Automatische programmageneratie uit rekenmodelspecificaties

Afstudeerverslag
door
A.P. van Waas

17 oktober 2006

Afstudeercommissie:

Prof. Dr. A. van Deursen
Dr. P.G. Kluit
Ir. B.R. Sodoyer
Ir. G.F. Zegwaard

QQQ Delft B.V.
Delft

Technische Universiteit Delft
Faculteit Elektrotechniek, Wiskunde en Informatica
Afdeling Softwaretechnologie, basiseenheid Software Engineering

Inhoudsopgave

Voorwoord

Het verslag dat voor u ligt is onderdeel van een afstudeeropdracht in het kader van de studie Technische Informatica aan de faculteit Elektrotechniek, Wiskunde en Informatica van de TU Delft.
Opdrachtgever is QQQ Delft B.V.; de afstudeeropdracht is aldaar uitgevoerd in de periode mei 2005 tot en met juli 2006.

Bij deze wil ik de volgende mensen bedanken voor hun bijdrage aan de totstandkoming van dit geheel: Peter Kluit, Gideon Zegwaard, Dik Leering, Jelle de Haan, Femke van Waas, Rogier Geerts en Walter Rutten.

Delft, september 2006

Andries van Waas

Samenvatting

Rekenmodelspecificaties bestaan uit vele rekenregels die onderling vaak weinig in structuur verschillen. Het handmatig vertalen van dergelijke specificaties naar broncode is tijdrovend en foutgevoelig. Het doel van dit afstudeerwerk is om dit vertaalproces te automatiseren om daarmee sneller, efficiënter en met minder fouten tot een implementatie van een rekenmodel te komen.

Hiertoe is een prototype ontwikkeld van het Murray framework, waarmee rekenmodelspecificaties geautomatiseerd omgezet kunnen worden naar broncode. Het framework bestaat uit een standaard specificatietaal, een codegenerator en runtime-componenten waarmee zowel metagegevens als data van een model eenvoudig en efficiënt te benaderen zijn.

De binnen het framework gebruikte specificatietaal bestaat uit dataflowdiagrammen en wiskundige formules waarmee datastromen en rekenregels van rekenmodellen eenduidig zijn vast te leggen. Deze taal, de Murray Specification Language (MSL), is een striktere vorm van reeds bestaande specificatierichtlijnen. Deze richtlijnen zijn aangescherpt en uitgebreid tot een specificatietaal waarvan syntax en semantiek eenduidig zijn vastgelegd. Als onderdeel van MSL is een notatietaal ontwikkeld om wiskundige expressies in ASCII-vorm op te kunnen schrijven voor efficiënte en correcte automatische verwerking: de Murray Expression Language (MEL). De Murray codegenerator vertaalt MSL-specificaties naar uitvoerbare broncode, namelijk .NET broncode of T-SQL statements.

Er is uitvoerig getest met een referentiemodelspecificatie. Hiervan is een handgecodeerde versie beschikbaar. De voor de specificatie gegenereerde broncode is op een aantal aspecten vergeleken met de handgecodeerde versie. Beide implementaties produceren identieke resultaten. Daarnaast is de snelheid van rekenen en van verwerken van data voor wat betreft de gegenereerde .NET code vergelijkbaar. De T-SQL variant is duidelijk langzamer. De codegenerator produceert vergeleken met handmatig coderen broncode met naar verwachting minder fouten in minder tijd. Dit laatste is goed te beredeneren maar lastig om echt aan te tonen.

Lijst van Figuren

1. Inleiding

1.1 Aanleiding voor Murray

QQQ Delft is een gespecialiseerd IT-bedrijf dat zich richt op onderzoeksinstituten en onderzoeksafdelingen van grote organisaties. QQQ ondersteunt haar klanten door middel van advisering over en realisatie van het IT-deel van projecten. Een belangrijk deel van de werkzaamheden bestaat uit het ontwikkelen en onderhouden van rekenmodellen. De rekenmodellen waar QQQ Delft aan werkt hebben betrekking op verschillende vakgebieden, van verkeer en vervoersmodellen tot modellen voor het taxeren van de bestaande woningvoorraad. De klant of een partner levert de vakinhoudelijke kennis en de modelformulering en QQQ Delft realiseert op basis hiervan een rekenmodel.

De ontwikkeling van rekenmodellen heeft een sterk iteratief karakter. Het komt in de praktijk zeer regelmatig voor dat modelspecificaties gedurende het proces van de modelimplementatie worden bijgesteld. Er is sprake van een 'kip – ei' probleem bij het opstellen van volledig correcte specificaties. Een modeldeskundige kan de juistheid van zijn modelspecificatie namelijk pas beoordelen na uitvoerige analyse van de modeluitkomsten. Hiervoor moet er een modelimplementatie beschikbaar zijn. Na analyse van de uitkomsten is de kans groot dat de modelspecificatie op onderdelen wordt aangepast en dat leidt tot een nieuwe versie van de modelimplementatie. Er zijn met andere woorden vaak meerdere ontwikkelcycli nodig om tot een definitieve vorm van een model te komen. Om doorlooptijd en kosten acceptabel te houden moet de cyclus van modelspecificatie en implementatie snel en efficiënt uitgevoerd kunnen worden.

De omzetting van modelspecificaties in broncode gebeurt handmatig en omvat veel rekenregels die onderling vaak weinig in structuur verschillen. Dit maakt het bovenstaande proces van meerdere cycli waarin een rekenmodel tot stand komt tijdrovend en foutgevoelig. Om het proces van het ontwikkelen van rekenmodellen te verbeteren is QQQ Delft in 1999 begonnen met de ontwikkeling van het LionFish framework. Een belangrijk onderdeel daarvan zijn de specificatierichtlijnen, die geleid hebben tot volledigere en eenduidigere modelspecificaties. Naast de specificatierichtlijnen, die fungeren als efficiënt communicatiemiddel tussen informatici en modeldeskundigen, biedt het framework softwarecomponenten waarmee (op modellen toegespitste) datatoegang en gebruikersinterfaces snel en eenvoudig te realiseren zijn. Ook zijn er tools beschikbaar waarmee zowel gegevens over modellen als hun in- en uitvoerdata bekeken en bewerkt kunnen worden.

De ontbrekende schakel in het verbeteren van het ontwikkelproces van rekenmodellen is de automatische vertaling van modelspecificaties naar broncode. Het Murray framework is als afstudeerwerk voor dit doeleinde ontwikkeld. Met behulp van het Murray framework worden modelspecificaties automatisch omgezet in broncode, wat in vergelijking met het handmatig omzetten sneller, efficiënter en minder foutgevoelig is.

1.2 Leeswijzer

De opbouw van dit verslag is als volgt. Hoofdstuk twee geeft een beschrijving van de uitgevoerde opdracht, de gestelde doelen, de achtergrond, de aanpak en een beschrijving van de uitgevoerde werkzaamheden.

De software requirements voor het Murray framework staan beschreven in hoofdstuk drie.

Hoofdstuk vier geeft een uitvoerige beschrijving van de Murray Specification Language. Het Murray framework werkt met rekenmodelspecificaties die in deze taal zijn uitgedrukt. Daarnaast wordt in dit hoofdstuk een algoritme gegeven waarmee een juiste volgorde van uitvoeren kan worden bepaald voor van elkaar afhankelijke rekenprocessen. Het betreffende algoritme kan omgaan met iteratieve berekeningen.

In hoofdstuk vijf wordt vervolgens een architectuurbeschrijving gegeven van het Murray framework. Ook wordt er in dit hoofdstuk dieper ingegaan op de MetaDB en DataDB onderdelen van het framework.

Onderwerp van hoofdstuk zes is het codegeneratieonderdeel van het Murray framework. De opzet en implementatie van de Murray codegenerator en diens .NET en T-SQL backends worden hier behandeld.

Hoofdstuk zeven ten slotte geeft een opsomming van de resultaten van het gedane werk en de conclusies die hieruit zijn op te maken. Voorts worden ook enkele aanbevelingen gegeven voor het uitbreiden en verbeteren van het Murray framework.

2. Projectbeschrijving

2.1 Inleiding

Dit hoofdstuk beschrijft het afstudeerwerk. Allereerst wordt in paragraaf 2.2 de probleemstelling uitgewerkt en wordt in paragraaf 2.3 de context beschreven waarbinnen alle werkzaamheden moeten worden uitgevoerd. Vervolgens wordt in paragraaf 2.4 beschreven welke werkwijze is gevolgd en tot welke planning dat heeft geleid. Daarna beschrijft paragraaf 2.5 hoe de werkzaamheden daadwerkelijk zijn uitgevoerd.

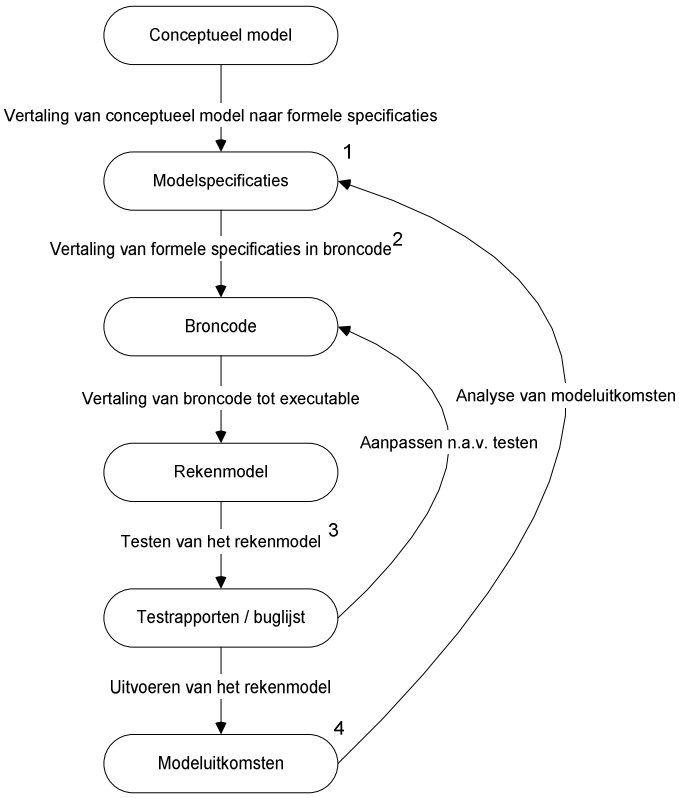
2.2 Probleemstelling

2.2.1 Formulering probleemstelling

In overleg met de opdrachtgever is de volgende probleemstelling geformuleerd:

Is het mogelijk om met behulp van automatisering het huidige proces van het ontwikkelen van rekenmodellen te versnellen en de foutgevoeligheid van de ontwikkeling van rekenmodellen te verminderen?

De ontwikkeling van een rekenmodel, van conceptueel model tot bruikbare modeluitkomsten, kent een aantal fasen. De opdrachtgever heeft een framework ontwikkeld dat bij een aantal fasen van de ontwikkeling van een rekenmodel ondersteuning biedt. Dit framework draagt de naam LionFish en wordt verder toegelicht in paragraaf 2.3. In figuur 2-1 is grafisch weergegeven welke fasen onderscheiden kunnen worden bij het ontwikkelen van een rekenmodel. In de figuur is met behulp van cijfers aangegeven waar het bestaande LionFish framework ondersteuning biedt.



Figuur 2-1 Interatieve ontwikkeling van een rekenmodel

Toelichting op de figuur:

Sommige fasen kunnen ertoe leiden dat er wordt teruggekeerd naar eerdere fasen. Dit kan het geval zijn bij het testen van een rekenmodel. Hierbij kan naar voren komen dat er functionaliteit ontbreekt, of dat er simpelweg 'fouten' worden gemaakt. Dit leidt eventueel tot een nieuwe versie van de broncode en dit proces herhaalt zich totdat er een stabiele versie is die alle functionaliteit bevat. Vervolgens kunnen de modeluitkomsten aanleiding geven tot het herzien van de modelspecificaties. Pas als alle fasen met succes doorlopen zijn kan gesproken worden van een succesvolle ontwikkeling van een rekenmodel.

Ad 1: modelspecificaties
LionFish bevat richtlijnen voor het opstellen van modelspecificaties. In hoofdstuk 4 van dit verslag wordt hier uitgebreid op ingegaan.

Ad 2: vertaling in broncode
LionFish bevat een Data Access Layer in de vorm van een klassenbibliotheek waarmee toegang tot modelgegevens sterk wordt vereenvoudigd. Daarnaast bevat LionFish functionaliteit waarmee rekenmodellen op een uniforme manier kunnen worden aangesproken. Het implementeren van rekenmodellen volgens het LionFish stramien verkort de benodigde ontwikkeltijd aanzienlijk.

Ad 3: testen van het rekenmodel
LionFish bevat een plugin voor Microsoft Excel waarmee toegang verkregen kan worden tot modelgegevens. Op deze manier kunnen individuele modelberekeningen in Excel worden nagebouwd en uitgevoerd. Door het vergelijken van de resultaten uit Excel met die van het geïmplementeerde rekenmodel kan er een uitspraak worden gedaan over de correctheid van de rekenmodelimplementatie.

Ad 4: modeluitkomsten
LionFish bevat een standaard interface, de LionFish Explorer, waarmee het eenvoudig is om uitkomsten in tabel/grafiek of kaartvorm te presenteren en te vergelijken.

In overleg met de opdrachtgever is besloten om bij de beantwoording van de in deze paragraaf gepresenteerde probleemstelling de aandacht te richten op één fase van de ontwikkeling van rekenmodellen, namelijk die van het vertalen van modelspecificatie naar broncode.

Zoals eerder vermeld biedt LionFish de modelbouwer enkele hulpmiddelen aan waarmee het implementeren van rekenmodellen kan worden versneld. De modelbouwer hoeft zich niet meer bezig te houden met het inlezen, wegschrijven en presenteren van modelgegevens. Wel moet hij de rekenregels van een rekenmodel handmatig vertalen naar broncode. Een rekenmodel bestaat uit vele rekenregels die onderling vaak weinig in structuur verschillen. Deze rekenregels moeten in een juiste volgorde worden uitgevoerd. De modelbouwer kan zowel de volgorde als de inhoud van deze rekenregels verkeerd lezen of interpreteren en daardoor fouten in de implementatie introduceren. Naast het feit dat er tijd verloren gaat aan het uitvoerig testen en eventueel repareren van de broncode kost ook het vertalen van de rekenregels naar broncode veel tijd.

Door het automatisch omzetten van rekenmodelspecificaties naar broncode verwacht de opdrachtgever het rekenmodelimplementatieproces sneller, efficiënter en met minder fouten te kunnen uitvoeren dan voorheen.

De redenen om te kiezen voor het automatisch omzetten (codegenereren) van rekenmodelspecificatie naar broncode liggen voor de hand. Hieronder worden er enkele opgesomd.

Kwaliteit

Gegenereerde code is consistent van kwaliteit. De codegenerator produceert altijd dezelfde code bij dezelfde invoer. Wanneer er zich toch fouten in de gegenereerde code bevinden, hoeven deze slechts éénmalig en op één plek te worden gerepareerd. Niet alleen fouten maar ook verbeteringen zijn snel door te voeren door simpelweg de codegenerator aan te passen en daarmee opnieuw code te genereren. De kwaliteit neemt zo dus alleen maar toe.

Door het automatisch genereren van code blijft er meer tijd over voor andere zaken, zoals specificeren en testen van de software. Dit vergroot de kwaliteit van het eindproduct.

Flexibiliteit

Het synchronisatieprobleem, waarbij specificatie en implementatie out-of-sync raken speelt bij codegeneratie veel minder. Het opnieuw genereren van code uit specificatie kan bij wijze van spreken met één druk op de knop.

Automatische codegeneratie uit specificatie maakt het mogelijk om de specificatie veel vaker en rigoureuzer aan te kunnen passen zonder grote gevolgen voor de doorlooptijd. Het leent zich dus goed voor het iteratief ontwikkelen van rekenmodellen.

Wanneer een nieuwe programmeertaal in gebruik wordt genomen hoeven niet alle bestaande implementaties handmatig te worden vertaald naar de nieuwe taal maar hoeft alleen support te worden toegevoegd voor de nieuwe taal aan de codegenerator. Idem voor bijvoorbeeld het ondersteunen van een nieuwe databackend.

Snelheid

De computer genereert uiteraard sneller code dan menselijk mogelijk is. Er moet weliswaar tijd worden geïnvesteerd in het bouwen en onderhouden van een codegenerator, maar deze tijd is snel terug te verdienen wanneer de betreffende codegenerator in gebruik wordt genomen.

2.2.2 Doelstelling

De voornaamste doelstelling van dit afstudeerwerk is het aantonen dat het mogelijk is om automatisch broncode te genereren uit LionFish rekenmodelspecificaties. Deze doelstelling is uitgewerkt in de vorm van een prototype. Dit prototype maakt het mogelijk om de bovengenoemde probleemstelling te kunnen beantwoorden. Samen met de opdrachtgever zijn een aantal requirements geformuleerd waaraan het prototype dient te voldoen. Deze requirements zijn uitgewerkt te vinden in hoofdstuk 3.

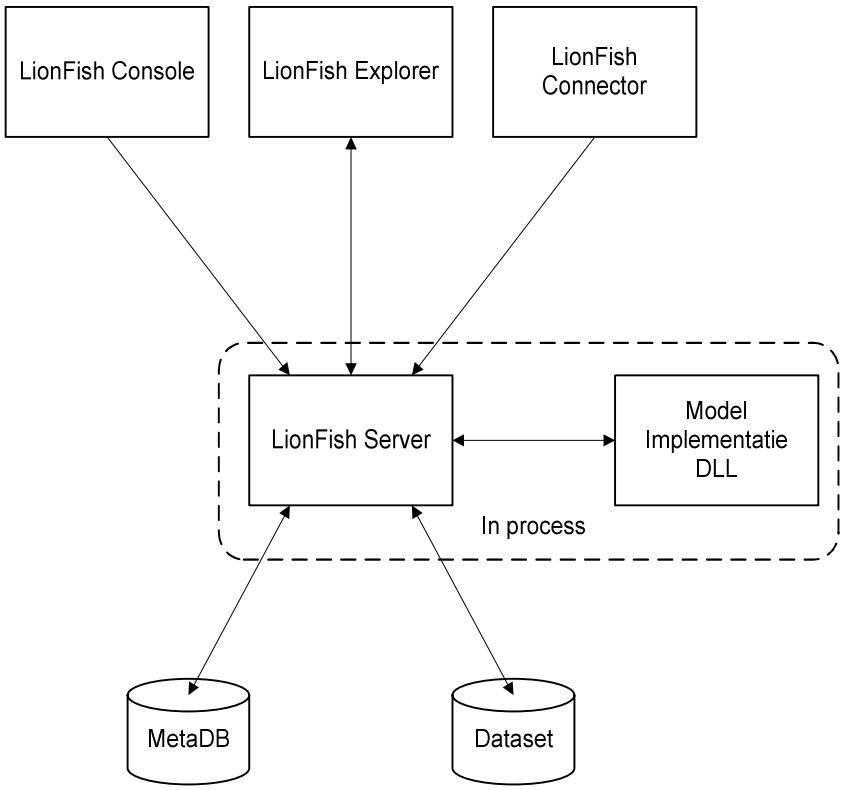
Er is nadrukkelijk gekozen voor het ontwikkelen van een prototype en niet voor het uitbreiden van LionFish met codegeneratiefunctionaliteit om de volgende redenen. Allereerst is LionFish gebaseerd op verouderde technologie (COM). De opdrachtgever heeft te kennen gegeven LionFish opnieuw te willen ontwikkelen in Microsoft.NET. Ten tweede is de beschikbare tijd voor dit project niet voldoende om een volledig functioneel product af te leveren. Wel is het mogelijk om de belangrijkste functionaliteit (core requirements) in de vorm van een verticaal prototype te implementeren. Hiermee wordt zowel de functionaliteit die benodigd is om zowel succesvol code te kunnen genereren, als de functionaliteit om deze code te kunnen uitvoeren (in de vorm van een runtime library), bedoeld. Het is de bedoeling om het ontwikkelde prototype in een later stadium uit te breiden tot een opvolger van het LionFish framework.

