



Architecturale transformaties ter verbetering van de performantie van object-georiënteerde systemen

Tussentijds verslag IWT specialisatiebeurs.

Tom Tourwé
Laboratorium voor Programmeerkunde
Departement Informatica
Faculteit van de Wetenschappen
Vrije Universiteit Brussel

Inhoudsopgave

1	Oorspronkelijke probleemstelling	1
2	Oorspronkelijke doelstelling	1
3	Onderwerp van de doctoraatsthesis	2
4	Overzicht van de bereikte resultaten in de eerste twee beursjaren	2
4.1	Literatuurstudie	3
4.1.1	Huidige optimalizatietechnieken	3
4.1.2	Transformatiesystemen	5
4.2	Classificatie van object-georiënteerde technieken	6
4.2.1	Programmeren naar interfaces	6
4.2.2	Omvormen van overerving naar aggregatie	6
4.2.3	Objectificatie	7
4.2.4	Compositietechnieken	7
5	Onderzoek verricht in het derde beursjaar	8
5.1	Situering	8
5.2	Annotatie- en transformatietaal	10
5.3	Opzet van de experimenten	12
5.4	Uitdrukken van de ontwerp informatie	12
5.4.1	Beschrijving van een patroon	13
5.4.2	Beschrijving van een instantie van een patroon	13
5.4.3	Beschrijving van de vereisten van een patroon	14
5.5	Transformaties voor ontwerp patronen	16
5.6	Detecteren van evolutieconflicten	17
5.6.1	Nagaan van de vereisten	18
5.6.2	Structurele conflicten	18
5.6.3	Niet toepasbare transformaties	20
5.6.4	Gedragsconflicten	22
5.6.5	Samenvoegconflicten opsporen	22
6	Verder onderzoek	23
7	Conclusie	24
8	Overzicht van de nevenactiviteiten	27
8.1	Academiejaar 1997-1998	27
8.2	Academiejaar 1998-1999	27
8.3	Academiejaar 1999-2000	27

1 Oorspronkelijke probleemstelling

Het origineel projectvoorstel beoogde de optimalizatie van de performantie van object-georiënteerde systemen door middel van architecturale transformaties. Twee observaties lagen aan de basis van het voorstel. Enerzijds bemerken we een verschuiving in de ontwikkeling van object-georiënteerde systemen. De eerste generatie van zulke systemen concentreren zich vooral op hergebruik van code, en gebruiken klassen vooral om data te encapsuleren en berichten om data uit te wisselen en nieuwe data te genereren. Tweede generatie object-georiënteerde systemen daarentegen spitsen zich meer toe op de globale architectuur van het systeem. Zulke systemen worden raamwerken genoemd en pogen om naast hergebruik van code ook hergebruik van het volledige ontwerp van het systeem te bekomen. Dit betekent dat de manier waarop klassen gecombineerd worden, hoe ze samenwerken en welke rol ze spelen zeer belangrijk wordt. Anderzijds kunnen we vaststellen dat huidige vertalers voor object-georiënteerde systemen niet in staat zijn om de performantie van zulke raamwerken significant te optimalizeren. De belangrijkste reden hiervoor is dat zulke vertalers geen notie hebben van de architectuur van een systeem, omdat ze enkel over lokale of globale informatie kunnen beschikken. Raamwerken introduceren echter allerlei abstracties en extra indirecties om de nodige flexibiliteit te bekomen. Deze abstracties en indirecties kunnen niet doorbroken of weggewerkt worden door de huidige vertalers. Dit leidt tot een aanzienlijk verlies in performantie en er is dus nood aan optimalizatietechnieken die specifiek op raamwerken gericht zijn.

2 Oorspronkelijke doelstelling

Het oorspronkelijk doel van het onderzoek was het ontwikkelen van technieken om de performantie van raamwerken significant te optimalizeren. Vermits deze extra indirecties en abstracties introduceren, blijkt het onmogelijk om volkomen automatisch de nodige informatie hiervoor af te leiden. We opteerden er dus voor om de ontwikkelaar zelf belangrijke informatie over het systeem ter beschikking te laten stellen van de vertaler. Ontwikkelaars hebben immers een zeer grote kennis van het raamwerk, en weten perfect op welke plaatsen het ontwerp de performantie beïnvloedt. We voorzien een speciale annotatietaal waarin we de ontwikkelaar toelaten om op een eenvoudige en duidelijke manier deze informatie uit te drukken. Aangenomen dat deze informatie beschikbaar is, moeten we eveneens een manier voorzien waarop de vertaler deze kan aanwenden om de performantie van het raamwerk te optimalizeren. Dit moet gebeuren door het doorbreken van de ingevoerde abstracties en het wegwerken van de extra indirecties daar waar mogelijk. Omdat dit weer niet op een eenvoudige manier kan geautomatiseerd worden, kiezen we ervoor om de ontwikkelaar toe te laten aan te

geven hoe dit dient te gebeuren. We voorzien dus ook een transformatietaal waarin kan uitgedrukt worden hoe de implementatie van een raamwerk kan omgevormd worden naar een efficiëntere implementatie.

Twee punten van deze aanpak verdienen extra aandacht. Vooreerst is het zo dat we moeten nagaan of de beschrijvingen die door de ontwikkelaar gespecificeerd worden in de annotatietaal overeenkomen met het ontwerp en de architectuur van het raamwerk. Indien dit immers niet het geval is zal het uiteindelijke resultaat na optimalizatie niet het correcte gedrag vertonen. Verder is het ook zo dat we willen bekomen dat de optimalizerende transformaties het gedrag van het raamwerk behouden. Dit betekent dat deze transformaties niet zullen uitgevoerd worden in bepaalde omstandigheden. We kunnen ons hiervan verzekeren door met elke transformatie een preconditionie te associëren, die de voorwaarden voor het toepassen van de transformatie uitdrukt.

3 Onderwerp van de doctoraatsthesis

De vorderingen en de resultaten die in het derde jaar van het onderzoek bereikt werden, hebben tot een verbreding van het onderwerp van het doctoraat geleid. De aanpak die voorgesteld werd om de performantie van raamwerken te optimalizeren kan immers in een veel bredere context gebruikt worden om ondersteuning te bieden voor software ontwikkeling gebaseerd op raamwerken. Deze specifieke manier van software ontwikkelen kent immers een aantal belangrijke problemen, die allen aangepakt kunnen worden door het expliciteren van ontwerp informatie en het definiëren van de gepaste transformaties op de implementatie van het raamwerk. Optimalizatie van de implementatie van het raamwerk vormt hier dan een apart onderdeel van. We zullen in latere secties in meer detail bespreken wat dit concreet inhoudt.

4 Overzicht van de bereikte resultaten in de eerste twee beursjaren

Deze sectie geeft een beknopt overzicht van de resultaten bereikt in de eerste twee jaren van het doctoraatsonderzoek. We bespreken deze resultaten niet in detail, maar verwijzen naar de aanvraag voor de tweede termijn waarin een meer gedetailleerde beschrijving wordt gegeven [Tou99].

Om vertrouwd te raken met de verschillende domeinen van het onderzoek werd eerst een literatuurstudie uitgevoerd. Met het oog op het ontwikkelen van de juiste technieken werd daarna onderzocht welke de oorzaken zijn van het performantieprobleem van raamwerken. Tevens werd, op basis van het bovenstaande onderzoek, een formalisme ontwikkeld waarin de door ons voorgestelde oplossing kan in uitgedrukt worden. Dit formalisme bespreken we echter pas in sectie 5.2.

4.1 Literatuurstudie

De literatuurstudie besloeg twee specifieke domeinen. Enerzijds werden huidige vertalers voor object-georiënteerde systemen onder de loep genomen en werden de specifieke technieken die ze gebruiken om de performantie te optimaliseren bestudeerd. Er werd dan nagegaan welke de specifieke redenen zijn waarom deze technieken falen indien ze toegepast worden voor raamwerken. Anderzijds werden bestaande transformatiesystemen bestudeerd, zowel voor object-georiënteerde programmeertalen als voor meer traditionele paradigma's. Hieruit werd een lijst van eigenschappen gedistilleerd waaraan een voor onze doeleinden geschikt transformatiesysteem dient te voldoen.

4.1.1 Huidige optimalizatietechnieken

Huidige optimalizatietechnieken zijn tot stand gekomen in drie fases. In een eerste fase werd getracht om optimalizatietechnieken van traditionele (procedurele) programmeertalen aan te wenden voor object-georiënteerde programmeertalen. Daarna werd ingezien dat deze enkel lokale informatie exploiteerden voor de optimalizaties, en dat dit in een object-georiënteerde omgeving vaak niet optimaal rendeerd. De logische stap was dan ook om globale informatie te gaan gebruiken. Voor echt complexe en flexibele systemen volstaat zelfs dit niet. Daarom werden technieken ontworpen die de tussenkomst van de ontwikkelaar vereisen.

