ACM Press, New York, NY, 214-223.

[5] Elisa Gonzalez Boix, Stijn Mostinckx and Tom Van Cutsem *JavaPic% Evaluator* Technical report,Vrije Universiteit Brussel, November 2003

[6] Theo D'Hondt, Wolfgang De Meuter *The Pico Programming Project* http://pico.vub.ac.be, 1996

[7] Richard Kelsay, William Clinger, and Jonathan Rees *Revised(5) Report on the Algorithmic Language Scheme* http://swiss.csail.mit.edu/ jaffer/r5rs_toc.html

[8] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT press, 1986.

[9] D. Ungar and R. Smith. *Self: The power of simplicity.* In Conference proceedings on Object-oriented Programming Systems, Languages and Applications, pages 227-242. ACM Press, 1987.

[10] Barbara Liskov en Liuba Shrira *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems.* In PLDI, pages 260–267, July 1988.

[11] Barbara Liskov *Distributed programming in argus.* Communications of the ACM, March 1988 Volume 31 Number 3.

[12] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. *The Emerald Programming Language.* DIKU Report no. 87/22, Department of Computer Science, University of Copenhagen, Denmark, October 1987. Also Technical Report 87-10-07, Department of Computer Science, University of Washington.

[13] Black A., Hutchinson N., Jul E., and Levi H.. *Object Structure in the Emerald System.* In OOPSLA'86 Proceedings, pp.78-86, Portland, Oregon, September.

[14] Mark S. Miller, E. Dean Tribble, and Jonathan S. Shapiro. *Concurrency Among Strangers: Programming in E as Plan Coordination* Proc. 2005 Symposium on Trustworthy Global Computing (European Joint Conference on Theory and Practice of Software 2005)

[15] Baude F., Baduel L., Caromel D., Contes A., Huet F., Morel M. and Quilici R. *Programming, Composing, Deploying for the Grid* in "GRID COMPUTING: Software Environments and Tools", Jose C. Cunha and Omer F. Rana (Eds), Springer Verlag, January 2006.

[16] Sun Microsystems. *Java RMI specification* http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html, October 1998.