2.3 LionFish

2.3.1 Inleiding

In deze paragraaf wordt ingegaan op de technische opzet en de mogelijkheden van het LionFish framework, waarmee de context waarbinnen deze afstudeeropdracht valt geschetst wordt.

Om rekenmodellen snel en efficiënt te kunnen implementeren en doorrekenen heeft QQQ Delft het LionFish framework ontwikkeld. Naast een standaard specificatiemethode die gebruikt kan worden als efficiënt communicatiemiddel tussen informatici en modeldeskundigen biedt het framework softwarecomponenten aan waarmee (op modellen toegespitste) datatoegang en gebruikersinterfaces snel en eenvoudig te realiseren zijn. Ook zijn er tools beschikbaar waarmee zowel gegevens over modellen als hun in- en uitvoerdata bekeken en bewerkt kunnen worden. figuur 2-2 toont globaal de architectuur van het LionFish framework.



Figuur 2-2 Het LionFish framework

2.3.2 Specificatiemethode

De LionFish specificatiemethode [1] maakt gebruik van dataflowdiagrammen en wiskundige formules om datastromen en rekenregels binnen een model eenduidig te kunnen specificeren. De specificatiemethode is gebaseerd op opslag van data in multidimensionale arrays (variabelen). De bewerkingen op deze data zijn gespecificeerd in wiskundige formules.

2.3.3 De LionFish Server

Centraal in het LionFish framework staat de LionFish Server. Een gebruikersinterface (bijvoorbeeld de *LionFish Explorer*) maakt contact met de server om invoer te bewerken, een rekenmodel uit te voeren of uitvoer te bekijken. De *LionFish Server* kan een rekenmodel laten doorrekenen, voorziet het rekenmodel van data en verwerkt de uitvoer.

Het LionFish framework is geheel gebaseerd op het Component Object Model (COM) van Microsoft. Een COM-component biedt zijn diensten aan via één of meerdere interfaces. Een interface maakt op een taalonafhankelijke manier de functionaliteit van het component beschikbaar. Clients kunnen COM-objecten alleen via hun interfaces benaderen en niet direct. Een COM-object kan in hetzelfde proces draaien als de client, maar ook in een ander proces (op mogelijk een andere machine). COM-componenten, geschreven in verschillende talen, kunnen zonder problemen met elkaar communiceren. COM ondersteunt elke taal zolang de compiler ondersteuning biedt voor het kunnen vertalen van interne datastructuren naar het binaire COM-formaat voor uitwisseling via COM-interfaces.

2.3.4 Gegevensbeheer

De metagegevens van een model zijn opgeslagen in een *LionFish MetaDB*. Dit betreft een database in Microsoft Access formaat die informatie bevat over het model en zijn submodellen, variabelen, indices en enumeraties. Ook bevat de MetaDB informatie over de volgorde waarin submodellen worden aangeroepen, zodat de LionFish Server het desbetreffende rekenmodel kan aansturen.

Met behulp van de gegevens in een metadatabase kunnen geautomatiseerd (lege) LionFish datasets worden aangemaakt. Een *LionFish Dataset* is een Microsoft Access database met de in- en uitvoergegevens van een model.

Via de LionFish Server kunnen tegelijkertijd verschillende modellen (metadatabases) en verschillende datasets per model worden benaderd.

2.3.5 Gebruikersinterfaces

De LionFish Toolkit ondersteunt verschillende gebruikersinterfaces. De standaardinterface is de *LionFish Explorer*. Naast deze standaard interface is er een plug-in voor Excel en een command-line applicatie om LionFish rekenmodellen in batch-omgevingen toe te passen (*LionFish Console*).

Met behulp van de *LionFish Explorer* kan een rekenmodel worden uitgevoerd. De Explorer biedt verder de mogelijkheid om de invoer van een model te wijzigen en de uitvoer van een model (grafisch) te bekijken. Ook kunnen met behulp van de Explorer scenario's (resultaten van verschillende invoer-datasets) met elkaar worden vergeleken.

Het is ook mogelijk om zelf interfaces te ontwikkelen die met behulp van COM-technologie eenvoudig kunnen aansluiten op de LionFish Server. De bovengenoemde standaardinterfaces zijn grotendeels opgebouwd uit COM-objecten die hierdoor ook in andere interfaces kunnen worden gebruikt.

2.3.6 Implementatie rekenmodel

De in een model gebruikte rekenregels moeten handmatig worden gecodeerd en omgezet tot een COM-object. LionFish biedt geen ondersteuning voor het automatisch omzetten van rekenregels naar code. Wel biedt het LionFish framework ondersteuning bij het implementeren van rekenmodellen in de vorm van een klassenbibliotheek die efficiënt toegang verschaft tot zowel model meta-informatie als model in-, door- en uitvoerdata.

Een rekenmodel moet worden geïmplementeerd als COM-object en kan worden ontwikkeld in elke programmeeromgeving die COM ondersteunt. Het rekenmodel dient de ISubModel interface te implementeren waarmee de LionFish server het rekenmodel aanspreekt. Het rekenmodelcomponent wordt *binnen* het proces van de LionFish Server uitgevoerd (*in process*) zodat gegevens efficiënt tussen de server en het model uitgewisseld kunnen worden.

Via de Run-methode van deze interface krijgt het rekenmodel een Dataset-object binnen waarmee onder andere toegang wordt verkregen tot de in- en uitvoerdata van het model. De modelbouwer moet in deze methode de berekening van het rekenmodel starten.

Figuur 2-3 geeft een voorbeeld van een LionFish rekenregel. De Birth (*bir*) variabele geeft het aantal geboortes weer, uitgesplitst naar geslacht (g), regio (r) en jaar (y). De BirthTotal (*bto*) variabele geeft het aantal geboortes weer per jaar (y). De BirthTotal variabele wordt afgeleid uit de Birth variabele door de laatste over geslacht en regio te sommeren.

1.2.2 Calculate total of annual births

$$bto_y = \sum_g \sum_r bir_{g,r,y}$$

bto	**BirthTotal**
bir	**Birth**

Figuur 2-3 LionFish rekenregel

Figuur 2-4 geeft een implementatie van de rekenregel uit figuur 2-3. Deze code is *handmatig* opgesteld in C++. De rekenregel is geïmplementeerd als methode. Deze methode krijgt een Dataset-object binnen waaruit de voor de rekenregel benodigde variabelen en indices als objecten worden opgehaald. De variabele-objecten (van het type `CDEMatrix`) fungeren als een *type-safe* interface voor het kunnen uitlezen en wegschrijven van de modelgegevens. Onderhuids maken deze objecten weer gebruik van functionaliteit van de LionFish Server om de gegevens te lezen uit of weg te schrijven naar geheugen en/of database. De index-objecten (van het type `CDEIndex`) geven toegang tot de indexwaarden.

```
//  1.2.2 -- Calculate total of annual births
void CDemographicModel::Model_1_2_2(CComPtr<IDataset> piDataset)
{
	CDEMatrix<double> matBirth(piDataset,
		VARID_DEMOGRAPHICMODEL_BIRTH);
	CDEMatrix<double> matBirthTotal(piDataset,
		VARID_DEMOGRAPHICMODEL_BIRTHTOTAL);

	CDEIndex idxGender(piDataset, IDXID_GENDER);
	CDEIndex idxRegion(piDataset, IDXID_REGION);

	double dSum = 0;

	FOR_INDEX(idxGender)
	FOR_INDEX(idxRegion)
	{
		dSum += matBirth[idxGender][idxRegion][m_idxYear];
	}

	matBirthTotal[m_idxYear] = dSum;
}
```

Figuur 2-4 Implementatie van een LionFish rekenregel

2.4 Projectaanpak

2.4.1 Inleiding

In deze paragraaf wordt het gehanteerde werkplan beschreven. Daarbij wordt ingegaan op de te verrichten werkzaamheden en de op te leveren (tussen)producten.

Na afloop van dit project is een prototype ontwikkeld van het Murray framework waarmee rekenmodelspecificaties automatisch vertaald kunnen worden naar uitvoerbare broncode. Dit prototype zal in een later stadium worden uitgebreid tot een volwaardige vervanger van het LionFish framework.

Het prototype zal op een iteratieve manier worden ontwikkeld. Dit houdt in dat het ontwikkelproces in meerdere korte cycli (fasen) wordt opgedeeld. In elke cyclus wordt een deel van de gewenste functionaliteit uitgewerkt en geïmplementeerd. Hierdoor heeft de opdrachtgever al in een vroeg stadium de beschikking over een bruikbare versie (iteratie) van het te ontwikkelen product. Omdat de cycli kort zijn is het mogelijk per cyclus heldere doelstellingen te formuleren. Daarnaast krijgt de opdrachtgever de mogelijkheid om gedurende het traject de accenten te verleggen. Zo is er aan het einde van het traject een product opgeleverd waarin in ieder geval de voor de opdrachtgever belangrijkste functionaliteit is opgenomen.

2.4.2 Geplande fasering

Voor de aanvang van elke fase wordt in samenspraak met de opdrachtgever bepaald welke onderdelen van de applicatie in die fase worden uitgewerkt en geïmplementeerd. Wensen en eisen naar aanleiding van een vorige fase kunnen direct worden meegenomen. Deze werkwijze geeft de opdrachtgever de mogelijkheid om vroegtijdig bij te sturen zodat het uiteindelijke resultaat overeenkomt met de verwachting.

Tijdens de evaluatie van een fase krijgt de opdrachtgever de mogelijkheid om het prototype uit te proberen. Hij zal moeten beoordelen in hoeverre het prototype aan de verwachting voldoet. Deze evaluatie resulteert in een beschrijving van fouten, verbeterpunten en (door bijvoorbeeld voortschrijdend inzicht) nieuwe wensen en eisen aan het Murray framework. Aan de hand van deze gegevens wordt er bepaald hoe de volgende stap in het ontwikkelproces van het framework er uit zal zien.

Elke fase heeft een aantal vaste onderdelen, een eigen tijdsplanning en eigen deliverables. Aan het begin van een fase worden de voor die fase te bereiken doelen in detail geïdentificeerd. De volgende onderdelen worden telkens uitgevoerd in een fase: onderzoek, ontwerp, bouw, test en evaluatie.

In totaal zijn er vooraf vier fasen gedefinieerd die hieronder globaal worden beschreven.

Fase 1 - Basis
De eerste fase zal vooral bestaan uit het (in detail) uitwerken van de applicatie-architectuur, waarvan in de volgende fases gebruik gemaakt kan gaan worden. Hierbij zal er op gelet moeten worden dat de ontworpen architectuur open en eenvoudig uit te breiden is. Het is niet wenselijk dat de architectuur in de nog te doorlopen fases veel wordt aangepast: deze architectuur vormt het fundament van het Murray framework.

Ook zal er in de eerste fase al enige basisfunctionaliteit worden geïmplementeerd die zichtbaar is voor de gebruiker van het framework, zodat de opdrachtgever na deze fase direct kan evalueren en feedback kan geven.

Fase 2 – Generieke datalaag
De tweede fase heeft betrekking op de te benaderen databronnen. Eerder is aangegeven dat het wenselijk is dat het prototype in staat is om data in verschillende formaten te benaderen. In de tweede fase wordt dit onderdeel toegevoegd en zal er voor gezorgd worden dat gegenereerde code efficiënt verschillende databronnen kan aanspreken.

Fase 3 - Aansluiting specificatietools
In de derde fase wordt voor aansluiting gezorgd met specificatietools, in dit geval met Microsoft Visio dat bij QQQ Delft gebruikt wordt als specificatie-omgeving. In deze fase zal een plugin voor Microsoft Visio worden ontwikkeld waarmee modellen uit de MetaDB in Visio grafisch kunnen worden bewerkt.

Fase 4 – Uitbreiden ondersteunde expressies
De vierde fase bestaat uit het uitbreiden van de ondersteunde expressies zoals die in de eerste fase die ontwikkeld. Welke expressies ondersteund worden wordt in overleg met de opdrachtgever bepaald. Ook wordt er in deze fase een optimalisatieslag toegevoegd aan de applicatie om ervoor te zorgen dat de gegenereerde code de gespecificeerde rekenregels zo efficiënt mogelijk implementeert.

2.5 Projectrealisatie

2.5.1 Inleiding

In paragraaf 2.4.2 is de oorspronkelijk geplande fasering van Murray beschreven. Gedurende de uitvoering van de werkzaamheden is besloten om van deze fasering af te wijken. Fase 3 van de oorspronkelijke fasering is in overleg met de opdrachtgever komen te vervallen. De redenen hiervoor zijn:

- in fase 1 is meer functionaliteit geïmplementeerd dan oorspronkelijk was voorzien. Dit was noodzakelijk om de opdrachtgever in een zo vroeg mogelijk stadium een goed beeld te kunnen bieden van de mogelijkheden, grenzen en risico's van de gekozen oplossingsrichting. Dit had tot gevolg dat deze fase langer duurde dan gepland.
- Na afloop van fase 2 was er nog slechts beperkt tijd beschikbaar voor de voltooiing van het project en diende er een keuze gemaakt te worden tussen het uitvoeren van fase 3 dan wel fase 4. Aangezien fase 3 volledig los van de overige fasen kan worden uitgevoerd lag de beslissing tot het laten vallen van de derde fase min of meer voor de hand.

In totaal zijn er drie fasen uitgevoerd en drie versies van het prototype opgeleverd. Hieronder wordt kort beschreven hoe deze drie resterende fasen zijn ingevuld en welke werkzaamheden zijn uitgevoerd.

2.5.2 Fase 1 – Basis

Deze paragraaf geeft een overzicht van de belangrijkste werkzaamheden die in de eerste fase zijn verricht.

Vooronderzoek

De eerste fase is gestart met een vooronderzoek. Tijdens dit vooronderzoek zijn het LionFish framework en daaraan gerelateerde producten bekeken. Per product is gekeken naar de gebruikte modelleringstechniek, het grafisch kunnen invoeren van (rekenmodel-)specificaties en het kunnen genereren van uitvoerbare (bron)code uit (rekenmodel-)specificaties.

Er is voornamelijk gekeken naar simulatiesoftware. De rekenmodellen waar QQQ Delft over het algemeen mee te maken heeft kunnen hiermee worden gespecificeerd en gesimuleerd. De meeste pakketten ondersteunen het genereren van broncode uit specificaties. Deze broncode is echter voor QQQ Delft om de volgende redenen niet bruikbaar:

- de taal van de gegenereerde broncode (ansi-c) komt niet overeen met een gewenste taal (.NET taal);
- de gegenereerde code is slecht leesbaar en daarom niet geschikt om uit te breiden of aan te passen, en
- de gegenereerde code maakt gebruik van runtimecomponenten waar licentiegelden voor betaald moeten worden.

De conclusies van het vooronderzoek zijn: QQQ Delft moet zelf het automatisch omzetten van LionFish(-achtige) specificaties implementeren. Omdat het bestaande LionFish framework gebaseerd is op verouderde technologie is het verstandig een nieuw, op LionFish gebaseerd, framework te ontwikkelen in Microsoft .NET.

Opstellen Murray specificatiemethode

LionFish-specificaties worden aan de hand van richtlijnen opgesteld. Deze richtlijnen geven de modelbedenker een bepaalde vrijheid bij het noteren van zijn formules. LionFish-specificaties zijn met andere woorden niet geschikt voor automatische codegeneratie.

In de eerste fase zijn de LionFish-specificatierichtlijnen aangescherpt en uitgebreid tot een specificatietaal waarvan syntax en semantiek eenduidig zijn vastgelegd. Hiertoe zijn de volgende bestaande LionFish-specificaties geanalyseerd:

1. Strategisch Model Integrale Logistiek en Evaluatie (SMILE), scenarioverkenner voor goederenvervoer, 103 pagina's;
2. Demografisch prognosemodel, 27 pagina's;
3. Strategisch Keuzemodel ProRail (SKM), 31 pagina's, en
4. BVMS, rekenmodel/modelsysteem voor de binnenvaart, 42 pagina's.

Hieruit is gedestilleerd welke onderdelen van de LionFish specificatierichtlijnen geschikt zijn voor automatische codegeneratie. Vervolgens is uit deze verzameling weer een selectie gemaakt voor Murray. Hiervan is een ASCII-representatie ontwikkeld voor eenvoudige automatische verwerking. De Murray specificatiemethode wordt uitvoerig beschreven in hoofdstuk 4.

Requirements-analyse

In de eerste fase is een requirements-elicitatieproces uitgevoerd waarin de requirements voor het Murray framework zijn bepaald. Het resultaat van deze fase is een Software Requirements Specification (SRS), terug te vinden in hoofdstuk 3.

Aan de hand van dit SRS is een basisarchitectuurontwerp gemaakt.

Vastleggen architectuur en opstellen ontwerpen

De architectuur van het Murray framework is terug te vinden in paragraaf 5.2. De ontwerpen van de verschillende onderdelen van het framework zijn te vinden in de paragrafen 5.3 (MetaDB), 5.4 (DataDB) en hoofdstuk 6 (codegeneratie).

Implementatie eerste prototype

Voor het eerste prototype van het Murray framework zijn de volgende onderdelen geïmplementeerd:

- de MetaDB klassen. Het kunnen inlezen van MSL specificaties is essentiële functionaliteit;
- een eerste, eenvoudige implementatie van de DataDB klassen, waarmee modeldata uit een databron gelezen en naar een databron geschreven kunnen worden, en
- een eerste versie van de codegenerator, met de mogelijkheid C# code te genereren.

Het in fase 1 onstane framework is getest met de specificatie van het arbeidsmarkt-submodel (onderdeel van het demografisch prognosemodel, zie appendix B). De zo gegenereerde code produceert (na compilatie en uitvoeren) dezelfde resultaten als de LionFish referentie-implementatie. Hierbij moeten wel de volgende kanttekeningen worden geplaatst:

- het arbeidsmarkt-submodel kent slechts eenvoudige berekeningen, ingewikkelde constructies uit MSL zijn dus ofwel nog niet geïmplementeerd ofwel nog niet getest in fase 1;
- de DataDB-implementatie uit fase 1 is *bijzonder* traag en daarom eigenlijk niet bruikbaar.

2.5.3 Fase 2 – Generieke datalaag

Inbouwen gelaagde objectstructuur

Het objectmodel uit Fase 1 was niet gelaagd. De MetaDB klassen waren ‘vervuild’ met zaken die slechts voor codegeneratie relevant zijn. De DataDB klassen hadden een eigen implementatie voor het inlezen van de MetaDB vanwege deze vervuiling. In Fase 2 zijn de objectmodellen gesplitst. De MetaDB klassen bevatten nu nog slechts de velden van hun database equivalent, zonder extra functionaliteit.

Verbeteren data-handling

De DataDB-implementatie uit fase 1 is bijzonder traag. In fase 2 is het DataDB onderdeel opnieuw ontworpen en geïmplementeerd. Daarnaast is ook de database (DataDB-)implementatie gewijzigd (hoofdzakelijk vanwege aanvullende eisen gesteld door de SQL codegenerator).

Er zijn benchmarks gedraaid met de oude DataDB-implementatie uit fase 1 en de nieuwe DataDB-implementatie uit fase 2. Onderstaande tabel geeft de resultaten weer, uitgesplitst naar inlezen, rekenen en wegschrijven.