Technieken die lokale informatie gebruiken Zoals reeds gezegd probeerden de eerste vertalers voor object-georiënteerde systemen bestaande optimalizatietechnieken [ASU86] aan te wenden in een object-georiënteerde omgeving. Vertalers zoals [Cha92] probeerden dan ook om bijvoorbeeld *inlining* en *copy propagation* in te bouwen in een object-georiënteerde vertaler. Omdat object-georiënteerde programmeertalen een aantal specifieke eigenschappen hebben die niet aanwezig zijn in traditionele programmeertalen (zoals bijvoorbeeld late binding en polymorfisme), werkten deze technieken echter niet zo goed als verwacht. Het grote obstakel bleek te zijn dat in een object-georiënteerde programmeertaal zeer moeilijk statisch te voorspellen valt welke specifieke methoden er tijdens de uitvoering van een programma opgeroepen worden (het zogenaamde statisch binden van berichten aan methoden). Inlining werd dus zeer moeilijk. Bovendien werd er geobserveerd dat programma's het grootste deel van de uitvoeringstijd spendeerden aan het uitvinden welke methode moest uitgevoerd worden, zodat technieken als copy propagation en soortgelijke slechts een verwaarloosbare winst in performantie opleverden.

Technieken die globale informatie gebruiken Om de geobserveerde problemen te verhelpen werden meer geavanceerde vertalers ontwikkeld die

een aantal nieuwe technieken voor het optimaliseren van de performantie bevatten. Deze nieuwe technieken gaan ervan uit dat globale informatie over het programma beschikbaar is. Concreet komt dit neer op het feit dat de vertaler toegang heeft tot de verschillende klassehierarchieën waaruit het programma bestaat. Hierdoor is expliciet gekend in welke klasse een methode gedefinieerd wordt en in welke van zijn subklassen ze al dan niet overschreven wordt. Deze informatie kan aangewend worden om meer methoden statisch te binden. Immers, als we weten in welke klasse een methode effectief gedefinieerd wordt, dan kunnen we in sommige omstandigheden statisch afleiden wanneer deze zal opgeroepen worden, aan de hand van het statische type van de ontvanger van het bericht.

Hoewel aangetoond is dat zulke technieken betere resultaten bereiken dan technieken die enkel lokale informatie exploiteren [CDG96, DGC95, DDG⁺96], zijn er toch nog een aantal belangrijke nadelen. Eerst en vooral is het zo dat een lokale verandering aan het systeem een impact heeft op de globale situatie. Dit heeft als gevolg dat telkens het hele systeem opnieuw moet vertaald worden als de ontwikkelaar ook maar de kleinste verandering aanbrengt, omdat niet meer voldaan wordt aan globale condities. Verder is het zo dat informatie over de klassehierarchieën niet voldoende is om echt significante optimalizaties door te voeren. Omdat meer methoden statisch gebonden worden zullen andere optimalizaties makkelijker kunnen uitgevoerd worden, maar dit lost nog steeds niet het probleem op van de extra abstracties en indirecties die raamwerken zo moeilijk optimaliseerbaar maken. Hiervoor is immers informatie nodig over de globale structuur van het systeem, over de specifieke interacties en de samenwerking tussen objecten en over de configuratie van objecten en klassen. Deze informatie is echter niet beschikbaar en kan ook zeer moeilijk automatisch afgeleid worden.

Technieken die tussenkomst van de ontwikkelaar vereisen Het gebruiken van zowel lokale als globale informatie over een raamwerk blijkt niet voldoende om de performantie ervan significant te verbeteren. Bovendien is er een limiet aan de hoeveelheid statische informatie die automatisch kan afgeleid worden uit een brontekst [ZC90, Rob99]. Dit heeft ertoe geleid dat een vertaler werd ontworpen waarin de ontwikkelaar zelf informatie kan beschikbaar maken die van nut is voor het optimalizatieproces [VCMC97, JGS93]. De achterliggende idee van deze techniek is dat de ontwikkelaar informatie geeft over de context waarin een specifieke klasse (of methode) zal gebruikt worden, en dat de vertaler dan automatisch de methoden van deze klasse (of de enkele methode) optimalizeert voor deze context. Deze techniek noemt men *programma specializatie*, omdat het programma gespecialiseerd wordt voor de context waarin het gebruikt wordt.

Het probleem van deze aanpak is dat deze zich nog steeds op een te lokaal niveau situeert. Het optimaliseren van een individuele klasse of methode

voor een bepaalde context zorgt er immers wel voor dat de methoden op zich sneller uitvoerbaar zijn, maar lost nog steeds niet de problemen op van raamwerken. De informatie die een ontwikkelaar kan specificeren is immers nog steeds van een te laag niveau.

4.1.2 Transformatiesystemen

In deze tak van het onderzoek werden verschillende soorten van transformatiesystemen bestudeerd. Het meest geavanceerde transformatiesysteem, dat bovendien het dichtst aansluit bij ons onderzoek, is ongetwijfeld het refactoringsysteem. Verder werden ook de meer traditionele herschrijfsystemen op syntaxbomen bestudeerd.

Refactoring Het refactoringsysteem [Opd92] voorziet een aantal transformaties die gebruikt kunnen worden om de kwaliteit van de structuur en het ontwerp van een object-georiëteerd systeem te verbeteren. De basis voor deze transformaties zijn de laag-niveau refactorings, die verantwoordelijk zijn voor het aanbrengen van primitieve veranderingen aan de brontekst van het systeem. Door deze laag-niveau transformaties te combineren worden hoger-niveau transformaties gedefinieerd, die in staat zijn om de volledige structuur en het ontwerp van een systeem aan te passen.

Het probleem van het refactoringsysteem is dat het sterk gekoppeld is aan de Smalltalk programmeertaal. Alle transformaties zijn gedefinieerd in Smalltalk en werken op Smalltalk objecten. Dit maakt het zeer moeilijk voor gebruikers om hun eigen transformaties te definiëren. Verder is het zo dat deze transformaties niet automatisch uitgevoerd worden. Het refactoringsysteem gaat niet automatisch op zoek naar plaatsen in de implementatie waar een zekere refactoring kan toegepast worden. De ontwikkelaar moet zelf aangeven welke transformatie op welke plaats moet worden toegepast. Dit is een gevolg van het feit dat er nergens informatie over de structuur en het ontwerp van het systeem bijgehouden wordt. Alles zit impliciet vervat in het hoofd van de ontwikkelaar die een goede kennis moet hebben van het systeem om de impact van zijn operaties te kunnen inschatten.

Herschrijfsystemen op syntaxbomen De enige andere transformatiesystemen die min of meer gerelateerd zijn aan ons onderzoek zijn herschrijfsystemen op syntaxbomen [CCH95]. Deze laten immers de definitie toe van elke mogelijke transformatie die toegepast kan worden op een syntaxboom. Zulke systemen hebben tevens het voordeel dat ze volledig onafhankelijk zijn van de specifieke programmeertaal waarop ze toegepast dienen te worden. Ze voorzien immers maar twee operaties op deze syntaxbomen: het toevoegen en het verwijderen van een knoop. Andere transformaties worden dan gedefinieerd, eventueel door de gebruiker zelf, in functie van deze twee operaties, of in functie van andere reeds bestaande operaties.

Hoewel ook deze aanpak duidelijk een aantal voordelen biedt, zijn er ook weer inherente nadelen aan verbonden. De transformaties die mogelijk zijn op syntaxbomen zijn over het algemeen van een zeer laag en lokaal niveau. Het is zeer moeilijk om de volledige structuur van een systeem aan te passen door middel van zulke transformaties, omdat dit aanpassingen vraagt op verscheidene plaatsen in het systeem. Een systeem bestaat echter uit een verzameling van verschillende syntaxbomen, die bij deze aanpak allemaal ongerelateerd zijn. Verder is het ook hier weer zo dat er geen expliciete informatie over het systeem beschikbaar is, en dat de ontwikkelaar dus zelf moet aangeven wanneer welke transformatie op welke plaats moet toegepast worden.

4.2 Classificatie van object-georiënteerde technieken

Versillende technieken en richtlijnen voor het schrijven van flexibele software zijn gedocumenteerd in de literatuur. Nergens wordt echter beschreven welke de impact van deze technieken is op de performantie van een systeem. Om technieken te ontwikkelen die de performatie verbeteren is dit echter noodzakelijk. Daarom werden enkele representatieve technieken bestudeerd [Cop92, Bec97, GHJV95] en werd er een classificatie opgesteld waarin alle bestaande technieken kunnen onderverdeeld worden. We bespreken deze classificatie in het kort en verwijzen naar de aanvraag voor de tweede termijn voor een gedetailleerdere beschrijving [Tou99].

4.2.1 Programmeren naar interfaces

Een goed raamwerk biedt de juiste abstracties aan voor het domein waarvoor het opgesteld is en is volledig geïmplementeerd in functie van deze abstracties. Dit maakt het mogelijk om verschillende concrete implementaties te voorzien die allemaal passen binnen dit raamwerk.

De impact op de performantie van deze techniek is niet gering. Binnen een raamwerk zijn immers vaak meerder concretisaties van een enkele abstractie aanwezig. Voor specifieke applicaties wordt vaak maar een van deze concretisaties gebruikt. Omdat het volledige raamwerk echter geïmplementeerd is in functie van de abstractie, en omdat niet expliciet gekend is welke specifieke concretisatie gebruikt wordt, kan er geen efficiënte code gegenereerd worden.

4.2.2 Omvormen van overerving naar aggregatie

Overerving legt de onderlinge relatie tussen klassen vast op een statische manier. Om een flexibel raamwerk te bekomen wordt dan ook vaak de voorkeur gegeven aan aggregatie [JO92]. Dit maakt het immers mogelijk om de concrete relatie tussen instanties van bepaalde klassen tijdens de uitvoering van het programma te veranderen.