Model	Fase	Oude runtime (sec)	Nieuwe runtime (sec)
AOW Model	Inlezen	6	2
	Rekenen	82	< 1
	Wegschrijven	1054	41
Demografisch Model	Inlezen	< 1	< 1
	Rekenen	167	2.5
	Wegschrijven	1112	~250

In de tabel met benchmarks is te zien dat het rekenen met de nieuwe DataDB-implementatie vele malen sneller verloopt. Hierbij moet worden opgemerkt dat niet het rekenen is versneld, maar het opzoeken en wegschrijven van waarden van variabelen in het geheugen. In de nieuwe implementatie wordt er gebruik gemaakt van arrays waarin de data van een variabele volledig uitgeklapt in het geheugen opgeslagen staan. In de oude implementatie werd gebruik gemaakt van .NET DataTable objecten om waarden in op te slaan (niet te verwarren met de nieuwe DataTable klasse uit Murray).
Een .NET DataTable object representeert een databasetabel in geheugen. Voor elke combinatie van indexwaarden bevat een DataTable object een DataRow object met daarin de waarden van de variabelen. Deze implementatie is niet geschikt voor grote hoeveelheden data, vanwege de grote overhead. Dit volgt ook duidelijk uit de tabel met resultaten.

SQL codegeneratie

In Fase 2 is een eerste implementatie gemaakt van de SQL codegenerator. Deze implementatie kan slechts code genereren voor het arbeidsmarkt-submodel. Ingewikkelde variabele- en aggregatie-expressies worden nog niet ondersteund.

2.5.4 Fase 3 - Uitbreiden ondersteunde expressies

Procesafhankelijkheden en iteraties

In fase 3 is een algoritme ontwikkeld waarmee een juiste volgorde van uitvoeren kan worden bepaald voor van elkaar afhankelijke rekenprocessen. Bij het bepalen van een correcte volgorde wordt ook rekening gehouden met eventuele iteraties. Het algoritme wordt beschreven in paragraaf 4.4.

Ook is in deze fase het algoritme in het Murray framework ingebouwd.

.NET codegeneratie

In fase 3 is het C# backend vervangen door een nieuw ontworpen en geïmplementeerd .NET backend, zodat nu in principe elke door het .NET Framework ondersteunde taal ook door de

Murray codegenerator kan worden gegenereerd. Paragraaf 6.4 geeft een uitvoerige beschrijving van het .NET backend van de Murray codegenerator.

Uitbreiden expressie-ondersteuning SQL codegenerator

In fase 3 is ondersteuning aan de SQL codegenerator toegevoegd voor variabele-expressies en bepaalde aggregatie-expressies. Daarnaast is er ook support toegevoegd voor iteraties en het in de juiste volgorde uitvoeren van processen. De SQL codegenerator uit fase 3 genereert nu ook code (SQL statements) voor het demografisch submodel van het demografisch prognosemodel. De resultaten komen overeen met die van de referentie-implementatie. Zie paragraaf 6.5 voor een volledige beschrijving van de SQL codegenerator.

3. Software Requirements

3.1 Inleiding

3.1.1 Doel

Dit hoofdstuk fungeert als Software Requirements Specification (SRS) voor versie 1.0 van het Murray framework. In dit document zijn zowel de functionele als de niet-functionele eisen vastgelegd die aan het framework worden gesteld.

Beoogde gebruikers van dit document zijn leden van het Murray projectteam (projectleiders, software-ontwerpers, ontwikkelaars en testers) en potentiële gebruikers.

De inhoud van dit document is het resultaat van het requirements-identificatieproces dat ten behoeve van het Murray framework is uitgevoerd binnen QQQ Delft. Het requirements-identificatieteam bestond uit de volgende personen:

- Andries van Waas: software ontwerp en implementatie
- Gideon Zegwaard: modeldeskundige, modelbouwer, opdrachtgever
- Dik Leering: modeldeskundige, modelbouwer, modelgebruiker

Tenzij anders vermeld hebben alle in dit hoofdstuk gespecificeerde requirements een hoge prioriteit.

3.1.2 Scope van het project

Het Murray framework stelt de gebruiker in staat om specificaties van rekenmodellen automatisch naar direct uitvoerbare broncode te vertalen. Door het automatisch vertalen naar broncode wordt de hoeveelheid tijd die nodig is om een rekenmodel te implementeren drastisch verminderd. Bovendien zal de gegenereerde broncode minder fouten bevatten ten opzichte van een met de hand gecodeerde implementatie en zal de code geheel conformeren aan de gegeven specificatie.

3.2 Productbeschrijving

3.2.1 Context

Het Murray framework wordt ontwikkeld als vervanging voor het LionFish framework. Het Murray framework breidt de door het LionFish framework geboden functionaliteit uit met de mogelijkheid om specificaties automatisch naar direct uitvoerbare broncode te kunnen vertalen.

Versie 1.0 van het Murray framework biedt functionaliteit die nodig is om dit automatisch vertalen te kunnen faciliteren. Ook is er functionaliteit aanwezig in de vorm van componenten om gegevens van en over modellen efficiënt te kunnen benaderen. Deze componenten worden door de gegenereerde code gebruikt.

Versie 1.0 van het Murray framework biedt dus niet alle functionaliteit van het LionFish framework. Dit is pas voorzien in een latere versie. Het LionFish framework zal in de toekomst blijven bestaan aangezien de rekenmodelimplementaties die met behulp van dit framework zijn ontwikkeld niet compatibel zijn met het Murray framework.

3.2.2 Productkenmerken

Het Murray framework ondersteunt de gebruiker bij de implementatie van rekenmodellen. Met behulp van het Murray framework is het mogelijk specificaties van rekenmodellen automatisch naar uitvoerbare broncode te vertalen. Het framework bestaat uit een standaard specificatietaal, een codegenerator en runtime-componenten waarmee zowel metagegevens als data van een model eenvoudig en efficiënt te benaderen zijn.

Modelspecificaties dienen te worden uitgedrukt in de Murray Specification Language (MSL). Deze taal bestaat uit dataflowdiagrammen en wiskundige formules waarmee datastromen en rekenregels van rekenmodellen eenduidig zijn vast te leggen. Rekenregels worden uitgedrukt in de Murray Expression Language (MEL). MEL maakt onderdeel uit van MSL.

MSL-specificaties worden ingelezen uit de Murray MetaDB, een metadatabase waarin naast de specificatie zelf ook de metagegevens van het gespecificeerde model zijn opgeslagen. Dit betreft gegevens over de in de specificatie gebruikte processen, variabelen, indices, enumeraties en dergelijke.

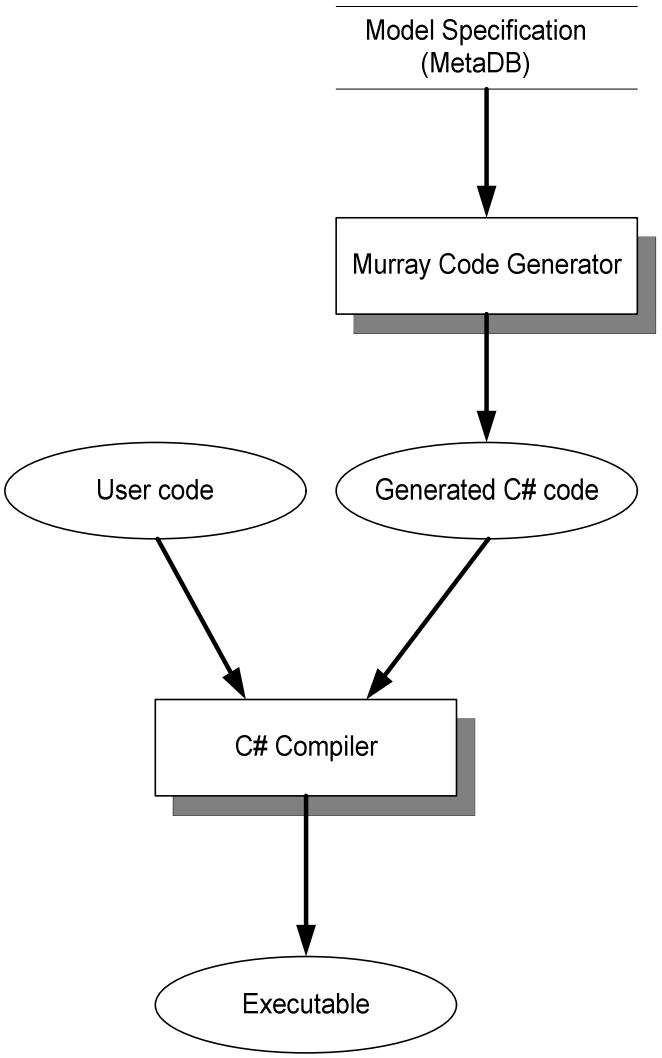
De Murray codegenerator leest een modelspecificatie uit de metadatabase, verifieert deze op correctheid, voert eventuele optimalisaties uit en genereert broncode voor een target. Het Murray framework ondersteunt de volgende twee targets:

1. het Microsoft .NET platform 1.1 met elke daarin ondersteunde taal (waaronder C# en VB.NET), en
2. Microsoft SQL Server 2000 met Transact-SQL als taal.

Alle geconverteerde rekenregels zijn (na eventuele compilatie) direct uitvoerbaar. Uitzondering hierop zijn specificaties van rekenregels die niet zijn uitgewerkt in formules. Van deze rekenregels zijn slechts beschrijvingen opgenomen in de specificatie. De codegenerator

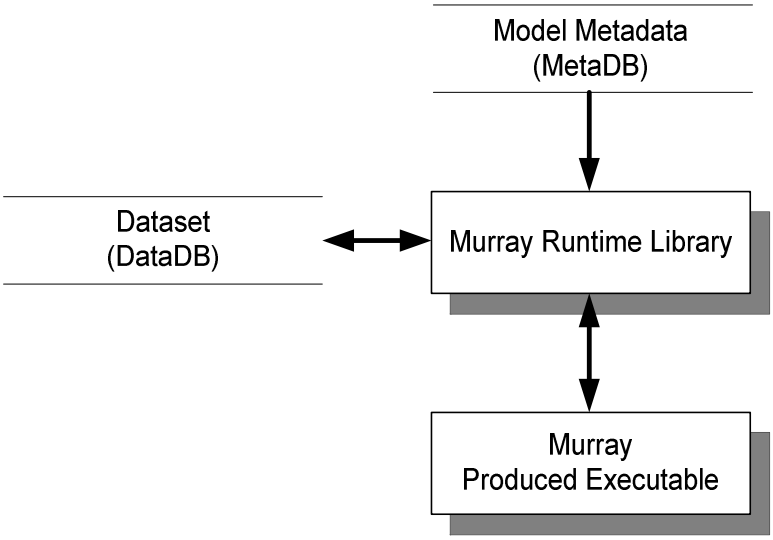
produceert voor deze rekenregels zogenaamde stubcode. Deze stubcode moet door de gebruiker worden vervangen door code die de beschreven rekenregels implementeert. De gebruiker moet in dat geval dus handmatig code toevoegen. Ook kan de gebruiker delen van de gegenereerde code vervangen door eigen code.

Figuur 3-1 geeft schematisch weer hoe een modelimplementatie in C# met behulp van het Murray framework tot stand komt. De codegenerator leest de modelspecificatie uit de MetaDB en genereert C#-broncode. De gebruiker compileert deze code samen met eventuele eigen code tot een executable.



Figuur 3-1 Modelimplementatie in C#

De gegenereerde code maakt gebruik van de in het framework aanwezige componenten voor toegang tot de model(meta-)data. Deze componenten zijn verzameld in de Murray Runtime Library (MRL). Elk target kent zijn eigen versie van de MRL. Figuur 3-2 toont het uitvoeren van de executable uit figuur 3-1. De gegenereerde code voert de rekenregels uit en heeft via de Murray Runtime Library toegang tot de (meta-) data van het model.



Figuur 3-2 Het uitvoeren van gegenereerde .NET code

3.2.3 Beoogde gebruikers

Beoogde gebruikers van het Murray framework zijn in de volgende twee categorieën onder te verdelen:

Categorie	Omschrijving
Modelbouwers	Gebruikers uit deze categorie hebben een informatica-achtergrond en houden zich voornamelijk bezig met het implementeren van rekenmodellen. Een modelbouwer zal met behulp van het Murray framework code genereren voor een modelspecificatie. Eventueel zal hij of zij de gegenereerde code uitbreiden (bijvoorbeeld door het implementeren van onderdelen van de specificatie die niet in formules zijn opgeschreven) en/of aanpassen (bijvoorbeeld het handmatig optimaliseren van delen van de gegenereerde code). De zo ontstane modelimplementatie zal worden gebruikt door gebruikers uit de tweede categorie.

Modeldeskundigen	Gebruikers uit deze categorie hebben een wiskundige of econometrische achtergrond en zijn vooral bezig met het ontwerpen en specificeren van rekenmodellen. Voor hen is het Murray framework van belang, omdat dit hen uiteindelijk in staat stelt om sneller en met een lagere foutkans te komen tot een werkend rekenmodel. Het Murray framework stelt eisen (beperkingen) aan hun modelspecificaties, en zal in enkele gevallen hun formules niet direct kunnen omzetten in broncode. Deze groep gebruikers zal vooral de modeluitkomsten analyseren om te controleren of de door het Murray framework gegenereerde code ook tot correcte berekeningen leidt.

3.2.4 Productieomgeving

Het Murray framework draait volledig op het Microsoft .NET platform versie 1.1 en zal dus functioneren op elke door dit platform ondersteunde combinatie van operating systeem en hardware configuratie.

3.2.5 Ontwerp- en implementatierestricties

R-1	Ontwikkeling van de software moet plaats vinden in Microsoft Visual Studio 2003.
R-2	Alle documentatie dient te worden opgeleverd in Microsoft Word 2003 formaat.
R-3	Ontwerpdiagrammen dienen te worden gemaakt in Microsoft Visio for Enterprise Architects.
R-4	Alle broncode moet worden geschreven in C# en compatibel zijn met het .NET Framework versie 1.1.
R-5	Alle broncode moet onder versiebeheer staan. Hiervoor dient Microsoft Source Safe 6.0 gebruikt te worden.
R-6	Zowel de broncode van het Murray framework als de gegenereerde code moet voldoen aan de eisen en richtlijnen die zijn gesteld in [7].

3.3 Functionele eisen

Hieronder worden de functionele eisen beschreven van achtereenvolgens de Murray MetaDB, de Murray codegenerator en de Murray Runtime Library.

3.3.1 Murray MetaDB

FE-1 De Murray MetaDB moet rekenmodelspecificaties uitgedrukt in de Murray Specification Language (MSL) samen met de daarbij horende metadata kunnen lezen uit een datastore en deze programmatisch beschikbaar kunnen stellen.

FE-1.1 Tijdens het inlezen van de specificatie uit de datastore moet worden gecontroleerd of de syntax van de specificatie correct is. Hierbij hoeft de syntax van formules niet te worden gecontroleerd.

FE-1.2 Bij het detecteren van een syntaxfout moet het inlezen worden afgebroken met een foutmelding.

FE-1.3 De door de Murray MetaDB gebruikte datastore moet runtime en in configuratie gekozen kunnen worden.

FE-1.4 De Murray MetaDB moet Microsoft SQL Server 2000 als datastore ondersteunen.

3.3.2 Murray codegenerator

Inlezen modelspecificatie uit MetaDB

FE-2 De Murray codegenerator moet een modelspecificatie uit de MetaDB kunnen lezen.

FE-2.1 De gebruiker moet kunnen aangeven voor welk model de specificatie dient te worden ingelezen.

Valideren modelspecificatie

FE-3 De Murray codegenerator moet een modelspecificatie kunnen controleren op syntax en semantiek.

FE-3.1 De Murray codegenerator moet kunnen rapporteren over het wel of niet valide zijn van een modelspecificatie.

Optimaliseren modelspecificatie

FE-4 De Murray codegenerator moet een modelspecificatie kunnen optimaliseren. Dit heeft een lage prioriteit.

FE-4.1 Zowel rekenregels als dataflow moeten kunnen worden geoptimaliseerd.

FE-4.2 Er moet geoptimaliseerd kunnen worden naar ofwel geheugengebruik ofwel naar snelheid van de berekeningen.

FE-4.3 De gebruiker moet het optimalisatieproces kunnen beïnvloeden door aan te geven wat hij belangrijker vindt (geheugengebruik of snelheid).

FE-4.4 De gebruiker moet individuele optimalisaties aan of uit kunnen zetten.

FE-4.5 De gebruiker moet de mogelijkheid hebben om het gehele optimalisatieproces uit te schakelen.

Codegenereren uit modelspecificatie

FE-5 De Murray codegenerator moet broncode kunnen genereren uit een modelspecificatie.

FE-5.1 Er moet voor de volgende targets broncode gegenereerd kunnen worden:
1. het Microsoft .NET platform versie 1.1 met elke daarin ondersteunde taal (waaronder C# en VB.NET);
2. Microsoft SQL Server 2000 met Transact-SQL als taal

FE-5.2 Per target moet geconfigureerd kunnen worden welke onderdelen van MSL worden ondersteund (aggregators, functies, operatoren, constanten).

FE-5.3 De gebruiker moet aan kunnen geven voor welk target er broncode dient te worden gegenereerd.

FE-5.4 De codegenerator moet de broncode naar een door de gebruiker gespecificeerd bestand kunnen wegschrijven.

FE-5.5 De gegenereerde broncode moet de modelspecificatie implementeren.

FE-5.6 De gegenereerde broncode moet (indien van toepassing: na compilatie) direct uitvoerbaar zijn.

FE-5.7 De gegenereerde broncode moet gebruik maken van de Murray Runtime Library om toegang te krijgen tot de gegevens van een model.

FE-5.8 De codegenerator moet stubcode genereren voor processen in de volgende gevallen:
1. user defined processen;
2. niet door een target ondersteunde operaties, en
3. niet valide expressies.

FE-5.9 De codegenerator moet de gebruiker melden voor welke processen er stubcode wordt gegenereerd.

FE-5.10 In de broncode moet met commentaar worden aangegeven met welk deel van de specificatie de code correspondeert. Concreet houdt dit in dat voor processen de naam en omschrijving in de broncode terug te vinden moeten zijn. Voor formuleprocessen moet de formule (in ASCII-formaat) in het commentaar meegenomen worden.

FE-5.11 Indien mogelijk moeten de namen van identifiers in de code corresponderen met de namen van de elementen in de modelspecificatie. Bij het eerste gebruik van variabelen en indices moet de omschrijving in commentaar worden meegenomen.

FE-5.12 De gegenereerde code moet debug-uitvoer kunnen genereren, die met levels te controleren moet zijn.

FE-5.13 De gegenereerde broncode moet voorzien zijn van assertions om er zeker van te zijn dat de invoer goed is.

FE-5.14 Aan broncode gemaakte aanpassingen en uitbreidingen mogen niet verloren gaan bij het opnieuw genereren van broncode.

FE-5.15 De gegenereerde code moet ondersteuning bieden voor het kunnen starten, pauzeren en stoppen van de berekeningen. Het kunnen pauzeren van berekeningen heeft een lage prioriteit.

FE-5.16 De gegenereerde code moet ondersteuning bieden voor het bepalen van de voortgang van de berekeningen. Dit heeft een lage prioriteit.

FE-5.17 De codegenerator moet debug-uitvoer kunnen produceren waarmee het codegeneratieproces door de gebruiker kan worden gevolgd.

3.3.3 Murray Runtime Library

FE-6 De Murray Runtime Library moet de gegevens van een model kunnen beheren en programmatisch beschikbaar kunnen stellen.

FE-6.1 Modelgegevens moeten kunnen worden gelezen uit en weggeschreven naar een datastore.

FE-6.2 De door de Murray Runtime Library gebruikte datastore moet runtime en in configuratie gekozen kunnen worden.

FE-6.3 De Murray Runtime Library moet Microsoft SQL Server 2000 als datastore ondersteunen.

3.4 Interface-eisen

3.4.1 Gebruikersinterfaces

Het Murray framework communiceert met de gebruiker via een Command Line Interface (CLI) en configuratiebestanden. Configuratie-instellingen kunnen ook via de commandline worden opgegeven. Deze laatste instellingen hebben prioriteit boven de instellingen in de configuratiebestanden.

3.4.2 Hardware-interfaces

Het Murray framework maakt geen gebruik van hardware-interfaces.

3.4.3 Software-interfaces

Het Murray framework maakt gebruikt van de volgende software-interfaces:

Microsoft .NET 1.1	Het Murray framework draait op het Microsoft .NET 1.1 platform en maakt gebruikt van diens Application Programming Interfaces (API's).
ADO.NET 1.1	Het Murray framework maakt gebruik van ADO.NET 1.1, een databasetoegangs API voor op .NET-gebaseerde applicaties. Via ADO.NET kunnen zowel ODBC als OLE DB databronnen worden aangesproken. Ook is er een interface naar SQL Server databases.

3.5 Niet-functionele eisen

3.5.1 Performance-eisen

PE-1 Het genereren van code voor een model specificatie mag per expressie niet langer dan 1 seconde duren.

PE-2 De gegenereerde code moet ten opzichte van de modelspecificatie dezelfde of lagere tijdcomplexiteit bezitten.

3.5.2 Kwaliteitsattributen

Betrouwbaarheid

Gelijke resultaten	Het Murray framework / de Murray codegenerator moet bij het meerdere malen invoeren van een modelspecificatie iedere keer dezelfde resultaten leveren. Het wel of niet optimaliseren van een modelspecificatie mag geen significante invloed hebben op de uiteindelijke resultaten.
Nauwkeurigheid	Het vergelijken van twee floating-point getallen moet gebeuren met een configureerbare delta.

Onderhoudsvriendelijkheid

Onderhoudsvriendelijkheid van het Murray framework wordt gegarandeerd door het volgende:

- Ontwerpdocumenten zijn up-to-date;
- Broncode is voorzien van commentaar: elke class, interface, property, field en method moet voorzien zijn van commentaar. In het commentaar moet worden verwezen naar de ontwerpdocumenten.
- Broncode moet voldoen aan de codeerrichtlijnen gespecificeerd in [7].

Herbruikbaarheid

Waar mogelijk moeten het Murray framework en de gegenereerde code gebruik maken van dezelfde componenten.

Uitbreidbaarheid

Het Murray framework moet voor wat betreft onderstaande zaken uitbreidbaar zijn:

Target-taal	Het moet mogelijk zijn om een nieuwe target-taal toe te voegen of een dialect van een bestaande taal.
Expressie	Het moet mogelijk zijn om ondersteuning voor nieuwe functies, aggregatoren, operatoren en constanten toe te voegen.
Optimalisatie	Het moet mogelijk zijn om nieuwe optimalisatieroutines toe te voegen.
Data support	Het moet mogelijk zijn om support toe te voegen aan de Murray Runtime Library voor nieuwe data-backends.

Bovenstaande moet mogelijk zijn met minimale aanpassingen van bestaande code. Er moet gebruik gemaakt worden van base-classes, interfaces en configuratiebestanden om dit te faciliteren.

4. Murray Specification Language

4.1 Inleiding

Om rekenmodellen op een uniforme manier te kunnen specificeren heeft QQQ Delft de LionFish specificatierichtlijnen ontwikkeld [1]. Deze specificatierichtlijnen beschrijven het gebruik van dataflowdiagrammen en wiskundige formules voor het vastleggen van de datastromen en rekenregels binnen een rekenmodel.

De specificatierichtlijnen zijn opgezet als efficiënt communicatiemiddel tussen informatici en modeldeskundigen. De richtlijnen zijn voor mensen uit beide groepen goed te begrijpen en zijn daarom eenvoudig toe te passen. Er wordt geen onnodige tijd besteed aan het vertalen naar en interpreteren van elkaars specificaties, omdat er met slechts één (soort) specificatie wordt gewerkt.

LionFish specificaties zijn niet direct bruikbaar voor codegeneratie. Dit komt doordat deze specificaties zijn opgesteld aan de hand van slechts de eerder beschreven richtlijnen. Binnen deze richtlijnen is bijvoorbeeld de semantiek van de in formules gebruikte wiskundige notatie niet eenduidig vastgelegd. Zolang modelontwerper en modelbouwer de notatie op dezelfde manier interpreteren hoeft dat geen probleem te zijn. De exacte semantische betekenis wordt duidelijk uit de context of na uitleg van de ontwerper van het model.

Wanneer een modelspecificatie automatisch geïnterpreteerd moet kunnen worden is het nodig dat de semantiek van de in de specificatie gebruikte expressies volledig en eenduidig is gedefinieerd. De Murray Specification Language (MSL) is een striktere vorm van de LionFish specificatierichtlijnen, waarin expressies maar op één mogelijke manier geïnterpreteerd kunnen worden. Als onderdeel van MSL is een notatietaal ontwikkeld om wiskundige expressies in ASCII-vorm op te kunnen schrijven voor efficiënte automatische verwerking: de Murray Expression Language (MEL).

De indeling van dit hoofdstuk is als volgt: paragraaf 4.2 geeft een beschrijving van de in de Murray Specification Language gedefinieerde entiteiten; paragraaf 4.3 geeft een uitvoerige beschrijving van de Murray Expression Language.

4.2 Murray Specification Language

4.2.1 Inleiding

De belangrijkste entiteiten binnen MSL zijn *processen*, *variabelen*, *dataflows* en *formules.* In de volgende paragrafen worden deze begrippen nader toegelicht. Ook worden andere hieraan gerelateerde zaken besproken.

Een Murray specificatie bestaat uit datastromen en rekenregels. Rekenregels worden gespecificeerd in de vorm van formules; deze formules worden uitgerekend binnen processen. Processen zijn hiërarchisch gegroepeerd. In processen op het laagste niveau vinden de uiteindelijke berekeningen plaats. Datastromen (dataflows) worden gerepresenteerd door variabelen; variabelen zijn afkomstig uit invoer of worden uitgerekend in formules. Elke formule rekent één variabele uit. Deze uitgerekende variabele is vervolgens weer als invoer te gebruiken in andere formules. Ook kan deze variabele uitvoer zijn van het model.

Hieronder worden de binnen MSL voorkomende entiteiten één voor één beschreven.

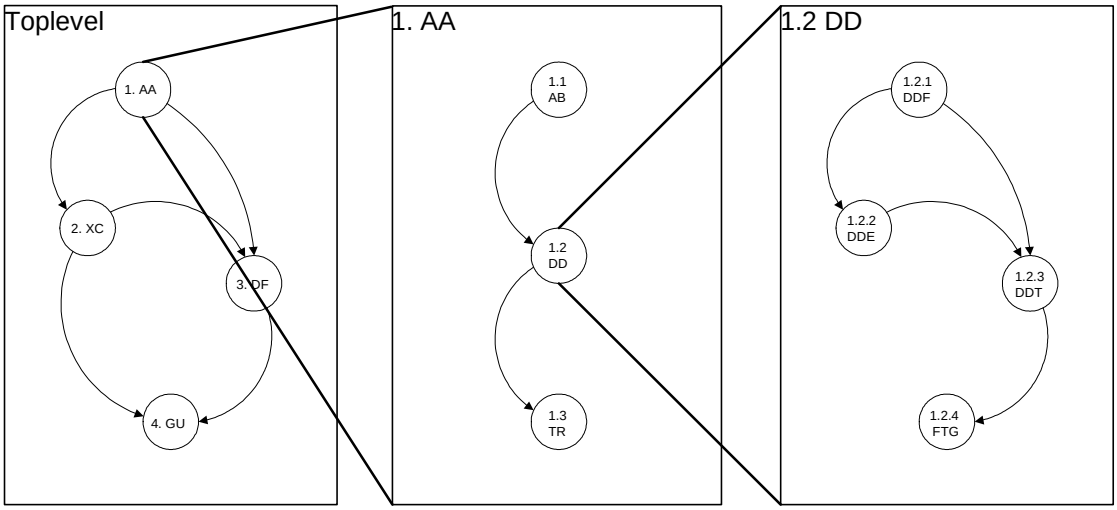
4.2.2 Processen

Een model is opgebouwd uit processen. Binnen deze processen vinden de berekeningen plaats. Processen worden opgedeeld in deelprocessen totdat elk deelproces een enkelvoudige bewerking is geworden. Deelprocessen op dit laagste niveau moeten in een enkelvoudige formule uit te drukken zijn en worden formuleprocessen genoemd.

Dit opdelen in hiërarchische processen bevordert de leesbaarheid van de modelspecificatie. Ook is het op deze manier mogelijk om ingewikkelde submodellen eenvoudig te vervangen door een vereenvoudigde versie, om zo de benodigde rekentijd tijdens het ontwikkelen en testen van een model te beperken.

Elk proces wordt geïdentificeerd door een nummer. Dit nummer bestaat uit het nummer van het proces waar het deel van uitmaakt, gevolgd door een punt en een volgnummer. Proces 1.2.3 bijvoorbeeld is een deelproces van proces 1.2, dat op zijn beurt een deelproces is van proces 1.

Grafisch worden processen weergegeven als een bol. In deze bol staat het nummer van het proces, samen met een omschrijving. Figuur 4-1 illustreert de opdeling van een model in deelprocessen.



Figuur 4-1 Opdeling in deelprocessen

4.2.3 In- en uitvoerprocessen

In- en uitvoerprocessen representeren de in- en uitvoerdata van een model. Dit type processen heeft een naam, geen nummer. Een model mag meerdere in- en uitvoerprocessen bevatten. Dit maakt het mogelijk om in- en uitvoervariabelen te groeperen. Grafisch worden in- en uitvoer processen weergegeven met een datastore-symbool. De naam van het proces staat binnen de lijnen van het symbool:

Figuur 4-2 Notatie in-/uitvoerproces

4.2.4 Variabelen

Data wordt gerepresenteerd door variabelen. Variabelen kunnen geïndexeerd zijn (multidimensionaal). Elke bij de variabele behorende index representeert een dimensie. Het bereik van een dimensie komt overeen met het bereik van de bijbehorende index. Indexbereiken zijn onderling niet afhankelijk, dat wil zeggen dat mogelijke indexwaarden van de ene index niet afhangen van de op dat moment gekozen waarden van andere indices. De data van een variabele is dus volledig multidimensionaal.

Variabelen krijgen slechts éénmaal een waarde toegekend voor een bepaalde combinatie van indexwaarden. Deze waarde mag niet meer veranderen tijdens het uitvoeren van het model. Variabelen hebben slechts één herkomst, dat wil zeggen ofwel invoer, ofwel (uitgerekend door) een proces.

Variabelen komen in drie vormen voor:

1. *Invoervariabelen* worden door de rekenprocessen slechts gelezen. De waarden van deze variabelen zijn afkomstig uit de invoer.
2. *Doorvoervariabelen* worden zowel geschreven als gelezen. De waarden van deze variabelen worden tijdens het rekenproces berekend. Eenmaal uitgerekend kunnen deze variabelen weer als invoer dienen binnen het rekenproces. Doorvoervariabelen kunnen ook als uitvoer van het model worden aangeboden; dit maakt het controleren van 'tussenstappen' eenvoudiger.
3. *Uitvoervariabelen* worden net als doorvoervariabelen uitgerekend binnen het model. De waarden van deze variabelen worden echter als uitvoer aangeboden en niet meer door het rekenproces gebruikt.

Een variabele heeft de volgende eigenschappen:

Naam De naam van de variabele; deze wordt gebruikt in de dataflow diagrammen. Een naam moet voldoen aan de volgende eisen:

- De naam van een variabele moet uniek (case-sensitive) zijn binnen een model. De richtlijn is om geen namen te gebruiken die slechts in hoofdlettergebruik van elkaar verschillen.
- De naam moet beginnen met een hoofdletter en mag verder bestaan uit letters, cijfers en underscores.
- Richtlijn voor de lengte is 4 tot 15 karakters.

Afkorting De afkorting van de variabele die wordt gebruikt in de formules. Een afkorting moet voldoen aan de volgende eisen:

- De afkorting van een variabele moet uniek (case-sensitive) zijn binnen een model.
- De afkorting moet beginnen met een kleine letter en mag verder bestaan uit kleine letters en cijfers. De afkorting mag niet eindigen op een cijfer.
- Richtlijn voor de lengte is 1 tot 5 karakters.

Type Het type van de variabele. Mogelijke types zijn:

- Integer (32 bits)
- Float (double precision, 64 bits)
- Bool

Bereik Het bereik van de variabele. De waarden van de variabele moeten binnen een bepaald bereik liggen. Van dit bereik worden de minimum en maximum waarde vastgelegd.

Defaultwaarde De defaultwaarde van de variabele. Indien de waarde van een variabele voor een bepaalde indexcombinatie niet is gedefinieerd wordt de defaultwaarde aangenomen. Een waarde is niet gedefinieerd wanneer de variabele voor de gevraagde combinatie wordt gebruikt vóórdat deze is uitgerekend (doorvoer- of uitvoervariabele) of wanneer de variabele voor de gevraagde combinatie niet in de invoer is gedefinieerd (invoervariabele). De defaultwaarde moet binnen het bereik van de variabele liggen.

Groep De groep waarin de variabele is ingedeeld. Dit is optioneel. Denk hierbij bijvoorbeeld aan het groeperen van in- en uitvoervariabelen. Groeperen kan nuttig zijn voor het efficiënt afhandelen van data (door bijvoorbeeld variabelen per groep in te lezen uit of weg te schrijven naar cache of datastore).

Indices De verzameling van bij de variabele horende indices. Deze verzameling kan leeg zijn. De variabele heeft in dat geval slechts één waarde. Een index mag in deze verzameling slechts éénmaal voorkomen.

Enumeratie De bij de variabele behorende enumeratie. Dit is optioneel. Wanneer een variabele een enumeratie heeft, gelden de volgende extra voorwaarden:

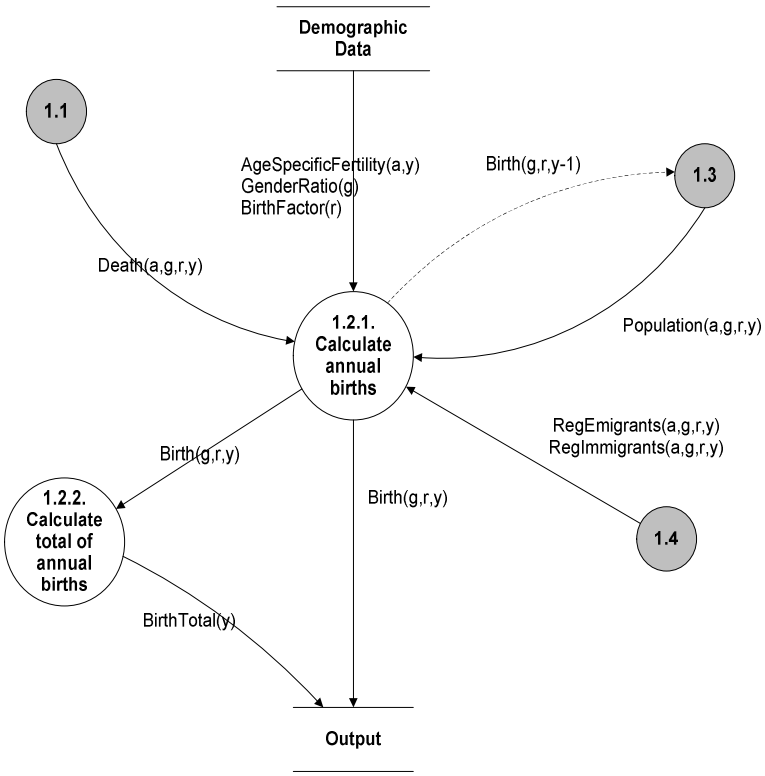
- De mogelijke waarden van de variabele komen overeen met de mogelijke waarden van de enumeratie. Het bereik van de variabele komt dus overeen met dat van de enumeratie.
- Het type van de variabele is Integer.

4.2.5 Dataflows

Alle gegevensstromen (dataflows) tussen processen onderling en tussen processen en in- en uitvoer worden expliciet weergegeven in het model. Deze dataflows worden grafisch weergegeven door een pijl met daarbij (onder elkaar) de variabelen. Variabelen worden bij dataflow weergegeven door de naam van de variabele, gevolgd door een opsomming van zijn indices (afkortingen) tussen haakjes, in alfabetische volgorde:

Population(a,g,r,y)

Figuur 4-3 geeft een voorbeeld van de verschillende soorten dataflows. Iteratieve dataflow wordt weergegeven met een onderbroken pijl. In paragraaf 4.2.10 wordt hier verder op ingegaan.



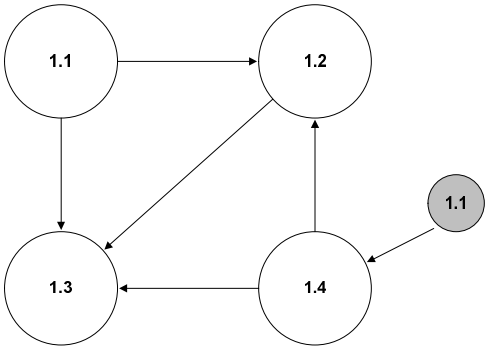
Figuur 4-3 Berekening geboortes

Een proces kan meerdere ingaande flows hebben uit verschillende processen. Bij een proces op het laagste niveau hebben alle uitgaande flows alleen betrekking op de in dit proces uitgerekende variabele. Processen op hogere niveaus kunnen verschillende uitgaande flows hebben, dat wil zeggen flows met (mogelijk) meerdere verschillende variabelen.

Bij het beschrijven van processen wordt onderscheid gemaakt tussen het definiëren van en het refereren naar processen. Processen worden gedefinieerd in de opdeling waar ze deel van uitmaken. Bij het beschrijven van processen in een bepaalde opdeling worden de processen die deel uitmaken van deze opdeling genoteerd als een grote witte bol met daarin het nummer en de omschrijving van het proces. Processen waarnaar gerefereerd wordt (via dataflow) worden genoteerd als kleine grijze bol.

In figuur 4-3 wordt de opdeling van proces 1.2 in de deelprocessen 1.2.1 en 1.2.2 beschreven. De processen 1.1, 1.3 en 1.4 zijn geen onderdeel van deze opdeling en worden daarom als kleine grijze bol genoteerd.

Er kan worden gerefereerd naar processen in dezelfde opdeling en naar processen uit een andere opdeling. Refereren naar processen in dezelfde opdeling kan handig zijn wanneer er zich in het diagram veel processen met onderling veel flows bevinden. Zo kan het kruisen van flows worden voorkomen. In de grijze bol wordt in dit geval het nummer van het proces genoteerd waarnaar wordt gerefereerd. Figuur 4-4 geeft hier een voorbeeld van.

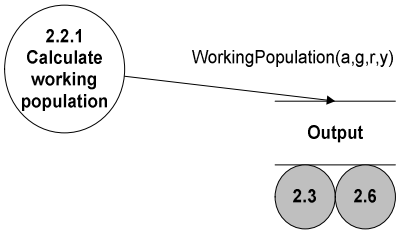


Figuur 4-4 Refereren naar processen in dezelfde opdeling

Bij het refereren naar processen in een andere opdeling wordt niet het nummer van dat proces in de grijze bol genoteerd, maar het nummer van het hoogste proces in de proceshiërarchie dat wel het proces bevat waarnaar gerefereerd wordt, maar niet het proces dat daarnaar refereert.

In figuur 4-3 wordt de variabele Death uitgerekend in deelproces 1.1.1 en vervolgens gebruikt in deelproces 1.2.1. Het hoogste nog verschillende niveau van 1.1.1 ten opzichte van 1.2.1 is 1.1 (1 is niet verschillend met 1.2.1). Daarom wordt 1.1 in de grijze bol genoteerd.

Identieke flows afkomstig uit één proces naar processen in een andere opdeling of uitvoer mogen in een diagram worden opgeschreven als één flow naar een groep van processen. Figuur 4-5 laat hier een voorbeeld van zien.



Figuur 4-5 Flow naar groep processen

Een invoerproces heeft alleen uitgaande flows, een uitvoerproces alleen ingaande flows. Er mogen geen (directe) dataflows bestaan tussen in- en uitvoerprocessen. Een variabele mag dus niet direct van invoer naar uitvoer gekopieerd worden.