Vanuit het oogpunt van performantie is het duidelijk dat een statische relatie te verkiezen valt boven een dynamische. Deze laatste zorgt immers voor meer communicatie tussen de verschillende objecten. Bovendien kunnen de berichten die heen en weer gestuurd worden tussen deze objecten niet statisch gebonden worden, juist omdat de relatie dynamisch kan aangepast worden. In sommige situaties is het nochtans mogelijk om hetzelfde gedrag te bereiken via een statische relatie. De resulterende code is echter moeilijker begrijpbaar en onderhoudbaar. Het zou dus ideaal zijn moest de vertaler de dynamische relatie automatisch kunnen omzetten naar de statische variant. Huidige vertalers voorzien echter geen technieken om dit te bereiken.

4.2.3 Objectificatie

Objecten bezitten een aantal eigenschappen die ze uiterst geschikt maken om flexibele software te implementeren. Enerzijds voorzien ze encapsulatie, waardoor het systeem niet afhankelijk is van de specifieke implementatiedetails van een concept. Anderzijds kunnen objecten gesubstitueerd worden voor andere objecten die dezelfde interface voorzien. Op die manier wordt het mogelijk om het gedrag van een systeem te veranderen door simpelweg een andere implementatie voor dezelfde interface te voorzien. Om een zeer flexibel systeem te bekomen wordt er dus aangeraden om zoveel mogelijk objecten te gebruiken en alle mogelijke concepten voor te stellen als een object.

Voor de performantie van een systeem is deze richtlijn natuurlijk bijzonder nadelig. Vermits objecten encapsulatie promoten, kan een object enkel gebruikt worden door er berichten naar te sturen, hetgeen minder efficiënt is. Bovendien ligt het feit dat objecten gesubstitueerd kunnen worden voor andere objecten met dezelfde interface aan de basis van het feit dat berichten naar dat object niet statisch gebonden kunnen worden. Als zelfs primitieve constructies uit de programmeertaal (zoals de constante *null* [Woo98] of het sturen van berichten [GHJV95]) geobjectifieerd worden, dan is het zeker het geval dat de code minder efficiënt zal zijn. Immers, voor primitieve constructies kan zeer efficiënte code gegenereerd worden, wat niet het geval is voor objecten.

4.2.4 Compositietechnieken

Bovenstaande technieken worden vaak in combinatie gebruikt om een uiterst flexibel systeem te bekomen. Alle ontwerppatronen [GHJV95], bijvoorbeeld, gebruiken het programmeren naar interfaces, promoten het gebruik van aggregatie en passen verschillende objectificaties toe. Ontwerppatronen kunnen dan ook beschouwd worden als geencapsuleerde componenten die een zeker gedrag implementeren en de gepaste interface voor dit gedrag aanbieden. Vermits deze componenten hergebruikt kunnen worden, worden systemen nu vaak geïmplementeerd door het op de juiste manier combineren

ervan. Het voordeel hiervan is dat er verschillende zulke herbruikbare componenten beschikbaar zijn, dat ze meestal apart getest zijn en ook apart kunnen evolueren. Er dient enkel nog code geschreven te worden die de componenten op de gepaste manier laat samenwerken (zogenaamde *glue code*).

Deze nieuwe vorm van programmeren heeft een grote impact op de efficiëntie van een systeem. Eerst en vooral dienen de ter beschikking gestelde componenten zo flexibel mogelijk geschreven te worden. Ze moeten immers kunnen gebruikt worden in verschillende situaties, en het is niet op voorhand geweten welke deze situaties zijn. Verder zorgt de combinatie van verschillende componenten voor nog meer performantieverlies. Zo moeten er vaak nieuwe klassen gedefinieerd worden die dienen om incompatibele interfaces compatibel te maken. Ook dient er data over en weer uitgewisseld te worden tussen verschillende componenten door middel van het over en weer sturen van berichten. Het is daarbij nog mogelijk dat deze data op verschillende manieren wordt voorgesteld. Dit betekent dat er convertiecode moet uitgevoerd worden om te switchen tussen twee formaten telkens er data uitgewisseld wordt.

5 Onderzoek verricht in het derde beursjaar

Na het korte overzicht van het verrichte onderzoek in de eerste twee jaar, bespreken we nu in detail het onderzoek dat uitgevoerd werd in het derde beursjaar. We dienen hierbij te vermelden dat een lichte aanpassing van het originele doctoraatsonderwerp zich opdrong. Na het ontwerpen van de geschikte annotatie- en transformatietaal en het uitvoeren van een aantal experimenten hiermee, werd immers duidelijk dat de oorspronkelijke aanpak (voorgesteld om de performantie van raamwerken te verbeteren), ook toepasbaar was in de veel bredere context van software ontwikkeling door middel van raamwerktechnologie. We schetsen dan ook eerst wat deze specifieke manier van software ontwikkelen inhoudt en welke problemen ermee geassocieerd zijn. Daarna bespreken we in detail de annotatie- en transformatietaal die ontworpen werd en tonen we hoe deze gebruikt kan worden voor het oplossen van de besproken problemen.

5.1 Situering

Software ontwikkeling door middel van raamwerktechnologie biedt een aantal voordelen ten opzichte van de traditionele vorm van ontwikkeling [FS97]. Zo laten raamwerken niet enkel hergebruik van code toe, maar ook van het algemene ontwerp en de architectuur van een systeem. Concrete applicaties kunnen gebouwd worden met een minimum aan inspanning, door het raamwerk op de juiste plaatsen te voorzien van de gepaste code.

Raamwerk-gebaseerde software ontwikkeling heeft echter ook een paar zeer belangrijke nadelen. De voornaamste oorzaak van deze problemen is het

feit dat raamwerken een vorm van *white-box* hergebruik toepassen. Om een concrete applicatie te bekomen dient de ontwikkelaar een zeer gedetailleerde kennis te hebben van de interne structuur en werking van het raamwerk. We bespreken hieronder de twee operaties die typisch uitgevoerd worden op een raamwerk en de problemen die hiermee geassocieerd zijn.

- instantieren van een raamwerk: een concrete applicatie gebouwd door middel van een raamwerk noemt men een *instantie* van het raamwerk. Om een raamwerk op de gepaste manier te instantieren moet een ontwikkelaar nagaan op welke plaatsen hij code die specifiek is voor zijn applicatie moet toevoegen. Concreet komt het erop neer dat hij moet weten welke klassen hij moet toevoegen, welke de superklassen moeten zijn van deze klassen, welke methoden van deze klassen overschreven dienen te worden (en bijgevolg ook welke niet overschreven mogen worden), welke concrete klassen gebruikt dienen te worden en hoe ze moeten gecombineerd worden om het juiste gedrag te implementeren.
- evolueren van een raamwerk: zoals elke software entiteit is een raamwerk onderhevig aan evolutie en dienen er na verloop van tijd een reeks aanpassingen aan de structuur en het gedrag gemaakt te worden. Dit is echter niet eenvoudig. Zulke verregaande aanpassingen zullen immers steeds een impact hebben op het gehele raamwerk, en bijgevolg een reeks andere aanpassingen vereisen. Wanneer een bepaalde functionaliteit uit het raamwerk verwijderd wordt, dient alle code die deze functionaliteit gebruikt aangepast te worden. Bovendien moet nagegaan worden of zulke aanpassing de algemene structuur en het gedrag van het raamwerk niet aantast. Indien bestaande functionaliteit aangepast wordt, dient nagegaan te worden of de code die deze functionaliteit gebruikt nog steeds het vereiste gedrag heeft. Ook het toevoegen van nieuwe functionaliteit kan een impact hebben op het raamwerk. Hoewel nog geen referenties naar deze functionaliteit kunnen bestaan, moet toch nagegaan worden of de implementatie ervan beantwoordt aan de algemene structuur en het ontwerp van het raamwerk.

Het moge duidelijk wezen dat het instantieren en evolueren van een raamwerk geen eenvoudige opgave is. Hiervoor is immers zeer gedetailleerde informatie nodig over de structuur en de interne werking van het raamwerk. Vermits deze informatie vaak impliciet vervat zit in de hoofden van de ontwikkelaars van het raamwerk, is het zeer moeilijke voor andere ontwikkelaars om het raamwerk te hergebruiken. Het werd dan ook al meermaals aangehaald dat softwareontwikkeling door middel van raamwerktechnologie een stijle leercurve heeft [FS97].