4.2.6 Formules

Bij een proces op het laagste niveau hoort een formule. Formules worden in de vorm van formuleblokken opgenomen in de dataflowdiagrammen. Figuur 4-6 geeft een voorbeeld van een formuleblok. Dit formuleblok hoort bij proces 1.2.1 uit figuur 4-3. Naast de formule zelf bevat een formuleblok het volgnummer en de titel van het deelproces waar het bij hoort. Daarnaast bevat het een lijst van gebruikte variabelen in de vorm *afkorting, volledige naam.*

1.2.1. Calculate annual births

$$\mathbf{a1} = \{a1 \mid a1 \in a.(a1 \geq 15 \wedge a1 \leq 49)\}$$

$$bir_{g,r,y} = gra_{g} \cdot \sum_{a1} \left(asf_{a1,y} \cdot \left(pop_{a1,g='female',r,y} + \frac{rim_{a1,g='female',r,y} - rem_{a1,g='female',r,y} - dea_{a1,g='female',r,y}}{2} \right) \right) bfa_{r}$$

asf	AgeSpecificFertility
bir	Birth
dea	Death
rem	RegEmigrants
gra	GenderRatio
rim	RegImmigrants
pop	Population
bfa	BirthFactor

Figuur 4-6 Voorbeeld formuleblok

Een formule rekent één variabele uit door middel van een enkele assignment. Aan de linkerkant van het =-teken wordt de uit te rekenen variabele met bijbehorende indices opgeschreven, en aan de rechterkant de uit te voeren expressie. Eventuele indexdefinities die in de expressie worden gebruikt worden in de grafische notatie boven de formule genoteerd. Variabelen die als invoer dienen voor het deelproces kunnen in de formule worden gebruikt samen met (deelverzamelingen van) de beschikbare indices.

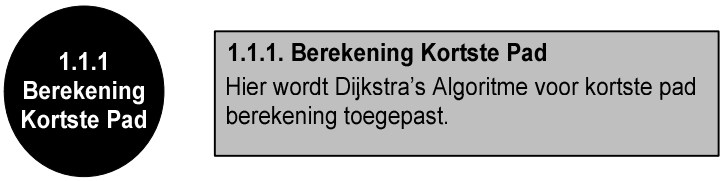
Expressies worden uitgedrukt in de Murray Expression Language (MEL). In paragraaf 4.3 is een beschrijving hiervan opgenomen.

4.2.7 User-defined processen

Wanneer het niet mogelijk is om de berekening van een variabele uit te drukken in MEL kent MSL de mogelijkheid om een zogenaamd user-defined proces te definiëren. Bij een dergelijk proces wordt de berekening van de variabele in woorden en/of (niet door MSL ondersteunde) wiskundige formules beschreven.

De uitgaande flows van een user-defined proces hebben net als bij een formuleproces slechts betrekking op de in het proces uitgerekende variabele. Ook mogen bij de berekening van de variabele slechts de variabelen gebruikt worden die via een flow het proces binnenkomen.

Grafisch wordt een dergelijk proces zwart ingekleurd weergegeven. Daarnaast wordt er in de specificatie een blok opgenomen met daarin naast het volgnummer en de titel van het proces de beschrijving van de berekening. Dit blok heeft een grijze achtergrond. Figuur 4-1 geeft een voorbeeld van deze grafische notatie.



Figuur 4-7 Notatie user-defined proces

4.2.8 Indices

Een index is een dimensie van een variabele. Meerdere variabelen kunnen als dimensie dezelfde index hebben. Er zijn twee typen indices: basisindices en afgeleide indices. Een basisindex is een index die voor het hele model is gedefinieerd. Een afgeleide index bezit een subset van de indexwaarden van een basisindex en is uniek binnen een formule (zie paragraaf 4.3.4). Een index heeft de volgende eigenschappen:

Naam De naam van de index. Een naam moet voldoen aan de volgende eisen:

- De naam moet uniek (case-sensitive) zijn binnen een model (basisindex) of formule (afgeleide index).
- De naam moet beginnen met een hoofdletter en mag verder bestaan uit letters, cijfers en underscores.
- Richtlijn voor de lengte is 4 tot 15 karakters.

Afkorting De afkorting van de index. Deze wordt bijvoorbeeld gebruikt bij het weergeven van een variabele bij een flow. Een afkorting moet voldoen aan de volgende eisen:

- De afkorting van een index moet uniek (case-sensitive) zijn binnen een model, met uitzondering van in aggregatorexpressies geïntroduceerde afgeleide indices.
- De naam moet beginnen met een kleine letter en mag verder bestaan uit kleine letters en cijfers. De afkorting mag niet eindigen op een cijfer.
- Richtlijn voor de lengte is 1 tot 5 karakters.

Bereik Het bereik van de index. Van het bereik worden de minimum en maximum waarden vastgelegd. Wanneer een index een enumeratie heeft, komt het bereik overeen met het bereik van de enumeratie.

Enumeratie De enumeratie van de index. Dit is optioneel. Wanneer een index een enumeratie heeft, komen de mogelijke waarden van de index overeen met de mogelijke waarden van de bijbehorende enumeratie.

4.2.9 Enumeraties

Een enumeratie is een verzameling labels met bijbehorende integere waarden. Enumeraties worden gebruikt om expressies leesbaarder te maken. Als er aan een variabele of index een enumeratie is gekoppeld, kunnen overal waar waarden van deze variabele of index worden gebruikt ook labels van de enumeratie worden gesubstitueerd.

De naam van een enumeratie moet uniek (case-sensitive) zijn binnen een model. Een naam moet beginnen met een hoofdletter en kan verder nog bestaan uit letters en cijfers.

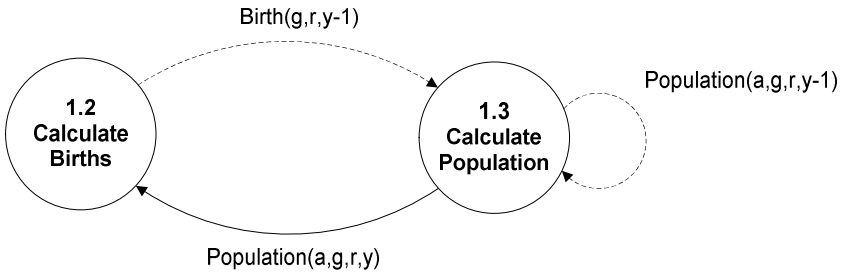
De naam van een label moet uniek (case-sensitive) zijn binnen de enumeratie. De naam moet beginnen met een letter en kan verder nog bestaan uit letters, cijfers en underscores. De bij de labels behorende waarden moeten zowel uniek als aaneensluitend zijn. Waarden mogen bij elk willekeurig getal beginnen.

4.2.10 Iteraties

Het is mogelijk iteratieve processen te modelleren. Hiervoor wordt gebruik gemaakt van één of meerdere iteratie-indices. In het model wordt een flow met een onderbroken pijl weergegeven wanneer een proces invoer krijgt van een proces uit een vorige iteratie. Een dergelijke flow heet een iteratieve flow. In tegenstelling tot gewone flows mogen iteratieve flows betrekking hebben op slechts één proces. In dat geval maakt het proces gebruik van gegevens die door *hetzelfde* proces in een vorige iteratie zijn berekend.

Bij het noteren van de flowvariabelen wordt er aan elke iteratie-index de expressie “-1” toegevoegd om aan te geven dat er over deze index wordt geïtereerd.

Figuur 4-8 geeft een voorbeeld van iteratie over één index (index y). Proces 1.3 rekent met de gegevens die in een vorige iteratieslag door proces 1.2 en zichzelf zijn berekend. Proces 1.2 maakt gebruik van de gegevens die proces 1.3 diezelfde iteratie heeft berekend.



Figuur 4-8 Iteratie voorbeeld

Tussen twee processen kunnen maximaal twee flows bestaan: een iteratieve flow met variabelen die betrekking hebben op de iteratie-index/indices en een normale flow met de variabelen die geen betrekking hebben op de iteratie-index/indices.

4.3 Murray Expression Language

4.3.1 Inleiding

In deze paragraaf wordt de Murray Expression Language (MEL) beschreven. De taal kent twee notatievormen: een ASCII-vorm en een grafische vorm. De ASCII-vorm wordt gebruikt voor het genereren van code. De grafische vorm wordt gebruikt voor het weergeven van specificaties. De ASCII-vorm is eenvoudig automatisch te verwerken, de grafische vorm is "human-readable". De complete ASCII syntax van MEL in EBNF-notatie is terug te vinden in appendix A.

4.3.2 Formule

Een in de Murray specificatiemethode gebruikte formule bestaat uit een assignment van een expressie aan een variabele. Aan de linkerkant van deze assignment wordt de variabele genoteerd, samen met al zijn indices op alfabetische volgorde. Aan de rechterkant van de assignment volgt de MEL-expressie. De variabele-expressie aan de linkerkant van de assignment mag alleen indexnamen bevatten (Zie 4.3.8 voor een uitleg over variabele-expressies). De MEL-expressie mag samengesteld zijn uit verschillende soorten MEL-expressies.

In onderstaand voorbeeld wordt het resultaat van het produkt van de variabelen *img* en *mfi* aan de variabele *rim* toegekend.

$$rim_{a,g,r,y} = img_{a,g,y} \cdot mfi_{a,g,r,y} \qquad \text{(Grafisch)}$$

```
rim[a,g,r,y] := img[a,g,y]*mfi[a,g,r,y]                    (ASCII)
```

Een variabele is multidimensionaal en heeft voor elke combinatie van indexwaarden een waarde. De aan de variabele toegekende expressie wordt voor elke combinatie van indexwaarden opnieuw uitgerekend. Een in deze expressie gebruikte index krijgt bij het uitrekenen de waarde van die index op dat moment. Onderstaande pseudo-code geeft aan hoe de formule in bovenstaand voorbeeld wordt uitgerekend. Met behulp van (geneste) *for*-lussen worden alle mogelijke combinaties van indexwaarden gegenereerd. Voor elke combinatie wordt de expressie uitgerekend en aan de variabele toegekend.

```
for a := min(a) to max(a)
   for g := min(g) to max(g)
      for r := min(r) to max(r)
         for y := min(y) to max(y)
            rim[a,g,r,y] := img[a,g,y]*mfi[a,g,r,y]
         next y
      next r
   next g
next a
```

De *for*-lussen worden geordend volgens de alfabetische volgorde van de bijbehorende indices.

Bij het iteratief uitrekenen van een formule worden de *for*-lussen die betrekking hebben op de iteratie-indices *buiten* de formule uitgevoerd.

4.3.3 Types

Aan iedere expressie is een type gekoppeld. Binnen MEL worden de volgende types ondersteund:

Type	Grootte (bits)	Bereik
Integer	32	-2^{32} t/m $2^{32}-1$
Float	64	$-1.79\cdot10^{308}$ t/m $1.79\cdot10^{308}$
Bool	1	true, false

Binnen expressies vindt waar nodig impliciete typeconversie plaats. Dit houdt in dat op de plek waar een expressie van een bepaald type wordt verwacht, er ook een expressie met een type dat geheel in het verwachte type past gebruikt mag worden. Zo mag een integerexpressie gebruikt worden waar een float expressie wordt verwacht. Bij de beschrijving van de verschillende expressiesoorten zal er waar nodig op impliciete typeconversie worden ingegaan. Expliciete typeconversie is geen onderdeel van MEL maar kan wel worden bewerkstelligd met behulp van functies, zoals de functie *convert*. Zie 4.3.11 voor een beschrijving van functies.

De modelontwerper dient erop te letten dat er geen type overflow plaatsvindt binnen het model. Een berekening waarbinnen type overflow plaatsvindt is niet gedefinieerd. Dit houdt in dat voor het model gegenereerde code in een dergelijk geval stopt met een foutmelding.

4.3.4 Indices

Een index-expressie bestaat uit de naam van een index en heeft als betekenis de waarde van die index op dat moment. Het type van een index-expressie is *Integer.* Indices mogen alleen in een expressie worden gebruikt wanneer ze op dat moment *beschikbaar* zijn. De indices van de variabele die wordt uitgerekend zijn altijd beschikbaar. Binnen aggregatie-expressies is de index waarover geaggregeerd wordt ook beschikbaar. Aggregatie-expressies worden beschreven in 4.3.9.

4.3.5 Indexdeclaraties

MEL ondersteunt het declareren van afgeleide indices. Afgeleide indices worden gebruikt bij aggregatie-expressies. Zie 4.3.9 voor een beschrijving van aggregatie-expressies.

Een afgeleide index is gebaseerd op een bronindex en bezit een subset van de indexwaarden van deze bronindex. De afgeleide index kan een synoniemindex zijn of een deelindex. Een synoniemindex verschilt met de bronindex alleen in naam. De synoniemindex heeft dezelfde indexwaarden als de bronindex. Een deelindex bevat een subset van de indexwaarden van de bronindex. Met een expressie wordt aangeduid welke indexwaarden van de bronindex ook onderdeel zijn van de deelindex. Deze expressie heeft het type *Bool* en geeft per mogelijke waarde van de deelindex aan of deze onderdeel is van de deelindex of niet. Het is niet verplicht om de index in deze expressie op te nemen.

Het is gebruikelijk om een afgeleide index de naam te geven van de originele index, aangevuld met een nummer.

In het volgende voorbeeld wordt index *a1* als een synoniemindex gedefinieerd van index *a.* Deze synoniemindex bevat dezelfde indexwaarden als index *a.*

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}\}$$ (Grafisch)

```
{a1|a}
```
(ASCII)

In onderstaand voorbeeld wordt index *a1* als een deelindex gedefinieerd van index *a*. Deze deelindex bevat alleen die elementen van de originele index waarvan de waarde ligt tussen de waarden van de variabelen smi en sma.

$$\mathbf{a1} = \left\{ a1 \middle| a1 \in \mathbf{a}.(a1 \geq smi \wedge a1 \leq sma) \right\}$$ (Grafisch)

```
{a1|a:a1 >= smi[] and a1 <= sma[]}
```
(ASCII)

Een synoniemindex is in principe een speciale deelindex met een bijbehorende expressie die altijd de waarde true oplevert: `{a1|a:true}`

Een indexdeclaratie staat in de grafische notatievorm boven de formule. De zo gedefinieerde index is vervolgens meerdere malen in de formule te gebruiken. In de ASCII-vorm moet de index lokaal aan een aggregatie worden gedeclareerd (zie 4.3.9). Dit houdt in dat wanneer de gedefinieerde index meerdere malen in de formule wordt gebruikt, deze mogelijk ook meerdere malen gedefinieerd moet worden (voor iedere aggregatie-expressie waarin de index wordt gebruikt).

In de grafische vorm is het gebruikelijk om met behulp van subscript de indices te vermelden waarvan de index definitie afhankelijk is. Dit bevordert de leesbaarheid. In de ASCII-vorm is dit niet nodig, omdat een index altijd lokaal aan een aggregatie wordt gedeclareerd en er zo altijd duidelijk is van welke indices de declaratie afhankelijk is:

$$\mathbf{l1}_{lf,ol} = \left\{ l1 \mid l1 \in \mathbf{l}.(lvk_{l1} = kok_{lf,ok=vkol_{lf,ol}} \wedge lnk_{l1} = kok_{lf,ok=nkol_{lf,ol}}) \right\}$$

Een indexdeclaratie kan ook recursief zijn. Dit wil zeggen dat het mogelijk is om bijvoorbeeld een deelindex te declareren van een synoniem- of deelindex.

In het volgende voorbeeld is index *a2* gedefinieerd als deelindex van *a1* (die weer een deelindex is van index *a*):

$$\mathbf{a2} = \left\{ a2 \middle| a2 \in \mathbf{a1}.(a2 \neq 10) \right\}$$

Index *a2* is ook direct als deelindex van index *a* uit te drukken:

$$\mathbf{a2} = \left\{ a2 \,\middle|\, a2 \in \mathbf{a}.(a2 \geq smi \wedge a2 \leq sma \wedge a2 \neq 10) \right\}$$

Voorts kan het zijn dat een deelindex leeg is (geen waarden bevat), omdat de bijbehorende expressie alle waarden uitsluit. Een lege deelindex kan als volgt worden gedefinieerd:

```
{a1|a:false}.
```

4.3.6 Constanten

Murray kent drie soorten constanten:

Integer constanten

Gehele getallen, bijvoorbeeld 123. Negatieve getallen worden ook ondersteund.

Float constanten

Gebroken getallen, bijvoorbeeld 123.45. Op dit moment wordt alleen de punt (.) als decimaal scheidingsteken herkend. De exponentnotatie (bijvoorbeeld 1.3e-10) wordt niet ondersteund. Negatieve getallen worden wel ondersteund, bijvoorbeeld -234.56.

Named constanten

Constanten met een naam. Een constante heeft een type en een waarde. De waarde wordt bij het uitrekenen van de expressie gebruikt. Er zijn twee soorten named constanten: de constanten die onderdeel zijn van de taal zelf, **true** en **false**, en een aantal bekende wiskundige constanten. De wiskundige constanten (zoals π) mogen grafisch met hun symbool worden weergegeven. In de ASCII-vorm van MEL moet de naam van deze symbolen worden gebruikt.

De voordelen van named constanten ten opzichte van variabelen zonder indices zijn:

- Een named constante is grafisch symbolisch weer te geven.
- De named constante kan bij het codegenereren worden vertaald naar een aanroep van een functie / constante binnen de target taal in plaats van direct de waarde van de constante. Zo kent het .NET Framework de constante Math.PI en T-SQL de functie pi().

De ondersteunde named constanten zijn te vinden in Appendix A.

4.3.7 Enumeraties

Een enumeratie is een groepering van waarden onder een naam. Aan elke waarde is een label gekoppeld. Een enumeratie-expressie bestaat uit de naam van de enumeratie, gevolgd door het label. Een enumeratie-expressie heeft als type *Integer*. Het resultaat van een enumeratie-expressie is de bij het enumeratielabel behorende waarde.

In de grafische notatievorm mag de naam van de enumeratie worden weggelaten wanneer duidelijk blijft bij welke enumeratie het label behoort.

Voorbeeld:

'female' (Grafisch)

`Gender:'female'` (ASCII)

Een enumeratie-expressie mag alleen in de context van variabelen en indices worden gebruikt. Dit beperkt zich tot het toekennen aan en vergelijken met variabele- en indexexpressies. De variabele of index moet wel aan de enumeratie gelinkt zijn. Een expressie als `Gender:'male'+1` is dus niet toegestaan.

4.3.8 Variabelen

Een variabele wordt gerepresenteerd door een variabele-expressie. Een variabele-expressie bestaat uit de naam van de variabele, gevolgd door een eventuele lijst van indexwaarde-expressies. Een dergelijke indexwaarde-expressie representeert de integerwaarde van de bij een variabele behorende index. De indexwaarde-expressies zijn geordend op naam van de daarbij behorende index. Een variabele zonder indices heeft geen indexwaarde-expressies.

Het type van een variabele-expressie is gelijk aan het type van de variabele (dat door de modelontwerper is gespecificeerd).

Grafisch worden de indexwaarde-expressies in subscript achter de variabele genoteerd. Om de leesbaarheid te bevorderen mag in de grafische vorm een indexwaarde-expressie ook worden genoteerd als *indexnaam* = *indexwaarde-expressie*, met als *indexnaam* de naam van de bij de indexwaarde-expressie behorende index.

In ASCII-vorm wordt een variabele-expressie genoteerd als de naam van de variabele met daarachter een kommagescheiden lijst van indexwaarde-expressies tussen blokhaken. Bij het noteren van een variabele zonder indices worden de blokhaken niet weggelaten.

De variabele-expressie uit het volgende voorbeeld geeft de bevolking (population) voor het jaar 2000.

$pop_{a,g,r,y=2000}$ (Grafisch)

`pop[a,g,r,Year:'2000']` (ASCII)

Onderstaande variabele-expressie betreft een variabele zonder indices. Let erop dat de blokhaken niet worden weggelaten in de ASCII-vorm.

smi (Grafisch)

`smi[]` (ASCII)

Bij het bepalen van de waarde van een variabele-expressie worden eerst de indexwaarde-expressies uitgerekend. Daarna wordt de waarde van de variabele opgevraagd voor deze uitgerekende indexwaarden.

Wanneer een uitgerekende indexwaarde buiten het bereik van de bijbehorende index ligt, is de waarde van de variabele-expressie niet gedefinieerd. Dit houdt in dat voor het model gegenereerde code in een dergelijk geval stopt met een foutmelding.

In het geval de variabele voor een combinatie van indexwaarden geen waarde heeft is het resultaat van de variabele-expressie de bij de variabele behorende defaultwaarde.

4.3.9 Aggregaties

Voor het aggregeren van data kan de aggregatie-expressie worden gebruikt. Er wordt geaggregeerd over één index. Aggregeren over meerdere indices is mogelijk door middel van geneste aggregaties. Een aggregatie-expressie bestaat uit de naam van de aggregatie, een aggregatie-index en een optionele accumulator-expressie.

De aggregatie-index mag niet reeds beschikbaar zijn. Is dit wel het geval, dan moet er een synoniemindex worden gedeclareerd waarover vervolgens kan worden geaggregeerd. De aggregatie-index mag ook een deelindex betreffen. Een zo gedeclareerde index is alleen in de aggregatie-expressie beschikbaar. Zie 4.3.5 voor een beschrijving van indexdeclaraties.

Voor iedere indexwaarde uit de aggregatie-index wordt de accumulatorexpressie uitgerekend. Dit resultaat wordt vervolgens verwerkt in de accumulator van de aggregatie. Dit verwerken verschilt per aggregatie-soort. Een sommatie bijvoorbeeld zal alle individuele resultaten optellen, een minimum zal telkens de huidige accumulator-waarde vergelijken met de uitgerekende waarde en de uitgerekende waarde toekennen aan de accumulator wanneer deze kleiner is dan de huidige waarde van de accumulator.

De beginwaarde van de accumulator is voor elke aggregatie-soort gelijk aan 0. Een aggregatie over een lege index levert dus de waarde 0 op. Het type van de aggregatie komt overeen met het type van de bijbehorende expressie. Bij het ontbreken van de accumulator-expressie is het type van de aggregatie *Integer*.

Een speciale vorm van aggregatie betreft een aggregatie zonder accumulatorexpressie. Dit soort aggregaties zegt iets over de aggregatie-index. Denk hierbij bijvoorbeeld aan het minimum van de index, of het aantal elementen in de index.

MEL ondersteunt de volgende aggregatie-soorten:

Grafisch	ASCII	Expressie	Omschrijving
$\sum$	sum	Verplicht	Sommatie over een index.
$\prod$	product	Verplicht	Uitvermenigvuldigen van een index.
min	min	Optioneel	Het minimum over/van een index.
max	max	Optioneel	Het maximum over/van een index.
count	count	Geen expressie	Het aantal elementen van een index.

In onderstaand voorbeeld wordt er gesommeerd over de indices *a*, *g* en *r*. Aggregeren over meerdere indices vindt genest plaats. De ontstane aggregatie-expressie is alleen nog 'afhankelijk' van index *y*.

$$\sum_{a}\sum_{g}\sum_{r} wp_{a,g,r,y} \cdot dc \qquad \text{(Grafisch)}$$

```
sum({a},sum({g},sum({r},wp[a,g,r,y]*dc[])))
```
(ASCII)

In het volgende voorbeeld wordt de deelindex *sn1* gedeclareerd en vervolgens wordt daarvan het minimum genomen. Grafisch wordt de indexdeclaratie boven de expressie geplaatst. In de ASCII-notatie is de indexdeclaratie onderdeel van de aggregatie-expressie.

$$\begin{array}{l}\mathbf{sn1}_{j,pg,sn} = \left\{ sn1 \mid sn1 \in \mathbf{sn} \mid mum_{j,pg,sn1} \geq 0.001 \right\} \\ \min(\mathbf{sn1}_{j,pg,sn})\end{array} \qquad \text{(Grafisch)}$$

```
min({sn1|sn:mum[j,pg,sn1] >= 0.001})
```
(ASCII)

De volgende pseudo-code geeft aan hoe aggregators worden uitgerekend. Het principe is voor alle aggregatiesoorten gelijk. De aggregatiesoorten verschillen van elkaar in hoe de accumulator expressie wordt verwerkt.

```
accumulator := 0
visited := false

for each element ∈ index do
begin
    if not visited then
    begin
        accumulator := <accumulator expressie>
        visited := true
    else
    begin
        accumulator := <aggregator afhankelijke expressie>
    end
end
```

De volgende tabel geeft voor de door MEL ondersteunde aggregatie-soorten de bijbehorende expressie.

Aggregator	Expressie
sum	`accumulator := accumulator + <expressie>`
product	`accumulator := accumulator * <expressie>`
min over index	`accumulator := min(accumulator,<expressie>)`
max over index	`accumulator := max(accumulator,<expressie>)`
min van index	`accumulator := min(accumulator,element)`
max van index	`accumulator := max(accumulator,element)`
count	`accumulator := accumulator + 1`

4.3.10 Conditie-expressies

Murray ondersteunt het conditioneel uitrekenen van expressies. Een conditie-expressie bestaat uit één of meer paren van expressies en één zogenaamde defaultexpressie. Een expressiepaar bestaat uit een logische-expressie en een resultaatexpressie. De expressieparen worden in volgorde van specificeren doorlopen totdat de logische-expressie van een paar de waarde *true* oplevert. Is dit het geval, dan wordt de resultaatexpressie van het paar uitgerekend en als uitkomst van de conditie-expressie aangeboden. Meerdere logische-expressies mogen tegelijk *true* opleveren en in feite dus elkaar overlappen, er wordt alleen gekeken naar de eerste die *true* oplevert. Wanneer geen enkele logische-expressie de waarde *true* oplevert wordt de defaultexpressie uitgerekend en als resultaat aangeboden.

Onderstaand voorbeeld geeft aan hoe een conditie-expressie wordt genoteerd. In dit geval heeft de conditie-expressie twee paren van expressies. Het resultaat van deze conditie-expressie hangt af van de waarden van indices *i* en *j*. Afhankelijk van deze waarden is het resultaat gelijk aan variabele x, y of z. De defaultexpressie wordt altijd als laatste opgeschreven.

$$\begin{cases} x & i = 0 \\ y & i > 0 \wedge j = 0 \\ z & \end{cases} \qquad \text{(Grafisch)}$$

```
if(i=0,x[],i>0 and j=0,y[],z[])
```
(ASCII)

De volgende pseudo-code geeft weer hoe het bovenstaande voorbeeld wordt geëvalueerd:

```
if (i = 0) then
begin
    result := x
end
else if (i > 0 and j = 0) then
begin
    result := y
end
else
begin
    result := z
end
```

De resultaatexpressies en defaultexpressie zijn niet gebonden aan restricties, elke MEL-expressie kan worden gebruikt. De logische-expressies moeten van het type *Bool* zijn. De resultaatexpressies en de defaultexpressie moeten allen van hetzelfde type zijn. Een uitzondering is wanneer de gebruikte types zich beperken tot *Integer* en *Float*. Het resultaattype is dan *Float*. De integer expressies worden dan impliciet naar *Float* geconverteerd.

4.3.11 Functies

Murray ondersteunt wiskundige functies. Er kunnen alleen functies worden gebruikt die in het systeem zijn gedefinieerd. Het is niet mogelijk om binnen een model functies te definiëren. Een functie is een relatie tussen nul of meer invoerwaarden en één uitvoerwaarde. Functies mogen geen side effects hebben.

Een functieaanroep bestaat uit de naam van de functie gevolgd door een lijst argumentexpressies tussen haakjes. De expressies in de lijst worden gescheiden door een komma. De lijst mag ook leeg zijn, de functie heeft dan geen parameters. Alle argumentexpressies worden uitgerekend voor de functie wordt aangeroepen.

Een functie heeft een type-signatuur. Deze signatuur geeft aan wat de types moeten zijn van de argumentexpressies en het type van het resultaat van de functie.

MEL ondersteunt het overloaden van functies. Dit houdt in dat er meerdere functies met dezelfde naam kunnen bestaan, elk met een andere signatuur. De enige beperking die hierbij geldt is dat functies met dezelfde naam niet mogen verschillen in het resultaattype.

Zo zijn de signaturen van de twee in MEL gedefinieerde min-functies:

$$int \times int \rightarrow int$$
$$float \times float \rightarrow float$$

Het minimum van twee integers levert weer een integerwaarde op. Het minimum van twee floats levert een float waarde op.

Wanneer een functieaanroep-expressie niet voldoet aan een in het systeem gedefinieerde signatuur wordt er getracht om door middel van impliciete conversie toch een match te vinden met de bestaande signaturen. Het minimum van een float en integer levert bijvoorbeeld een float op, aangezien de integere waarde impliciet naar een float waarde gecast wordt.

Wanneer een functie niet is gedefinieerd voor een bepaalde invoer (logaritme of wortel van een negatief getal bijvoorbeeld) is het resultaat 0. Het is aan de modelontwerper om ervoor te zorgen dat een functie geen “foutieve” invoer krijgt (door dit bijvoorbeeld af te vangen met een conditie-expressie).

Functies kunnen grafisch een speciale schrijfwijze hebben. De wortelfunctie bijvoorbeeld wordt grafisch met een wortelteken genoteerd.

Zie appendix A voor een lijst van in MEL ondersteunde functies.

4.3.12 Binaire-expressies

Een binaire-expressie bestaat uit twee operand-expressies gescheiden door een operator. De binnen MEL ondersteunde operatoren staan vermeld in Appendix A. Bij een operator horen één of meerdere signaturen. Zo levert de optelling van twee integer getallen ook weer een integer op. Een deling levert altijd een float waarde op. Ook bij operatoren wordt impliciete typeconversie toegepast wanneer een gevraagde signatuur niet is gedefinieerd.

De deling-operator is een bijzondere operator. Het resultaat van een deling door 0 is 0. Wanneer dit niet is gewenst dient de modelontwerper expliciet de noemer op 0 te controleren in de expressie (en in dat geval een andere waarde te specificeren).

4.3.13 Unaire-expressies

Naast binaire-expressies kent MEL ook unaire-expressies. Een unaire-expressie bestaat uit een (unaire) operator en een operand-expressie. Zie Appendix A voor een lijst van binnen MEL ondersteunde operatoren.

Een negatief getal wordt in MEL gelezen als een unaire-expressie.

4.4 Bepalen van een juiste procesvolgorde

4.4.1 Inleiding

Bij het uitrekenen van een model moeten de formuleprocessen in een juiste volgorde worden uitgevoerd om de correcte modeluitkomsten te verkrijgen. Deze paragraaf beschrijft een algoritme waarmee een[1] juiste volgorde kan worden bepaald. Dit algoritme wordt gebruikt door de Murray codegenerator.

4.4.2 Probleemdefinitie

Een variabele is afkomstig uit de invoer (gerepresenteerd door een invoerproces) of wordt in een formuleproces binnen het model uitgerekend. Binnen een formuleproces wordt één variabele uitgerekend. Dit gebeurt door middel van het uitrekenen van de bij het proces behorende formule. In deze formule worden mogelijk andere variabelen gebruikt. Deze variabelen moeten uitgerekend zijn voordat ze gebruikt kunnen worden. Een variabele afkomstig uit de invoer is per definitie al uitgerekend.

Het probleem bestaat uit het op volgorde plaatsen van de formuleprocessen, zó dat bij uitvoeren van de processen in deze volgorde elke door een proces benodigde variabele voor de gevraagde indexwaarden reeds is uitgerekend in een eerder uitgevoerd proces.

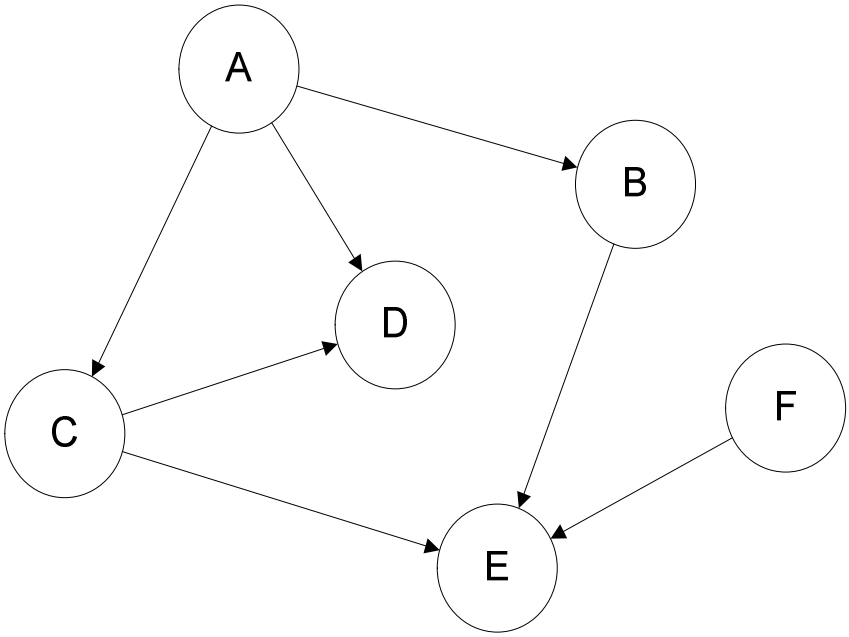
In bovenstaande probleemdefinitie wordt gesproken over “gevraagde indexwaarden”. Dit is van toepassing bij iteratieve processen. Dergelijke processen zijn onderdeel van één of meerdere iteraties. De bijbehorende variabelen worden daarom per combinatie van iteratie-indexwaarden uitgerekend. Iteratieve processen worden dus meerdere malen bezocht.

Onderstaand wordt allereerst een oplossing gepresenteerd voor modelspecificaties zonder iteraties. Vervolgens wordt beschreven hoe ook iteratieve processen op volgorde te plaatsen zijn.

4.4.3 Afbeelden model op graaf

Procesafhankelijkheden binnen een model *zonder* iteraties zijn af te beelden op een gerichte *acyclische* graaf. Elke knoop in de graaf representeert een formuleproces. Er bestaat een gerichte tak tussen knopen A en B in de graaf wanneer het door A gerepresenteerde formuleproces bij het uitrekenen van diens formule de variabele gebruikt die in het door B gerepresenteerde formuleproces wordt uitgerekend. User-defined processen worden hetzelfde behandeld als formuleprocessen. Overige procestypen zijn voor het probleem niet relevant en worden daarom niet meegenomen bij het opstellen van de graaf. Figuur 4-9 geeft een voorbeeld van een dergelijke graaf.

[1] Afhankelijk van het model kunnen er meerdere oplossingen mogelijk zijn.



Figuur 4-9 Voorbeeld van het vertalen van procesafhankelijkheden naar een graaf

4.4.4 Topologische sortering

Het bovenstaande probleem is te herleiden tot het bepalen van een topologische sortering van de opgestelde graaf. Onderstaande definitie is afkomstig uit [10]:

Een topologische sortering van een gerichte graaf G=(V,E) is een lineaire ordening van $v \in V$ zodanig dat als $(u,v) \in E$, u zich in deze ordening vóór v bevindt.

Een topologische sortering kan worden verkregen door het uitvoeren van een depth-first-search (DFS) op de graaf. Wanneer het DFS-algoritme klaar is met een knoop moet de knoop aan het begin van een gelinkte lijst worden toegevoegd. Na het uitvoeren van het DFS-algoritme bevat deze gelinkte lijst de knopen in topologische volgorde. De tijdcomplexiteit van DFS is O(V+E). Toevoegen van een knoop aan het begin van een gelinkte lijst is een O(1) operatie. De tijdcomplexiteit van een topologische sortering is daarom O(V+E).

Er kunnen meerdere topologische sorteringen voor een graaf bestaan. De gevonden sortering is afhankelijk van hoe het DFS-algoritme de graaf afloopt. Een topologische sortering van de graaf uit figuur 4-9 is bijvoorbeeld {A,C,D,B,F,E}. Een andere mogelijke sortering is {F,A,C,B,E,D}.

Een model mag geen circulaire procesafhankelijkheden bevatten. Een graaf die met een dergelijk model correspondeert bevat één of meerdere cykels en heeft daarom geen topologische sortering. Een graaf is acyclisch wanneer een depth-first-search van die graaf geen backedges oplevert. Wanneer er toch een backedge wordt gevonden moet het bovenstaande algoritme worden getermineerd met een foutmelding.

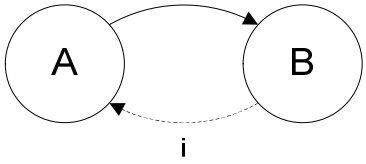
4.4.5 Iteraties

De Murray Specification Language ondersteunt het itereren over processen (zie 4.2.10). De gebruiker geeft met behulp van een *iteratieflow* aan dat een proces invoer krijgt van een proces uit een *vorige* iteratie (dit kan ook hetzelfde proces zijn). Bij een dergelijke flow wordt aangegeven over welke indices wordt geïtereerd. Itereren houdt in dat de variabelen die betrokken zijn bij de iteratie in stappen worden uitgerekend, per combinatie van iteratie-indexwaarden.

Het probleem bestaat uit het detecteren welke processen betrokken zijn bij een iteratie. Daarnaast moeten deze processen in de juiste volgorde uitgevoerd worden, voor iedere indexwaardencombinatie van de iteratie-indices waarover wordt geïtereerd.

Ook bovenstaand probleem kan worden vertaald naar een grafenprobleem. Hiervoor moeten iteraties worden toegevoegd aan de eerder gebruikte graafrepresentatie. Een iteratieflow tussen processen A en B correspondeert met een gerichte tak in de graaf tussen de bij die processen horende knopen. Van deze tak wordt bijgehouden dat het een *iteratietak* betreft. Daarnaast wordt bijgehouden welke indices bij deze iteratietak betrokken zijn. Voor de duidelijkheid worden iteratietakken in de getoonde voorbeelden onderbroken weergegeven en worden daarnaast de betrokken indices bij de iteratietak vermeld.

Figuur 4-10 geeft een voorbeeld van een simpele iteratie tussen processen A en B over index i. Proces A rekent met gegevens die in een vorige iteratie door proces B zijn berekend. Proces B maakt gebruik van de gegevens die proces A in diezelfde iteratie heeft berekend.



Dit vertaalt zich in:

```
for i = min(i) to max(i)
    A(i)
    B(i)
next i
```

Figuur 4-10 Simpele iteratie

Variabelen zijn multidimensionaal en worden per indexwaardencombinatie uitgerekend (zie 4.3.2). Normaal gesproken wordt een variabele in het bijbehorende proces volledig uitgerekend, dat wil zeggen voor het complete bereik van indexwaarden. Wanneer een proces onderdeel uitmaakt van een iteratie wordt de bijbehorende variabele voor wat betreft de iteratieindices alleen voor de huidige geldende iteratie-indexwaarden uitgerekend. De pseudo-code naast figuur 4-10 geeft aan hoe het in dat figuur getoonde iteratievoorbeeld wordt uitgevoerd.

Een iteratieflow is een bijzondere vorm van afhankelijkheid. Bij een normale flow tussen proces A en B *moet* proces A vóór proces B worden uitgerekend. Bij een iteratieflow is dit niet noodzakelijk, mits de processen ook echt onderdeel uitmaken van een iteratie over de bij de iteratieflow behorende indices. Is dit niet het geval, dan moet de betreffende iteratieflow als normale flow worden behandeld. De gebruiker geeft met een iteratieflow aan dat er in het proces waar deze flow naar binnen loopt wat betreft de iteratieindices alleen wordt 'teruggegrepen'.