Zelfs indien een raamwerk op de correcte wijze geïnstantieerd of geëvolueerd wordt, kunnen er nog andere problemen optreden die specifiek zijn voor het ontwikkelen van software door middel van raamwerktechnologie:

- twee mogelijke evolutiestappen kunnen onafhankelijk van elkaar toegepast worden op een raamwerk door twee verschillende ontwikkelaars. Om terug tot een eenvormig raamwerk te komen dat hergebruik van beide uitbreidingen toelaat, dienen deze samengevoegd te worden in een enkele versie. Dit kan echter leiden tot een aantal conflicten, veroorzaakt door het feit dat beide ontwikkelaar ervan uitgaan dat geen andere aanpassingen aangebracht werden. Zo kunnen bijvoorbeeld beide evoluties dezelfde entiteit aanpassen en bijgevolg is het onduidelijk hoe ze dienen samengevoegd te worden.
- een raamwerk promoot hergebruik van code en structuur, en vormt de basis voor verschillende concrete instanties die zoveel mogelijk functionaliteit van het raamwerk pogen te hergebruiken. Wanneer het raamwerk evolueert en de structuur ervan aangepast wordt, is het onduidelijk wat de impact hiervan is op de concrete instanties. Welke functionaliteit wordt nu aangeboden door het raamwerk en is bijgevolg overbodig in de instanties? Welke functionaliteit is niet meer aanwezig in het raamwerk en dient nu voorzien te worden door deze instanties? Indien een bepaalde functionaliteit van het raamwerk aangepast wordt, wordt deze dan nog op de juiste manier door de instanties hergebruikt? Er bestaat momenteel geen manier om een antwoord te voorzien op deze vragen.

Elk van de vernoemde problemen kunnen we aanpakken door informatie over het ontwerp en de interne structuur van het raamwerk expliciet te maken. Vooreerst laat dit toe om de impact van veranderingen aan het raamwerk na te gaan, omdat wederzijdse afhankelijkheden tussen delen van het raamwerk gedocumenteerd zijn. Verder helpt deze informatie bij het instantieren en evolueren van een raamwerk en kunnen we programmatransformaties voorzien die deze operaties implementeren en deels automatiseren. Het feit dat deze operaties dan ook expliciet voorgesteld zijn zorgt er bovendien voor dat we in staat zijn om evolutieconflicten zoals hierboven beschreven automatisch te detecteren en, in sommige omstandigheden, op te lossen.

In wat volgt bespreken we in detail de annotatie- en transformatietaal die nodig is voor het beschrijven van de informatie en het definiëren van programmatransformaties. We laten daarna zien hoe deze omgeving concreet kan gebruikt worden voor het uitdrukken van belangrijke informatie over een raamwerk, het definiëren van de gepaste transformaties hierop en het detecteren en eventueel oplossen van mogelijke conflicten.

5.2 Annotatie- en transformatietaal

De annotatietaal moet de ontwikkelaar in staat stellen alle mogelijke informatie over het ontwerp van het raamwerk uit te drukken. Bovendien

dient dit te gebeuren op een intuïtieve, begrijpbare en expressieve manier. Vermits het raamwerk evolueert dient het mogelijk te zijn dat er op een eenvoudige manier nieuwe informatie kan toegevoegd worden of bestaande informatie vervangen wordt. Om dit alles te realiseren werd geopteerd om een declaratieve (logische) programmeertaal te gebruiken.

De transformatietaal dient toe te laten dat transformaties op de broncode op een eenvoudige en expressieve manier gedefinieerd kunnen worden. Ze dient dus een geschikt model van het programma aan te bieden en moet de mogelijkheid hebben om de broncode van het programma rechtstreeks aan te passen. Bovendien moeten deze transformaties gecombineerd kunnen worden tot hoger-niveau transformaties die een hele reeks aanpassingen tegelijkertijd toepassen. Om deze redenen werd gekozen om een functionele taal te gebruiken die toelaat om transformaties te definiëren als functies die werken op een syntaxboom-representatie van de broncode. Een belangrijke extra vereiste is dat de transformatietaal in staat moet zijn om informatie over het raamwerk ten allen tijde op te vragen. De annotatie- en transformatietaal dienen dus sterk geïntegreerd te zijn.

Om aan al deze vereisten te voldoen werd een nieuwe programmeertaal ontwikkeld, gebaseerd op de Scheme programmeertaal [AS84], die een combinatie vormt van een logische en een functionele programmeertaal. Deze taal laat dus toe om zowel functies als (logische) relaties te definiëren. Functies dienen om bepaalde operaties te implementeren, en krijgen een aantal input parameters mee en geven een zeker resultaat terug. Relaties worden gedefinieerd door middel van logische feiten en regels. De speciale **query**-functie vormt de eigenlijke brug tussen de twee paradigma's en wordt gebruikt om informatie uit de logische databank te halen en te gebruiken voor de vereiste berekeningen. Verder heeft de programmeertaal toegang tot de brontekst van het programma (via een speciale primitieve constructie, zie [Wuy01] voor meer uitleg hierover) en voorziet ze een syntaxboom representatie ervan zodat deze brontekst op een eenvoudige manier kan gemanipuleerd worden door middel van functies.

Het integreren van het logische en het functionele paradigma is natuurlijk een technologische uitdaging en zorgt voor een aantal fundamentele problemen. Een functie in een normale programmeertaal kan maar een resultaat teruggeven, terwijl de **query**-functie meerdere resultaten kan teruggeven, net als alle functies die in termen van deze functie gedefinieerd zijn. Verder vereist een functieoproep dat er een actueel argument wordt gespecificeerd voor elke formele parameter uit de functiedefinitie. Bij een *query* is dit niet het geval: bepaalde argumenten kunnen oningevuld meegegeven worden, de waarden voor deze variabelen worden dan door de *query* zelf ingevuld. Het is duidelijk dat dit voor problemen zorgt, omdat een functie nu kan gedefinieerd zijn door middel van een *query* (en dus meerdere resultaten zou kunnen teruggeven) en een *query* gedefinieerd kan worden in termen van een bepaalde functie (waarbij alle actuele parameters dus dienen ingevuld

te zijn). Voor elk van deze problemen werd een oplossing gezocht, welke hier niet in detail besproken wordt omwille van de techniciteit ervan en omwille van plaatsgebrek. In de doctoraatscriptie zal hier echter een apart hoofdstuk aan gewijd worden, gezien het combineren van een functionele en logische programmeertaal een aparte bijdrage van het doctoraatsonderzoek is.

Hoe de annotatie- en transformatietaal concreet gebruikt wordt voor het uitdrukken van ontwerpinformatie en bijhorende transformaties wordt in de volgende secties uit de doeken gedaan.

5.3 Opzet van de experimenten

Om de algemene toepasbaarheid van de gekozen aanpak te valideren werden een aantal experimenten met ontwerppatronen (eng. *design patterns*) uitgevoerd [GHJV95]. Zulke patronen beschrijven succesvolle en veel gebruikte oplossingen voor vaak weerkerende problemen. Deze oplossingen introduceren een zekere micro-architectuur in het raamwerk en zijn erop gericht deze zo flexibel en herbruikbaar mogelijk te maken. Kwaliteitsvolle raamwerken gebruiken dan ook vaak een combinatie van zulke patronen om de gewenste flexibiliteit te verkrijgen. Het is bijgevolg zeer interessant na te gaan hoe zulke patronen kunnen beschreven worden in het voorgestelde formalisme en hoe dit ons kan helpen om instantiatie en evolutie van het raamwerk te ondersteunen.

De oplossing die een patroon voorstelt wordt beschreven aan de hand van de structuur en de samenwerking tussen verschillende klassen. Deze informatie dient dus te worden neergeschreven in het voorgestelde formalisme. Instantiatie van het raamwerk gebeurt dan door de implementatie van patronen uit te breiden met het gewenste gedrag. Evolutie van het raamwerk gebeurt door het evolueren van de patronen, met andere woorden, door het aanpassen, toevoegen of verwijderen van klassen en de interacties ertussen.

Vermits patronen de structuur en de interacties tussen klassen definiëren, leggen ze bepaalde vereisten op aan hun implementatie. Wanneer veranderingen aangebracht worden aan het patroon dient nagegaan of de implementatie ervan nog steeds aan de vooropgestelde eisen voldoet. Verder is het zo dat een bepaald patroon maar op een beperkt aantal manieren kan aangepast worden. Dit betekent dat we een aantal standaard transformaties op een patroon kunnen definiëren, die specifiek zijn voor dat patroon, en die toelaten om het patroon semi-automatisch aan te passen wanneer het geïnstantieerd of geevolueerd dient te worden.

5.4 Uitdrukken van de ontwerpinformatie

Patronen bestaan uit een verzameling participanten en beschrijven de specifieke interacties tussen deze participanten. Door aan te geven welke klassen

en methoden de rol van welke participant spelen in een zeker patroon, beschrijven we dat patroon volledig. Om zulk een beschrijving te realiseren gebruiken we de logische feiten uit ons formalisme. Om de interacties tussen de verschillende participanten te beschrijven gebruiken we logische regels. Deze gaan na of de participanten wel voldoen aan de vereisten van het patroon, door de specifieke implementatie ervan te raadplegen. Deze regels kunnen bovendien hergebruikt worden wanneer we dienen na te gaan of de implementatie van een patroon nog correct is na een zekere transformatie. We beschrijven dit alles nu in meer detail.