Op deze manier kunnen bijvoorbeeld jaar-op-jaar-berekeningen gemodelleerd worden. Een goed voorbeeld hiervan is het demografisch prognosemodel. Hierbinnen wordt de populatie in een bepaald jaar berekend aan de hand van de populatie, het aantal geboortes en sterftes en het aantal immigranten en emigranten in het jaar *daarvoor*. Ook geldt bijvoorbeeld dat het aantal geboortes in een bepaald jaar weer afhankelijk is van de (vrouwelijke) populatie in *datzelfde* jaar. Het uitrekenen van de populatie, geboortes, sterftes en dergelijke dient dus per jaar te worden gedaan, anders is er sprake van een cyclische afhankelijkheid en is het model niet uitvoerbaar.

4.4.6 Sterk samenhangende componenten

Sterk samenhangende componenten kunnen worden gebruikt om te detecteren welke processen bij iteraties betrokken zijn.

Een gerichte graaf G=(V,E) is sterk samenhangend wanneer er voor ieder paar knopen u,v $\in$ V een gericht pad bestaat tussen zowel u en v als v en u.

Een sterk samenhangende component van een graaf G is een maximale deelgraaf van G die sterk samenhangend is.

De sterk samenhangende componenten van een graaf G vormen samen een gerichte *acyclische* graaf. Deze graaf heeft de componenten van G als knopen en een tak (X,Y) wanneer X en Y verschillende componenten zijn en X een knoop u bevat en Y een knoop v die verbonden zijn met een tak (u,v) in G. Omdat de zo gevormde graaf geen cykels bevat, is er een topologische sortering van de componenten te bepalen.

Een algoritme om een graaf op te delen in zijn sterk samenhangende componenten is gegeven in [11]. Dit algoritme heeft een tijdcomplexiteit van O(V+E). Het algoritme heeft de eigenschap dat de componenten in omgekeerde topologische volgorde worden gedetecteerd [12]: als er een pad bestaat tussen component C en een ander component C' dan wordt component C' eerst gedetecteerd. Wanneer het algoritme een component heeft gedetecteerd moet deze dus aan het begin van een gelinkte lijst worden geplaatst. Na uitvoering van het algoritme bevat deze gelinkte lijst de componenten in topologische volgorde.

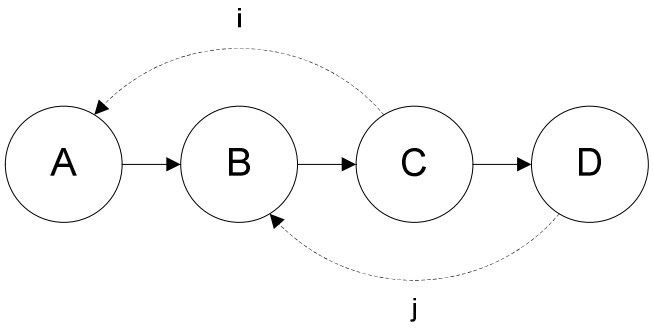
4.4.7 De oplossing

Een sterk samenhangend component bevat één of meer cykels. Er is immers een pad van iedere knoop naar zichzelf. Door iteratietakken als gewone takken te behandelen, ontstaan er circulaire afhankelijkheden in de graaf. Wanneer we ervan uitgaan dat de graaf zonder iteratietakken geen circulaire afhankelijkheden bevat, dan kunnen we na bovenstaande aanpassing aan de sterk samenhangende componenten van de graaf zien welke knopen betrokken zijn bij iteraties.

Wanneer een dergelijk samenhangend component één iteratietak bevat, dan is de oplossing simpel. Door deze tak te verwijderen kunnen alle knopen van het component op volgorde geplaatst worden. Deze knopen itereren dan over de bij de iteratietak horende indices.

In het geval een component meerdere iteratietakken bevat is de oplossing minder eenvoudig. Er is dan mogelijk sprake van meerdere iteraties, die genest en/of overlappend kunnen zijn. Het algoritme moet detecteren over welke index/indices de knopen van het component moeten itereren zodat de knopen weer (deels) op volgorde geplaatst kunnen worden. In het geval van geneste iteraties moet het algoritme dus de buitenste iteratie vinden. Bij overlappende iteraties zijn er meerdere mogelijkheden en moet er een keuze gemaakt worden.

Figuur 4-11 geeft een voorbeeld van twee elkaar overlappende iteraties. Er zijn twee oplossingen mogelijk voor deze graaf. Eén van de twee indices moet worden gekozen als buitenste iteratieindex. Kiezen we index i, dan itereert knoop A over i en de overige knopen over i *en* j (oplossing 1). Kiezen we voor index j, dan itereren knopen A, B en C over zowel j als i en knoop D alleen over j (oplossing 2). Beide oplossingen zijn even efficiënt.



Figuur 4-11 Overlappende iteratie

mogelijke oplossing 1:

```
for i = min(i) to max(i)
    A(i)
    for j = min(j) to max(j)
        B(i,j)
        C(i,j)
        D(i,j)
    next j
next i
```

mogelijke oplossing 2:

```
for j = min(j) to max(j)
    for i = min(i) to max(i)
        A(j,i)
        B(j,i)
        C(j,i)
    next i
    D(j)
next j
```

4.4.8 Het algoritme

Het algoritme bestaat uit het recursief opdelen van de graaf in sterk samenhangende componenten. Het algoritme is klaar wanneer alle componenten bestaan uit één knoop. Het algoritme begint met het bepalen van de sterk samenhangende componenten van de graaf. Tijdens het bepalen hiervan worden de componenten al in topologische volgorde geplaatst. De zo ontstane componenten worden in volgorde langsgelopen. Wanneer een component bestaat uit slechts één knoop, dan wordt deze knoop achter aan de lijst met processen toegevoegd. Bestaat het component uit meerdere knopen, dan is er sprake van een iteratie.

De volgende stap in het algoritme is dan het bepalen over welke indices de knopen in het component itereren. Dit wordt bepaald door systematisch alle mogelijke indexcombinaties te genereren die te maken zijn uit de bij de iteratietakken behorende indices en deze één voor één uit te proberen. De betreffende indices worden verwijderd van de iteratietakken in de component. Wanneer een iteratietak na verwijdering geen indices meer heeft, dan wordt de iteratietak zelf ook verwijderd. Als de zo ontstane graaf weer verder is op te delen in sterk samenhangende componenten is een juiste combinatie van indices gekozen. De gekozen indices worden toegevoegd aan de lijst van iteratie-indices van iedere knoop in de graaf. Vervolgens wordt het algoritme weer uitgevoerd op de zo ontstane componenten.

In het geval dat een combinatie van indices niet leidt tot een verdere opdeling van de graaf wordt de verwijdering van indices en iteratietakken ongedaan gemaakt en wordt er opnieuw begonnen met een andere indexcombinatie. Is er na uitproberen van alle mogelijke indexcombinaties geen oplossing gevonden, dan bezit de graaf circulaire afhankelijkheden en is het niet mogelijk om de knopen van de graaf op volgorde te plaatsen. Het model is dan niet valide.

De tijdcomplexiteit van het algoritme is exponentieel in het aantal verschillende indices behorende bij de iteratietakken van de graaf. Dit is in principe geen probleem omdat het aantal iteraties en bijbehorende iteratie-indices normaliter beperkt blijft. Wanneer de gegenereerde indexcombinaties van klein naar groot worden doorlopen zal in veruit de meeste gevallen snel een oplossing gevonden worden.

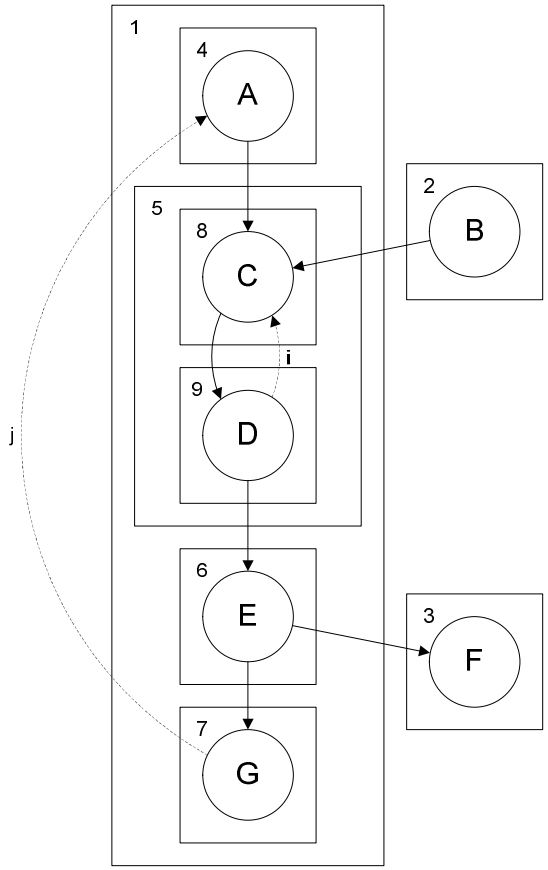
Het is mogelijk dat beide uiteinden van een iteratietak met dezelfde knoop zijn verbonden (eigeniteratie). Dit houdt in dat een variabele bij de berekening eerder uitgerekende waarden van zichzelf benodigt. In het demografisch prognosemodel bijvoorbeeld wordt de grootte van de bevolking in een bepaald jaar onder andere bepaald door de grootte van de bevolking in het jaar daarvoor (zie ook 1.3 in figuur 4-13).

Dergelijke eigeniteraties worden niet gedetecteerd bij het bepalen van sterk samenhangende componenten. Er wordt immers gesteld dat elke knoop zichzelf per definitie kan bereiken, of er nu sprake is van een eigeniteratie of niet. Daarom moet het algoritme voor elk component bestaande uit één knoop controleren of er sprake is van een eigeniteratie. In een dergelijk geval moeten de bijbehorende iteratie-indices worden toegevoegd aan de iteratie-indices van de graaf (in het geval dat deze nog niet aanwezig waren).

4.4.9 Voorbeelden

Figuur 4-12 toont een graaf met 7 processen en 2 iteraties. In totaal heeft het algoritme recursief 9 sterk samenhangende componenten geïdentificeerd. Deze zijn in de figuur aangegeven door een rechthoek met in de linker bovenhoek het nummer van het component in de volgorde van detecteren. De graaf bestaat uit drie sterk samenhangende componenten. Component 1 wordt verder opgedeeld na verwijderen van de iteratie over j. Component 5 wordt verder opgedeeld door de iteratie over index i te elimineren.

Naast de figuur is een mogelijke volgorde van uitrekenen aangegeven. Knopen B en F zijn geen onderdeel van een iteratie. De andere knopen zijn onderdeel van een iteratie over index j. Knopen C en D zijn tevens onderdeel van een iteratie over index i.

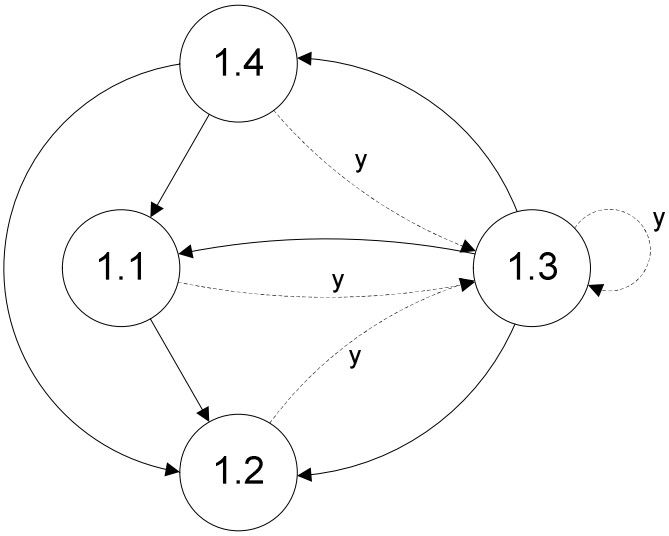


Dit vertaalt zich in:

```
B
for j = min(j) to max(j)
  A(j)
  for i = min(i) to max(i)
    C(i,j)
    D(i,j)
  next i
  E(j)
  G(j)
next j
F
```

Figuur 4-12 Opdelen in sterk samenhangende componenten

Figuur 4-13 toont de versimpelde afhankelijkheidsgraaf van het demografisch prognosemodel. Deze graaf heeft meerdere iteratietakken die allen teruggrijpen op index y. De graaf is in zijn geheel sterk samenhangend. Door te itereren over index y vervallen alle iteratietakken en zijn de knopen in de graaf op volgorde te plaatsen, en wel in de volgorde 1.3, 1.4, 1.1 en ten slotte 1.2. Bij knoop 1.3 is er sprake van een eigeniteratie. Aangezien alle knopen reeds itereren over index y hoeft hier verder geen rekening mee gehouden te worden.

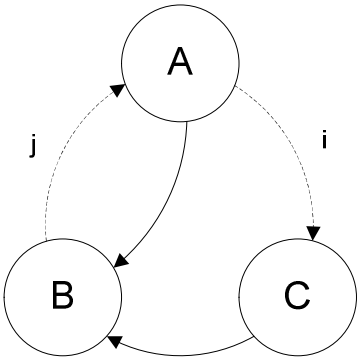


Figuur 4-13 Afhankelijkheidsgraaf demografisch prognosemodel

4.4.10 Observaties

Vinden van een betere oplossing

Wanneer meerdere oplossingen mogelijk zijn kiest het algoritme altijd de *eerste* oplossing die voldoet. Dit hoeft niet altijd de beste oplossing te zijn. Het vinden van een beste oplossing is lastig. Hiervoor moeten alle mogelijke oplossingen worden gegenereerd en met elkaar worden vergeleken, waarbij per oplossing het totaal aantal iteratiestappen moet worden bepaald.



De afhankelijkheidsgraaf in figuur 4-14 heeft twee mogelijke oplossingen. Itereren over index j geeft direct een oplossing, terwijl het itereren over i ook nog een geneste iteratie over j tot gevolg heeft. Het mag duidelijk zijn dat de eerste oplossing aantrekkelijker is. Bij het verwijderen van iteratietak (B,A) verdwijnen ook de circulaire afhankelijkheden in de graaf. Iteratietak (A,C) moet in dat geval als gewone tak worden behandeld.

Figuur 4-14 Meerdere iteraties mogelijk

Het algoritme dient feitelijk alleen nog te kijken naar iteratietakken die van zichzelf resulteren in een circulaire afhankelijkheid in de graaf. Iteratietakken die hier niet aan voldoen kunnen als gewone tak worden beschouwd. Wanneer dit niet gebeurt, kan het zijn dat er een minder efficiënte oplossing gevonden wordt.

Het in 4.4.8 beschreven algoritme hoeft niet te worden aangepast wanneer van te voren wordt bepaald welke iteratietakken *echte* iteratietakken zijn. Dit kan door te starten met de graaf zonder iteratietakken. Van deze graaf worden de sterk samenhangende componenten bepaald. Als de graaf één of meerdere componenten bevat, dan is er sprake van circulaire afhankelijkheden en is er geen oplossing. In het andere geval wordt vervolgens iedere iteratietak van de originele graaf één voor één aan de graaf toegevoegd. Per toevoeging worden de sterk samenhangende componenten van deze graaf bepaald. Wanneer er geen component is gevonden wordt de iteratietak vervangen door een normale tak in de graaf. Is er wel een component gevonden dan wordt de iteratietak weer uit de graaf verwijderd. Wanneer alle iteratietakken zo getest zijn worden de echte iteratietakken nogmaals toegevoegd aan de graaf. Vervolgens wordt het algoritme uit 4.4.8 als gebruikelijk toegepast op de graaf.

De hierboven beschreven aanpassing elimineert een aantal oplossingen voor de graaf die *over het algemeen* minder efficiënt zijn. Hierdoor wordt de kans groter dat het algoritme uit 4.4.8 een betere oplossing vindt.

Verantwoordelijkheid van de gebruiker

De gebruiker is verantwoordelijk voor het synchroon houden van de iteratieflows met de formules; de iteratie-eigenschappen van een formule moeten corresponderen met die van de bijbehorende iteratieflow(s).

Verder dient de gebruiker rekening te houden met de zogenaamde 0-iteratie. In de eerste iteratiestap is er geen vorige iteratiestap en zijn er dus ook geen waarden uitgerekend. De gebruiker wordt geacht dit randgeval met bijvoorbeeld een conditionele expressie af te handelen. Onderstaande formule uit het demografisch prognosemodel (1.3.1) geeft hier een voorbeeld van.

$$pop_{a,\,g,\,y,\,r} = \begin{cases} pop0_{a,g,r} & y = \min(\mathbf{y}) \\ bir_{g,r,y-1} & y > \min(\mathbf{y}) \wedge a = \min(\mathbf{a}) \\ pop_{a-1,g,r,y-1} - dea_{a-1,g,r,y-1} + rim_{a-1,g,r,y-1} - rem_{a-1,g,r,y-1} & \end{cases}$$

Soms kan de gebruiker voor wat betreft het kiezen van iteratieflows een keuze maken uit meerdere opties. In het geval van bovenstaande formule kan de gebruiker ofwel kiezen voor iteratieflows over index y ofwel iteratieflows over index a (kiezen voor de ene elimineert in feite de andere). De gebruiker moet voor zichzelf goed duidelijk hebben wat wenselijk is.

5. Murray framework

5.1 Inleiding

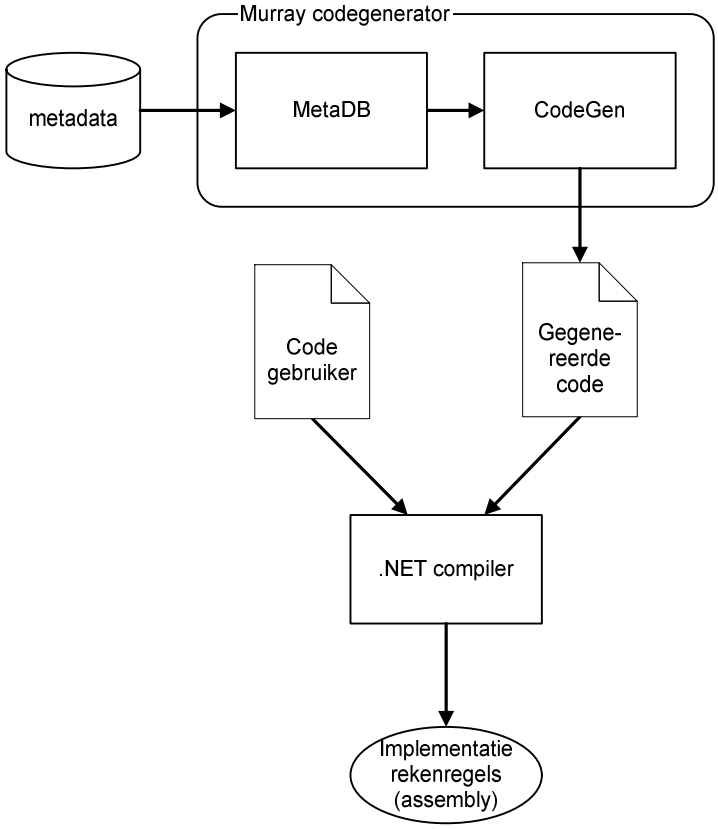
In dit hoofstuk wordt de globale architectuur van het Murray framework beschreven. Ook worden hier enkele onderdelen van het framework in detail besproken, namelijk de MetaDB en DataDB onderdelen. Voor een inleidende beschrijving van het Murray framework wordt verwezen naar paragraaf 3.2.