5.4.1 Beschrijving van een patroon

Het logische predikaat *pattern* wordt gebruikt om een ontwerppatroon voor te stellen. Het geeft aan met welk soort patroon we te maken hebben (b.v. *composite*, *visitor*, *abstractfactory*, etc) en voorziet een verzameling van rollen die moeten aanwezig zijn in een concrete implementatie van dat patroon. Het *abstractfactory* patroon [GHJV95] kan bijvoorbeeld als volgt beschreven worden:

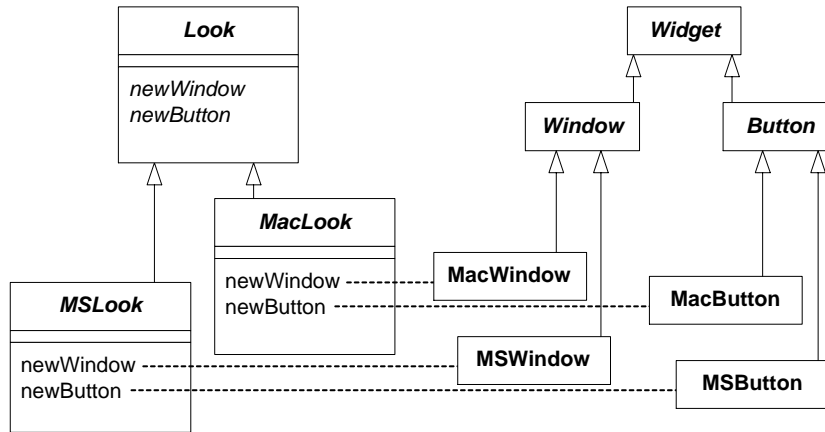
```
(assert pattern
  (abstractFactory
    (abstractFactory concreteFactory genericProduct
      abstractProduct concreteProduct abstractRelation
        concreteRelation abstractFactoryMethod concreteFactoryMethod)))
```

Omdat er natuurlijk verschillende manieren bestaan om een patroon te instantiëren en te implementeren, is het niet de bedoeling om een volledige formele beschrijving ervan te geven. We verkiezen eerder een pragmatische aanpak, waarbij we mogelijke varianten van een patroon apart beschrijven, en dan nog enkel voor deze patronen die effectief gebruikt worden in het raamwerk. Indien het nodig blijkt laat ons formalisme toe om nieuwe varianten toe te voegen.

5.4.2 Beschrijving van een instantie van een patroon

Een instantie van een patroon wordt volledig bepaald door een unieke naam, het soort van patroon en een verzameling van concrete participanten geassocieerd met elke rol in het patroon. Beschouw bijvoorbeeld het ontwerp van een instantie van het *abstractfactory* patroon afgebeeld in Figuur 1. Het bevat een aantal *widgets*, zoals *Window* en *Button*, en een aantal *Looks*, zoals *MSLook* en *MacLook*. Elke concrete *Look* klasse creëert zijn eigen specifieke widgets door middel van *factory* methoden zoals *newWindow* en *newButton*. De specificatie van deze instantie van het patroon ziet er als volgt uit:

```
(assert patternInstance (AF1 abstractFactory))
(assert role (AF1 abstractFactory Look))
```



Figuur 1: Instantie AF1 van het *abstractfactory* patroon

```
(assert role (AF1 concreteFactory MSLook))
(assert role (AF1 concreteFactory MacLook))
(assert role (AF1 genericProduct Widget))
(assert role (AF1 abstractProduct Window))
(assert role (AF1 abstractProduct Button))
(assert role (AF1 concreteProduct (MSWindow Window)))
(assert role (AF1 concreteProduct (MSButton Button)))
(assert role (AF1 concreteProduct (MacWindow Window)))
(assert role (AF1 concreteProduct (MacButton Button)))
(assert role (AF1 abstractRelation (Look Window)))
(assert role (AF1 abstractRelation (Look Button)))
(assert role (AF1 concreteRelation (MSLook MSWindow)))
(assert role (AF1 concreteRelation (MacLook MacWindow)))
(assert role (AF1 concreteRelation (MSLook MSButton)))
(assert role (AF1 concreteRelation (MacLook MacButton)))
(assert role (AF1 abstractFactoryMethod (newWindow Look)))
(assert role (AF1 abstractFactoryMethod (newButton Look)))
(assert role (AF1 concreteFactoryMethod (newWindow MSLook)))
(assert role (AF1 concreteFactoryMethod (newWindow MacLook)))
(assert role (AF1 concreteFactoryMethod (newButton MSLook)))
(assert role (AF1 concreteFactoryMethod (newButton MacLook)))
```

5.4.3 Beschrijving van de vereisten van een patroon

Zoals reeds vermeld leggen patronen vereisten op aan hun implementatie. Om deze vereisten formeel vast te leggen, gebruiken we het `patternConstraint` predikaat. Twee vereisten van het *abstractfactory* patroon zijn bijvoorbeeld de volgende:

1. alle klassen die de rol van `concreteFactory` spelen dienen een (indirecte) subklasse te zijn van de klasse die de rol van `abstractFactory` speelt en alle concrete subklassen van een klasse die de rol van `abstractFactory` speelt dienen de rol van `concreteFactory` te spelen.
2. alle *factory* methoden in een klasse die de rol speelt van `abstractFactory` dienen abstract te zijn en dienen overschreven te worden door concrete methoden in alle klassen die de rol spelen van `concreteFactory`.

Deze vereisten kunnen als volgt neergeschreven worden ¹:

```
(definereel (patternConstraint ?I abstractFactory)
  (exists ?AF
    (role ?I abstractFactory ?AF)
    (land (forall ?CF
      (role ?I concreteFactory ?CF)
      (hierarchy ?AF ?CF))
      (forall ?C
        (concreteSubclass ?AF ?C)
        (role ?I concreteFactory ?C))))))

(definereel (patternConstraint ?I abstractFactory)
  (forall (?M ?AF)
    (role ?I abstractFactoryMethod (?M ?AF))
    (land (role ?I abstractFactory ?AF)
      (abstractMethod ?M ?AF)
      (forall ?CF
        (role ?I concreteFactory ?CF)
        (land (role ?I concreteFactoryMethod (?M ?CF))
          (concreteMethod ?M ?CF))))))
```

Het *abstractfactory* patroon legt nog een aantal andere vereisten op, die we hier niet vermelden, maar die op eenzelfde manier kunnen gedefinieerd worden.

Om de vereisten van een patroon te kunnen nagaan moeten we in staat zijn om de nodige informatie uit de brontekst van het systeem te halen en voor te stellen in het gebruikte medium. Zoals reeds besproken voorziet de ontwikkelde programmeertaal een speciale primitieve hiervoor, wat toelaat om laag-niveau logische predicaten zoals `concreteMethod` en `concreteSubclass` te definiëren in functie van de implementatie van het systeem.

¹Logische variabelen worden voorafgegaan door een vraagteken, terwijl alle andere identifiers constanten voorstellen. Voor een gedetailleerde beschrijving van de syntax van de logische taal SOUL, waarop ons formalisme gebaseerd is, verwijzen we naar [Wuy01]

5.5 Transformaties voor ontwerppatronen

De instantiatie en evolutie van een raamwerk stellen we voor door de instantiatie en evolutie van de ontwerppatronen die in het raamwerk gebruikt worden. Deze operaties op patronen kunnen voorgesteld worden als programmatransformaties. Bovendien kan een bepaald ontwerppatroon maar op een bepaald aantal manieren aangepast worden. We kunnen dus voor elk patroon een aantal transformaties definiëren waarmee we de implementatie van dat patroon aanpassen en dus tegelijkertijd het raamwerk instantiëren of evolueren.

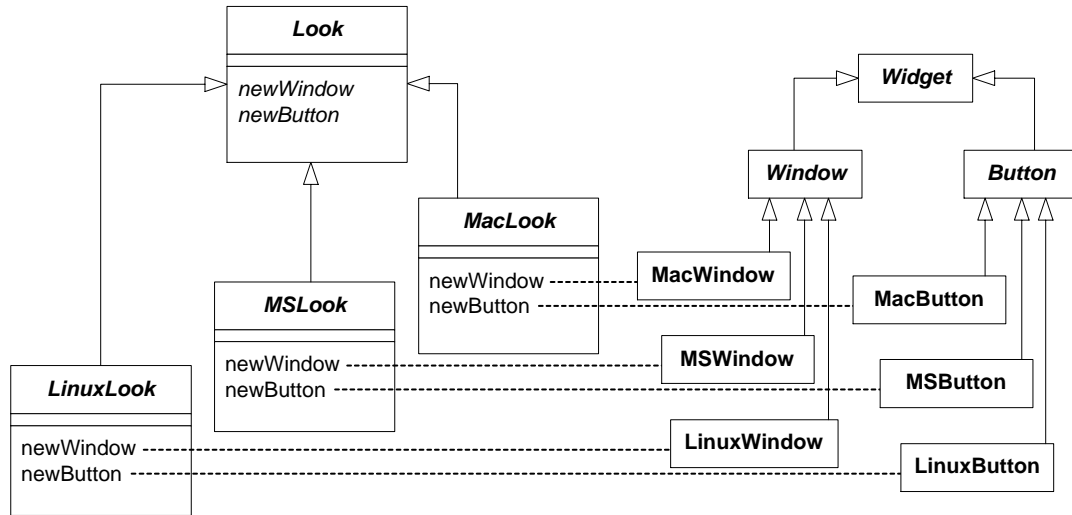
Als een concreet voorbeeld beschouwen we het toevoegen van een concrete *factory* klasse aan het *abstractfactory* patroon. Deze transformatie kan als volgt gedefinieerd worden:

```
(define (addConcreteFactory instance concreteFactoryName)
  (assert role (instance concreteFactory concreteFactoryName))
  (query
    (lambda (abstractFactory)
      (addClass concreteFactoryName abstractFactory))
    (role instance abstractFactory ?abstractFactory))
  (query
    (lambda (abstractProduct)
      (define concreteProduct
        (askUser 'Name of the concrete product created?'))
      (assert role (instance concreteProduct
        (concreteProduct abstractProduct)))
      (assert role (instance concreteRelation
        (concreteFactoryName concreteProduct)))
      (addClass concreteProduct abstractProduct))
    (role instance abstractProduct ?abstractProduct)))
```

Het oproepen van deze functie kan bijvoorbeeld als volgt gebeuren:

```
(addConcreteFactory AF1 LinuxLook)
```

Dit heeft als gevolg dat informatie over de AF1 instantie van het *abstract-factory* patroon zal opgevraagd worden uit de logische databank door middel van het oproepen van de `query`-functie. Zo wordt nagegaan welke klasse de *abstractFactory* rol speelt in deze instantie, en wordt een nieuwe subklasse `LinuxLook` van deze klasse aangemaakt. Dan wordt opgevraagd welke producten de concrete factories in deze patrooninstantie moeten aanmaken. Er wordt aan de gebruiker gevraagd welke concrete producten overeenkomen met deze nieuwe concrete *factory*, en de corresponderende klassen worden dan meteen gegenereerd. Het uiteindelijke resultaat van deze operatie wordt getoond in Figuur 2. Uit dit eenvoudige voorbeeld blijkt onmiddellijk dat informatie over het ontwerp van een raamwerk kan bijdragen in het evolueren en instantiëren ervan. Door middel van deze informatie kunnen



Figuur 2: Structuur van de AF1 instantie van het *abstractfactory* patroon na het toevoegen van een **LinuxLook** factory

immers transformaties gedefinieerd worden die een deel van het werk van de ontwikkelaar overnemen en hem bovendien helpen in het implementeren van de vereiste functionaliteit.

Voor het instantiëren van een raamwerk is het nuttig dat de specifieke patroontransformaties die hiervoor nodig zijn gecombineerd worden tot hoger-niveau transformaties. De instantiatie van een raamwerk vertrekt immers meestal vanuit een hoog niveau beschrijving die dient vertaald te worden in de concrete aanpassingen die dienen te gebeuren. Vermits instantiatie vaak voorkomt dient deze vertaling telkens opnieuw te gebeuren en is het nuttig van deze te kunnen hergebruiken. Onze aanpak laat toe om zulke generische hoog niveau transformaties te definiëren vertrekkende vanuit lager niveau transformaties. Het evolueren van een raamwerk daarentegen vereist niet dat de specifieke transformaties van de ontwerppatronen gecombineerd worden. Een bepaalde evolutie van een raamwerk is immers iets wat typisch maar een keer uitgevoerd wordt en dient bijgevolg niet hergebruikt te worden.

5.6 Detecteren van evolutieconflicten

Deze sectie bespreekt hoe onze aanpak toelaat om de evolutieconflicten op te sporen die kunnen optreden wanneer twee onafhankelijke evolutiestappen dienen samengevoegd te worden tot een enkele versie. We doen dit aan de hand van een eenvoudig voorbeeld dat telkens opnieuw geevolueerd zal worden.

5.6.1 Nagaan van de vereisten

Ontwikkelaars kunnen software op twee mogelijke manieren aanpassen: ze kunnen de veranderingen handmatig doorvoeren of ze kunnen gebruik maken van de specifieke transformaties verbonden aan een ontwerppatroon (zoals de `addConcreteFactory` transformatie voor het *abstractfactory* patroon). Wanneer veranderingen handmatig worden aangebracht, is de ontwikkelaar er zich vaak niet van bewust welke impact zijn veranderingen hebben op de implementatie van het patroon. Bijgevolg kunnen de volgende conflicten optreden:

- door het verwijderen van een bepaalde software entiteit (zoals een klasse, een methode of een variabele) die een zekere rol speelt in een patroon kan de ontwikkelaar de structuur van het patroon ongewild doorbreken of het gedrag ervan beïnvloeden.
- het toevoegen van een participant aan een bestaand patroon kan andere aanpassingen vereisen om de structuur van het patroon af te maken. Indien dit niet gebeurt voldoet het patroon niet meer aan de vooropgestelde eisen.

Dankzij de expliciete representatie van een patroon en zijn vereisten, kunnen deze conflicten tot op zekere hoogte gedetecteerd worden. Dit wordt bereikt door na te gaan of de implementatie nog steeds voldoet aan de vereisten van het patroon. Veronderstel bijvoorbeeld dat een ontwikkelaar een concrete `LinuxLook` klasse manueel toevoegt, maar vergeet om de concrete *factory* methoden `newWindow` en `newButton` te implementeren. Deze inconsistentie kan automatisch gedetecteerd worden door de vereisten van het patroon na te gaan. Meer bepaald, de vereiste dat alle concrete subklassen van de abstracte *factory* `Look` concrete *factories* zijn is niet voldaan, omdat de `LinuxLook` klasse niet als een concrete *factory* geregistreerd is. Zelfs indien dit het geval was, dan nog zou de vereiste dat elke abstracte *factory* methode overschreven moet worden door een concrete *factory* methode in een concrete *factory* niet voldaan zijn.

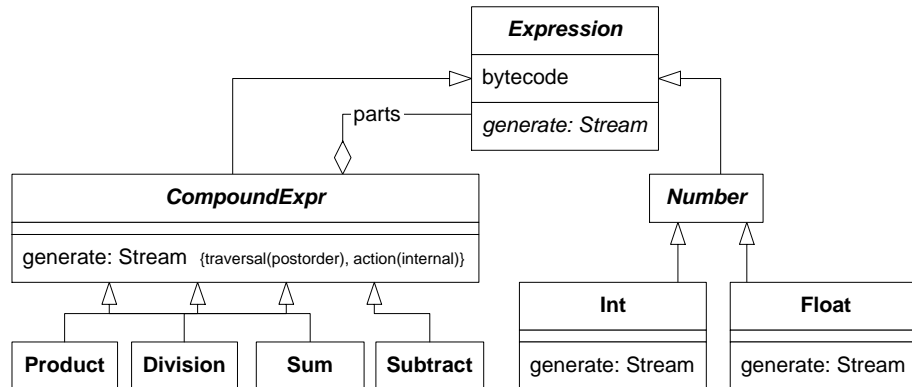
Dergelijke problemen kunnen voorkomen worden door gebruik te maken van de specifieke transformaties die gedefinieerd zijn voor het *abstractfactory* patroon. Zulke transformaties voeren immers automatisch alle nodige aanpassingen uit. Het blijft echter nog steeds mogelijk om andere soorten van conflicten te veroorzaken, zoals we verderop zullen zien.

5.6.2 Structurele conflicten

Wanneer twee specifiek patroontransformaties onafhankelijk worden doorgevoerd op dezelfde patrooninstantie moeten de resultaten van beide transformaties samengevoegd worden. Dit kan echter aanleiding geven tot bepaalde

gedragsconflicten, wat betekent dat de samengevoegde versie zich op onvoorziene en ongewenste manieren gedraagt. Dit zal in een volgende sectie in meer detail besproken worden. Echter, het kan ook voorkomen dat de samengevoegde versie zich wel gedraagt zoals verwacht, maar dat er structurele inconsistenties ontstaan. Hoewel deze conflicten het gedrag van het systeem niet beïnvloeden, is het toch belangrijk van ze te kunnen detecteren. Zulke conflicten resulteren immers vaak in een systeem dat steeds verder afwijkt van zijn oorspronkelijk ontwerp, met alle problemen vandien.

Figuur 3 illustreert een dergelijk conflict. Deze toont de structuur van

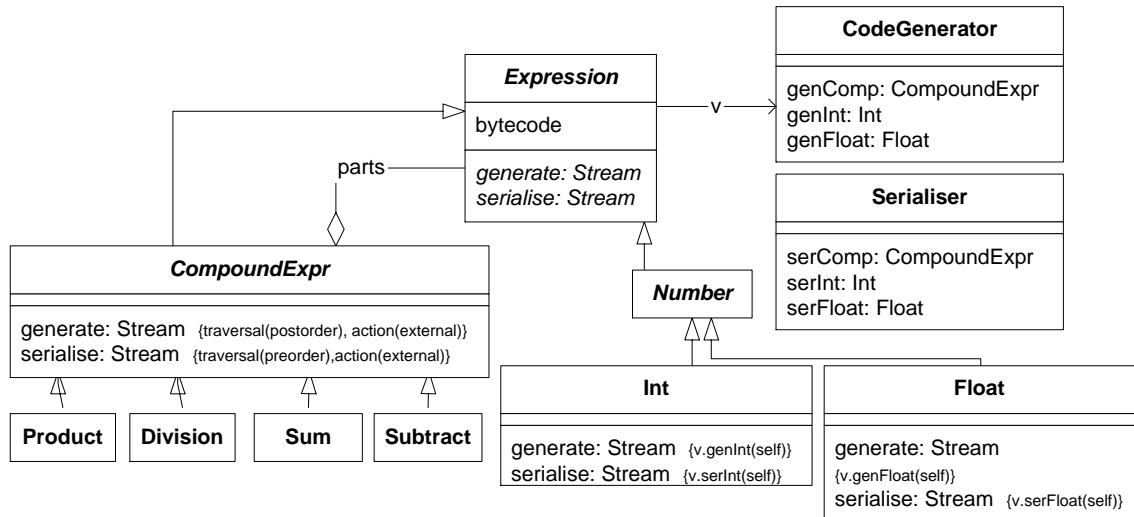


Figuur 3: Structuur van het *composite* patroon voor de **Expression** hiërarchie

een syntaxboom voor wiskundige expressies. Om code te generen voor zulke expressies is er een **generate** methode aanwezig, die geïmplementeerd is volgens het *composite* patroon. De **generate** methode van de **CompoundExpr** klasse zal zijn deelstructuren overlopen in een *postorder* volgorde en een zekere actie uitvoeren.

Beschouwen we als eerste evolutiestap het toepassen van de specifieke patroontransformatie **replaceMethodWithClass**, die de functionaliteit van het codegenereren uit de expressiehiërarchie extraheert en in een aparte **CodeGenerator** klasse onderbrengt. Parallel hiermee voeren we de **addCompositeMethod** patroontransformatie toe, die een nieuwe composite methode **serialize** toevoegt die de structuur op een preorder manier affloopt om de expressie te serialiseren naar een *output stream* (een bestand of het netwerk bijvoorbeeld). We kunnen deze twee evolutiestappen samenvoegen door simpelweg de ene toe te passen na de andere, nadat we er ons van vergewist hebben dat ze niet tot een gedragsconflict leiden. In dit specifieke geval komt er echter een structureel conflict aan het licht als we de twee transformaties trachten samen te voegen. De composite methode **generate** externalizeert zijn actie in de **CodeGenerator** klasse, terwijl de nieuwe composite methode **serialize** geïntroduceerd wordt met een interne actie. Bijgevolg gedragen

verschillende composite methoden zich structureel op een andere manier, wat waarschijnlijk niet de bedoeling is. Een manier om dit probleem op te lossen is om het gedrag van de `serialize` methode ook te externaliseren in een `Serializer` klasse door gebruik te maken van de `replaceMethodWithClass` transformatie. Het uiteindelijke resultaat wordt afgebeeld in Figuur 4.



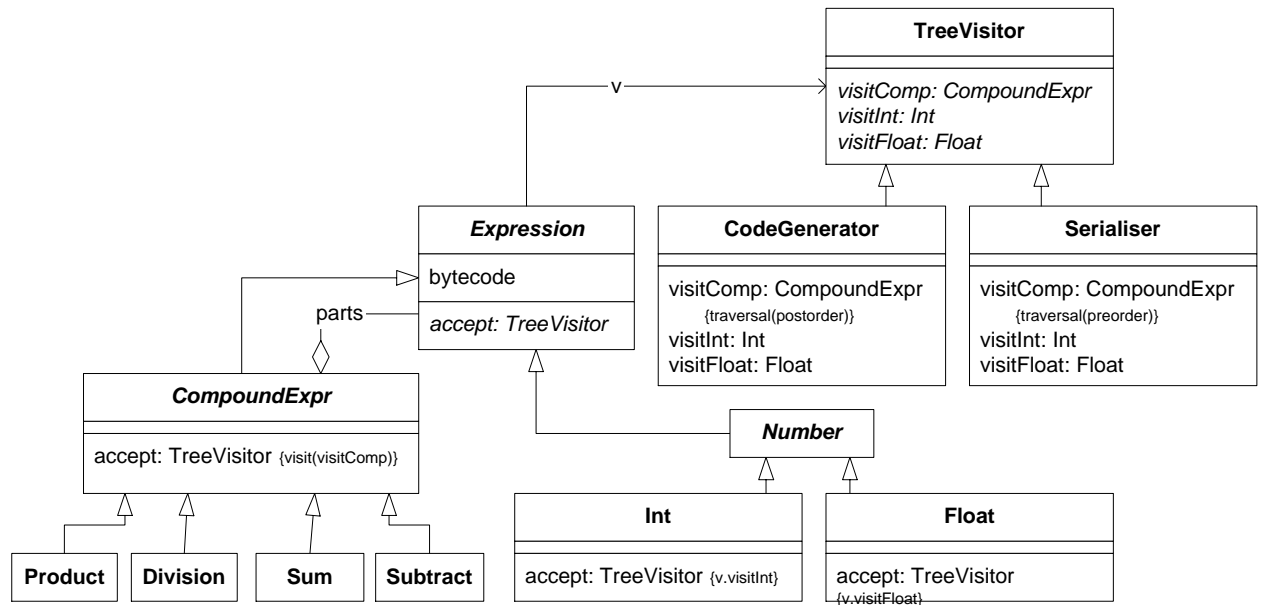
Figuur 4: Structuur van het *composite* patroon na het uitfactoriseren van het gedrag van `generate` en `serialize` methoden.

5.6.3 Niet toepasbare transformaties

Een tweede soort van evolutieconflict dat kan optreden is dat een zekere transformatie niet kan toegepast worden omdat een van zijn precondities niet geldt. Om dit te illustreren grijpen we terug naar de geexternaliseerde `generate` en `serialize` methode uit het *composite* patroon. We willen de structuur hiervan omvormen naar een volwaardige structuur voor het *visitor* patroon, wat kan gebeuren in drie stappen:

- we passen de transformatie (`extractSuperclass (CodeGenerator Serialiser) TreeVisitor`) toe om een `TreeVisitor` superklasse te definiëren voor de `CodeGenerator` en `Serialiser` klassen.
- we passen een `pullUpMethod` transformatie toe om een uniforme interface te voorzien voor de `TreeVisitor` klasse en zijn subklassen.
- we passen de `generaliseMethods` transformatie toe om de `generate` en de `serialize` methode in de `Expression` hiërarchie te generaliseren naar een algemene `accept` methode.

Vooraleer we de `generaliseMethods` transformatie kunnen toepassen dienen echter de bijhorende precondities hiervoor gecheckt te worden. Deze precondities specificeren dat alle methoden die moeten gegeneraliseerd worden dezelfde gedragseigenschappen moeten hebben. In het geval van de composite methoden betekent dit dat hetzelfde traversalgoritme dient gebruikt te worden en dat alle acties extern moeten zijn. Dit kan op een eenvoudige manier nagegaan worden, omdat alle informatie hiervoor expliciet aanwezig is in de specificatie van het patroon. Zo kunnen we zien dat in dit specifieke geval niet aan de preconditie voldaan wordt omdat de `generate` methode een postorder traversal implementeert, terwijl de `serialize` methode een preorder traversal gebruikt. Bijgevolg is het dus onmogelijk om hieruit een algemene `accept` methode te construeren. Om dit probleem aan te pakken moeten we de verantwoordelijkheid voor het overlopen van de elementen van een samengesteld object verhuizen van het samengesteld object zelf naar het *visitor* object. Dit is typisch een manueel proces, aangezien het bijzonder moeilijk is om het traversalgedrag automatisch uit een methode te extraheren. Pas nadat het systeem manueel op deze wijze aangepast is, wordt het mogelijk om de `generaliseMethods` transformatie toe te passen om een generische `accept` methode te creëren. Het uiteindelijke resultaat wordt getoond in Figuur 5.



Figuur 5: Structuur van het *visitor* patroon voor de `Expression` hiërarchie

5.6.4 Gedragsconflicten

Buiten structurele conflicten kan het samenvoegen van twee onafhankelijke patroonevoluties ook aanleiding geven tot gedragsconflicten. Zulke conflicten worden typisch veroorzaakt door twee onafhankelijke patroontransformaties die worden toegepast op dezelfde participant in het patroon. Deze situatie komt vaak voor, omdat participanten vaak een rol spelen in meer dan een patroon. We illustreren dit probleem aan de hand van een voorbeeld.

In het *visitor* voorbeeld van Figuur 5 kunnen we een extra `TypeCheckingVisitor` toevoegen door middel van de `addConcreteVisitor` patroontransformatie die gedefinieerd is voor het *visitor* patroon. We dienen echter de `accept` method van de `CompoundExpr` klasse te overschrijven in de `Division` subklasse. De reden is dat het type van een samengestelde expressie in het algemeen overeenkomt met het meest algemene type van al zijn argumenten (zo resulteert het optellen van een geheel getal bij een reeel getal in een reeel getal). Dit geldt echter niet voor de deling, waarbij het resultaat-type van de deling van twee gehele getallen een reeel getal kan zijn.

Een andere patroontransformatie die kan toegepast worden is de factorisatie van het traversalalgoritme. Nu is het immers zo dat dit algoritme hard gecodeerd zit in de *visitor* hiërarchie, wat hergebruik bemoeilijkt. We willen dit algoritme dus onderbrengen in een instantie van het *strategy* patroon, zodat we het kunnen variëren en hergebruiken. Dit kunnen we bekomen door een `createTraversal` methode toe te voegen aan de *visitor* hiërarchie die verantwoordelijk is voor het instantiëren van de juiste traversalstrategie voor de specifieke visitor. We moeten dan ook de `accept` methode uit de `CompoundExpr` klasse aanpassen zodat deze nieuwe methode ook effectief wordt opgeroepen.

Wanneer we deze twee onafhankelijke patroontransformaties proberen samen te voegen ontstaat er een zeer subtiel gedragsconflict. Het uitfactoriseren van het traversalalgoritme verandert de `accept` methode van de `CompoundExpr` klasse, terwijl het toevoegen van de `TypeCheckingVisitor` vereist dat de `accept` methode wordt overschreven in de `Division` subklasse. De implementatie van deze laatste zal dus niet correct zijn in de samengevoegde versie, omdat ze de `createTraversal` methode niet oproept.

5.6.5 Samenvoegconflicten opsporen

Structurele en gedragsconflicten worden allebei veroorzaakt door het samenvoegen van twee onafhankelijke transformaties en kunnen dus onmogelijk gedecteerd worden door simpelweg nagaan of de vereisten van een patroon nog gelden of door middel van precondities van de transformaties. De voornaamste reden is dat zulke conflicten zich op een ander niveau situeren: twee entiteiten die op een of andere manier gerelateerd zijn worden aangepast in parallel. Toch zijn we in staat om zulke conflicten op te sporen als we gebruik

maken van conflict tabellen zoals oorspronkelijk voorgesteld in [SLMD96]. Zulke tabellen laten toe om verschillende transformaties te vergelijken, en om de voorwaarden te specificeren onder dewelke ze tot een conflict leiden indien ze in parallel toegepast worden. Het verschil tussen onze aanpak en de vorige is dat wij conflicten kunnen detecteren op een hoger niveau, omdat de conflicten kunnen gerelateerd worden aan de specifieke patrooninstanties in dewelke ze voorkomen.

6 Verder onderzoek

Een probleem dat voorkomt bij het ontwikkelen van software door middel van raamwerktechnologie hebben we nog niet besproken: hoe kunnen we de impact van de evolutie van het raamwerk op zijn instanties detecteren. Dit dient verder onderzocht te worden. Vermits we expliciet aangeven hoe het raamwerk evolueert (door middel van de evolutietransformaties) en vermits we ook expliciet aangeven hoe een instantie van een raamwerk gecreeerd wordt (door middel van de instantiatietransformaties) kunnen we denken aan het opstellen van een conflict tabel om deze transformaties met elkaar te vergelijken. Hiermee kunnen we dan eventuele conflicten opsporen. Hoe we deze conflicten dan dienen op te lossen moet verder onderzocht worden.

Het dient opgemerkt dat optimalizatie van de performantie van een raamwerk kan gezien worden als een operatie op dit raamwerk. Vermits de nodig ontwerp-informatie beschikbaar is, en vermits we toelaten om programmatransformaties te definiëren, kan deze operatie voorgesteld worden in ons formalisme. Bovendien laat dit toe om de impact van bepaalde optimalizaties op de rest van de implementatie na te gaan, kunnen we controleren of een bepaalde optimalizatie wel toepasbaar is en het gedrag behoudt (door middel van precondities voor de transformaties) en kunnen we eventuele conflicten opsporen. Concreet pakken we dit aan door voor elk ontwerp-patroon specifieke optimalizatietransformaties te definiëren, net zoals we er instantiatie- en evolutietransformaties voor definieerden. Immers, patronen kunnen beschouwd worden als componenten die een zekere interface definiëren. De interne werking van het patroon wordt dus afgeschermd en kan dus makkelijk aangepast worden. Het dient dan natuurlijk nog proefondervindelijk bewezen te worden dat deze transformaties de performantie van het raamwerk effectief gunstig beïnvloeden.

Een ander punt dat we verder wensen te onderzoeken is hoe we ontwikkelaars kunnen afschermen van de concrete specificatie van patronen. Dit kan eerst en vooral gebeuren door een gestructureerde vorm van documentatie te voorzien (zoals bijvoorbeeld JavaDoc) om de rol van een zekere software entiteit in een bepaalde patrooninstantie aan te geven. Een andere mogelijkheid bestaat erin om patrooninstanties automatisch te detecteren in de broncode. Er werd reeds aangetoond dat declaratieve metaprogrammatie

hiertoe in staat is [Wuy01]. Een derde mogelijkheid is om automatisch code te laten genereren voor de patrooninstanties waarbij de informatie die hiervoor nodig is gevraagd wordt aan de ontwikkelaar en nadien toegevoegd wordt aan de logische databank.

Verder kan op basis van het voorgestelde formalisme een concrete omgeving gebouwd worden die toelaat om software ontwikkeling op basis van raamwerktechnologie te ondersteunen. Deze omgeving moet geïntegreerd zijn met bestaande omgevingen en moet toelaten om de implementatie van een raamwerk op het juiste niveau te inspecteren (bijvoorbeeld het inspecteren van een bepaalde patroon, het opvragen van alle patronen waarin een bepaalde participant een rol speelt, etc). Verder dient deze omgeving ondersteuning te bieden voor de evolutie van een raamwerk, door de gebruiker de gepaste operaties te presenteren, er een uit te laten pikken en deze automatisch uit te voeren. Hierbij dienen dan eventuele conflicten gedetecteerd en gerapporteerd te worden. De gebruiker moet dan de mogelijkheid krijgen om het conflict op te lossen, en eventueel kan de omgeving hem hierbij helpen door mogelijke oplossingen te suggereren.

7 Conclusie

De oorspronkelijke doelstelling van het doctoraatsonderzoek leidde tot de ontwikkeling van een formalisme dat na een aantal experimenten toepasbaar bleek in een meer algemene context. Bijgevolg werd besloten om het onderwerp van de doctoraatsthesis niet te beperken tot enkel de optimalisatie van object-georiënteerde raamwerken, maar het uit te breiden tot het ondersteunen van software ontwikkeling door middel van raamwerktechnologie in het algemeen. De oorspronkelijke doelstelling werd dus een onderdeel van de eigenlijke doelstelling van het doctoraat.

De experimenten die werden uitgevoerd en die beschreven staan in dit document tonen duidelijk aan dat het formalisme zich uitstekend leent tot het oplossen van de problemen die zich stellen bij het ontwikkelen van software door middel van raamwerktechnologie. In de doctoraatsthesis zal dit formalisme dan ook in detail besproken en uitgewerkt worden en zal verslag uitgebracht worden van de resultaten van meer uitgebreide experimenten op software ontwikkeld in een industriële context.

Referenties

- [AS84] Harold Abelson and Jay Sussman. *Structure And Interpretation Of Computer Programs*. MIT Press, 1984.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

- [Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.
- [CCH95] James Cordy, Ian H. Carmichael, and Russell Halliday. The txl programming language, version 8. Technical report, Queen's University, 1995.
- [CDG96] Craig Chambers, Jeffrey Dean, and David Grove. Whole program optimization of object-oriented languages. Technical report, Department of Computer Science and Engineering, University of Washington, 1996.
- [Cha92] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1992.
- [Cop92] James O. Coplien. *Advanced C++ programming styles and idioms*. Addison-Wesley Publishing Company, 1992.
- [DDG⁺96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex, an optimizing compiler for object-oriented languages. In *Proceedings of the OOPSLA 96 Conference*, pages 83–100. ACM Press, 1996.
- [DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the ECOOP 95 Conference*. Springer-Verlag, 1995.
- [FS97] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 1997.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.
- [JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science, 1993.
- [JO92] Ralph E. Johnson and William Opdyke. Refactoring and aggregation. Technical report, University of Illinois, 1992.
- [Opd92] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.
- [Rob99] Don Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

- [SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of the OOPSLA'96 Conference*. Springer-Verlag, 1996.
- [Tou99] Tom Tourwé. Architecturale transformaties ter verbetering van de performantie van object-georiënteerde systemen. IWT specialisatiebeurs projectvoorstel tweede termijn, 1999.
- [VCMC97] Eugen N. Volanschi, Charles Consel, Gilles Muller, and Crispin Cowan. Declarative specialization of object-oriented programs. Technical report, INRIA Rennes, 1997.
- [Woo98] Bobby Woolf. Null object. In *Pattern Languages of Program Design*, pages 5–18. Addison-Wesley, 1998.
- [Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.
- [ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. Addison Wesley, 1990.

8 Overzicht van de nevenactiviteiten

8.1 Academiejaar 1997-1998

- Begeleiding van de oefeningen voor het vak Algoritmen en Datastructuren II (Docent: Theo D'Hondt).
- Verzorgen van het avondonderwijs voor het vak Algoritmen en Datastructuren II (Docent: Theo D'Hondt).
- Begeleider bij twee programmeerprojecten voor studenten uit de eerste licentie.
- Bijwonen van de ECOOP'98 Conferentie te Brussel.

8.2 Academiejaar 1998-1999

- Begeleiding van de oefeningen voor het vak Algoritmen en Datastructuren II (Docent: Theo D'Hondt).
- Verzorgen van het avondonderwijs voor het vak Algoritmen en Datastructuren II (Docent: Theo D'Hondt).
- Begeleider van een licentiaatsthesis van Dirk Germonprez (Promotor: Theo D'Hondt).
- Bijwonen van de ECOOP'99 Conferentie te Lissabon.
- Bijwonen van de ETAPS'99 Conferentie te Amsterdam.

8.3 Academiejaar 1999-2000

- Begeleiding van de oefeningen voor het vak Algoritmen en Datastructuren II (Docent: Theo D'Hondt).
- Begeleider bij een programmeerproject voor studenten uit de eerste licentie.
- Begeleider bij twee licentiaatsthesissen van Kokobe Heber en Imre Lebr (Promotor: Theo D'Hondt).
- Bijwonen van de SACT2000 Conferentie te Twente.
- Bijwonen van de ECOOP2000 Conferentie te Cannes.