5.2 Architectuur

Het Murray framework bestaat uit drie onderdelen, te weten:

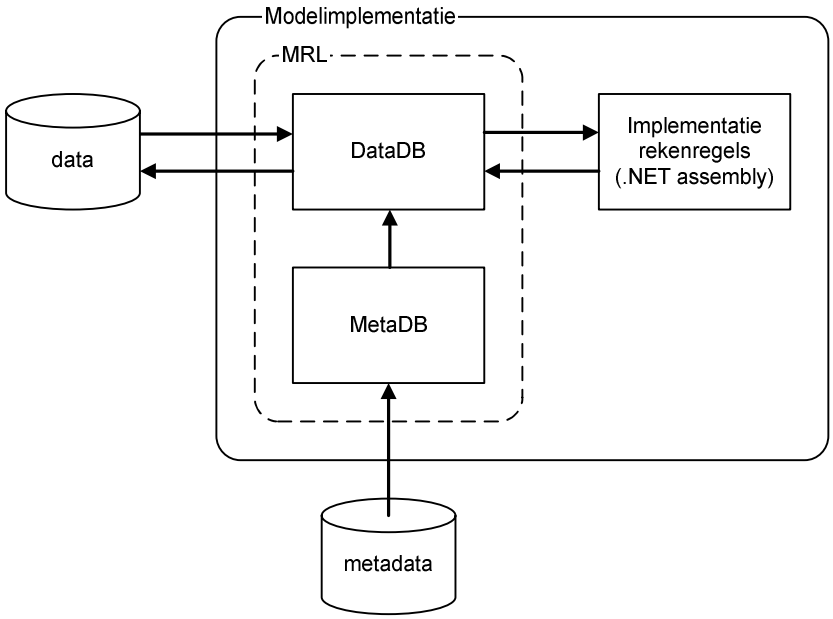
- MetaDB, dit onderdeel beheert rekenmodelspecificaties in MSL-formaat en bijbehorende metadata;
- CodeGen, het frameworkonderdeel verantwoordelijk voor codegeneratie voor in de MetaDB opgeslagen rekenmodelspecificaties;
- DataDB, hiermee wordt efficiënte toegang tot modelgegevens verkregen.

Figuur 5-1 laat zien hoe de MetaDB en CodeGen onderdelen samenwerken bij het genereren van .NET broncode uit rekenmodelspecificaties. Een rekenmodelspecificatie wordt door de MetaDB uit een databron ingelezen en aan het CodeGen onderdeel verder geleid. Dit onderdeel genereert broncode op basis van de ingelezen rekenmodelspecificatie en schrijft deze weg naar een bestand. Samen met eventuele code van de gebruiker vormt dit de input van de .NET compiler behorende bij de taal waarin de broncode is gegenereerd. Compileren van de broncode levert een .NET assembly op waarin de rekenregels van het betreffende rekenmodel zijn geïmplementeerd.



Figuur 5-1 Architectuur Murray framework: Murray codegenerator

Figuur 5-2 toont de relatie tussen de MetaDB en DataDB onderdelen van het Murray framework bij het uitrekenen van een rekenmodel. De rekenmodelimplementatie heeft via de DataDB toegang tot de modeldata. De DataDB maakt op zijn beurt gebruik van de MetaDB om toegang te verkrijgen tot de metadata van het rekenmodel. Bij het draaien van een modelimplementatie zijn alleen de MetaDB en DataDB onderdelen beschikbaar, onder de noemer Murray Runtime Library.



Figuur 5-2 Architectuur Murray framework: Murray Runtime Library (MRL)

De Murray codegenerator is ook in staat om T-SQL statements te genereren voor een rekenmodelspecificatie. Dit gaat analoog aan het genereren van .NET broncode. De zo gegenereerde statements moeten worden uitgevoerd in een Microsoft SQL Server database met modeldata. Er wordt geen gebruik gemaakt van het DataDB onderdeel van het Murray framework. De berekeningen vinden direct plaats op de in SQL Server opgeslagen modelgegevens.

Het MetaDB onderdeel wordt in de volgende paragraaf uitvoerig beschreven. Paragraaf 5.4 gaat dieper in op het DataDB onderdeel van het framework. In hoofdstuk 6 worden alle ins en outs gegeven van het CodeGen gedeelte.

5.3 MetaDB

5.3.1 Inleiding

De Murray MetaDB beheert MSL specificaties en metadata (eigenschappen) van de in deze specificaties gebruikte entiteiten. Dit is een uitbreiding ten opzichte van de Lionfish MetaDB, waarin slechts modelmetadata zijn opgeslagen.
MetaDB gegevens kunnen in verschillende (typen van) databronnen zijn opgeslagen. Op dit moment wordt alleen Microsoft SQL Server 2000 ondersteund voor de opslag van de MetaDB gegevens.

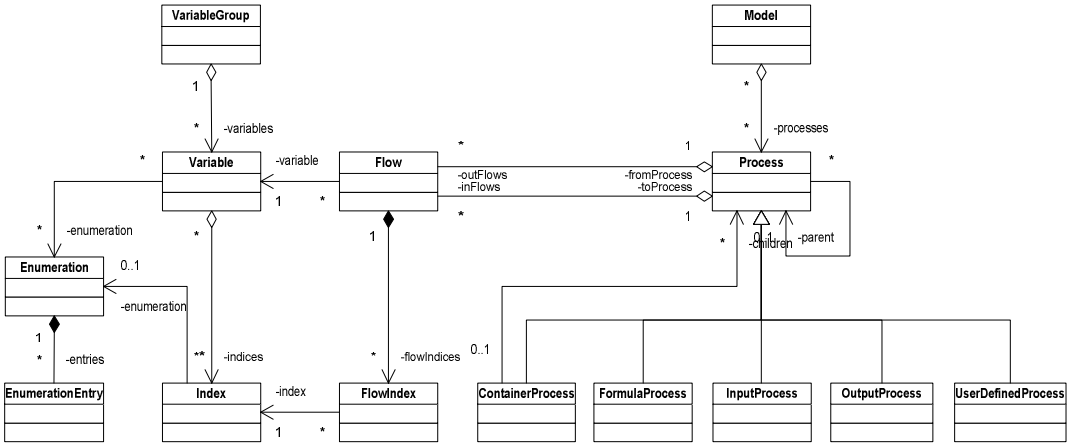
De MetaDB vormt de basis van het Murray framework. De gegevens in de MetaDB worden gebruikt bij:

- het beheren en tonen van rekenmodelspecificaties en bijbehorende metadata;
- het beheren en tonen van rekenmodelgegevens;
- het automatisch genereren van rekenmodelimplementaties, en
- het uitvoeren van rekenmodelimplementaties.

De MetaDB klassen bieden programmatische toegang tot de in de MetaDB opgeslagen informatie.

5.3.2 Overzicht MetaDB klassen

De *MetaDbProvider*-klasse is verantwoordelijk voor het correct inlezen van de gegevens uit de MetaDB naar een bruikbare objectstructuur. De (instanties van) klassen waaruit deze objectstructuur is opgebouwd worden getoond in figuur 5-3. Bij het inlezen wordt gecontroleerd of de in de MetaDB opgeslagen informatie syntactisch en semantisch correct is. Een uitzondering hierop vormen de MetaDB opgeslagen formules. Deze worden alleen tijdens het codegeneratieproces op syntax en semantiek gecontroleerd. Bij het inlezen van de formules door de *MetaDbProvider*-klasse wordt slechts gecontroleerd of er voor een formuleproces een formule in de MetaDB aanwezig is.
De *MetaDbProvider*-klasse is een abstracte klasse. In de configuratie van een applicatie kan worden aangegeven welke concrete implementatie (subklasse) van de *MetaDbProvider*-klasse moet worden gebruikt om gegevens uit de MetaDB te lezen. De gebruiker kan zo dus runtime kiezen uit wat voor een type databron de MetaDB-gegevens gelezen dienen te worden.
Op dit moment is er één concrete klasse in het Murray framework gedefinieerd: de *SqlMetaDbProvider*-klasse. Deze bevat functionaliteit om MetaDB-gegevens uit een SQL Server 2000 instantie te lezen.

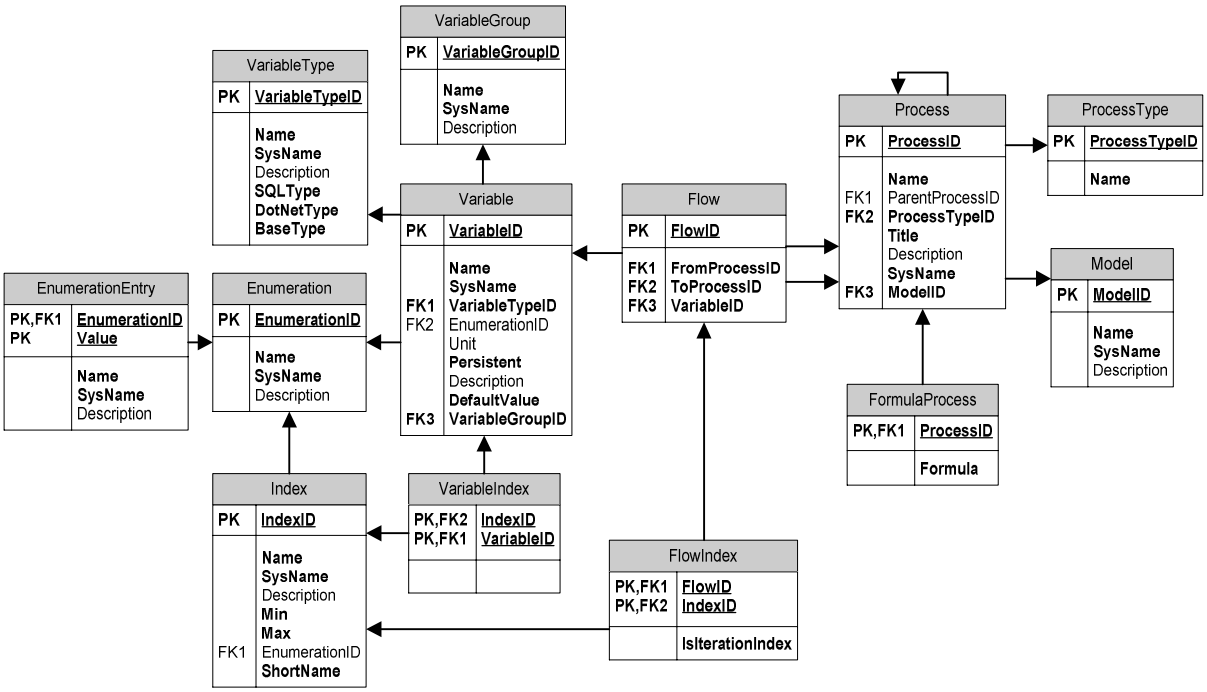


Figuur 5-3 MetaDB klassen

Zoals in Figuur 5-3 te zien is heeft iedere MSL entiteit een corresponderende MetaDB klasse. Er gaat geen informatie verloren bij het vertalen van een MSL specificatie naar een objectstructuur bestaande uit MetaDB klasse-instanties. Instanties van de *Variable*, *VariableGroup*, *Index*, *Enumeration* en *EnumerationEntry* klassen kunnen worden gedeeld door (objectrepresentaties van) modellen. Verschillende modellen kunnen immers gebruik maken van dezelfde variabelen en indices. Voor processen geldt dit bijvoorbeeld niet. Een proces met dezelfde naam kan in meerdere modellen voorkomen. Toch gaat het hier telkens om een ander proces.

5.3.3 Databaserepresentatie

Figuur 5-4 toont het ERD van de Murray MetaDb zoals het in SQL Server 2000 is geïmplementeerd.



Figuur 5-4 Databaserepresentatie MetaDB

5.4 DataDB

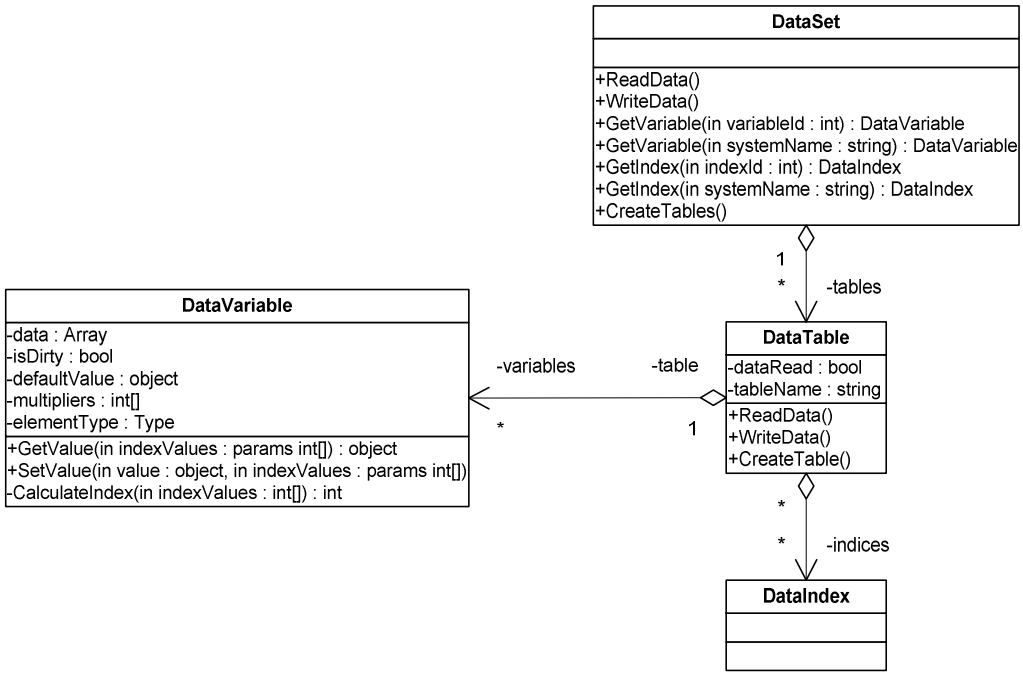
5.4.1 Inleiding

De DataDB klassen bieden toegang tot modelgegevens. Er is functionaliteit aanwezig om modelgegevens uit een databron in te lezen naar geheugen, modelgegevens in geheugen te bewerken en modelgegevens vanuit geheugen te synchroniseren met een databron. Op dit moment wordt alleen SQL Server 2000 als databron ondersteund.

De DataDB klassen dienen als een Data Access Layer waardoor rekenmodelimplementaties niet direct met databronnen hoeven te communiceren. Rekenmodelimplementaties kunnen zo met gegevens uit verschillende (typen) databronnen werken, zonder aanpassingen aan de code. Daarnaast hoeft een rekenmodelimplementatie zich alleen nog bezig te houden met diens kerntaak: het rekenen zelf.

5.4.2 Overzicht DataDB klassen

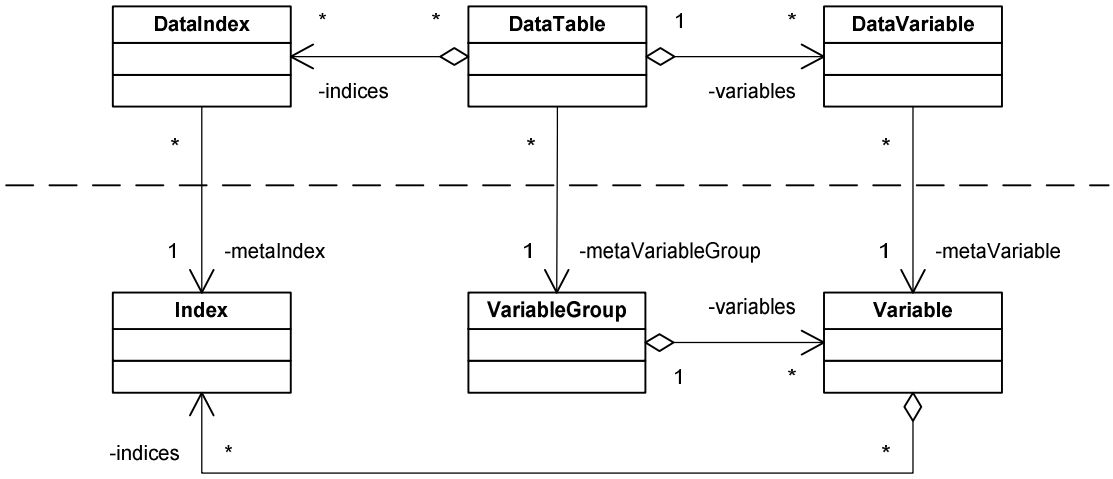
Het klassediagram in figuur 5-5 geeft de belangrijkste DataDB klassen weer met hun relevante velden en methoden. De getter-/setter-methoden die de private fields beschikbaar maken zijn niet afgebeeld.



Figuur 5-5 DataDB klassen

De *DataSet*-klasse beheert de verzamelingen van *DataTable*-, *DataVariable*- en *DataIndex*-objecten. De *DataSet*-klasse krijgt in de constructor een instantie van de *MetaDbProvider* klasse mee. De DataDB klassen hebben een verwijzing naar hun equivalente MetaDB-klasse. Voor ieder *Index*-object in de MetaDB wordt een corresponderend *DataIndex*-object aangemaakt. Hetzelfde geldt voor het aanmaken van *DataVariable*-objecten aan de hand van de in de MetaDB aanwezige *Variable*-objecten. Figuur 5-6 toont de relatie tussen de DataDB klassen en de MetaDB klassen.

DataVariable-objecten waarvan de corresponderende *Variable*-objecten dezelfde indices hebben en bovendien lid zijn van dezelfde groep, worden aan dezelfde *DataTable*-instantie gekoppeld.



Figuur 5-6 Relatie DataDB (boven) - MetaDB (onder) klassen

5.4.3 Rekenen met een DataVariable

Een instantie van de *DataVariable*-klasse beheert zijn eigen data. Deze (multidimensionale) data zijn in de klasse opgeslagen in een ééndimensionale array.

In de constructor van de klasse wordt met behulp van reflection het benodigde type opgezocht (in de MetaDB is dit type als string gespecificeerd). De defaultwaarde wordt uit de MetaDB gelezen (tevens als string) en wordt geconverteerd naar een object van het daarvoor bepaalde type. Vervolgens wordt er een array aangemaakt van dit type met de gevraagde grootte, gevuld met defaultwaarden.

Met de *GetValue* methode kan een waarde uit de interne array worden opgehaald. Met behulp van de *CalculateIndex* methode wordt de juiste positie in de array berekend voor de doorgegeven indexwaarden. Deze laatste methode berekent de juiste positie in de ééndimensionale array door gebruik te maken van multipliers. Het wegschrijven van waarden via de *SetValue* methode gebeurt op een zelfde manier. De *DataVariable* klasse beschikt ook over een indexer die onderhuids de *GetValue* en *SetValue* methoden aanroept. Het grote voordeel hiervan is dat code veel leesbaarder wordt: in de code kan het *DataVariable*-object worden aangesproken als gewone (multidimensionale) array. Figuur 5-7 toont de twee verschillende manieren om een waarde aan een variabele toe te kennen. De *DataVariable* klasse houdt zelf bij of zijn data zijn gewijzigd.

```
DataVariable someDataVariable = dataSet.GetVariable("ASystemName");

someDataVariable.SetValue(4321, 1, 2, 3, 4);
someDataVariable[1,2,3,4] = 1234;
```

Figuur 5-7 Voorbeeldaanroep DataVariable klasse

5.4.4 Synchronisatie met een databron

De *DataSet* klasse coördineert het lezen van data uit en schrijven naar een databron. De gebruiker kan een *DataSet*-object de opdracht geven data in te lezen of weg te schrijven. De *ReadData*-methode van de *DataSet*-klasse roept simpelweg de *ReadData*-methode aan van ieder bijbehorend *DataTable* object. Hetzelfde geldt voor de *WriteData* methode.

De *DataTable* klasse bevat functionaliteit om data te synchroniseren met een databasetabel. Dit is mogelijk via de *ReadData* en *WriteData* methoden. De *DataTable* klasse zorgt ervoor dat de bijbehorende *DataVariable* instanties voorzien worden van de juiste data uit de database(tabel).

Inlezen data

Voordat daadwerkelijk data worden ingelezen uit een tabel wordt er gekeken hoeveel rijen de databasetabel bevat. Wanneer de tabel volledig is gevuld (elke mogelijke indexwaardencombinatie is aanwezig) worden de data direct in de arrays van de bijbehorende *DataVariable*-instanties ingelezen. In het andere geval worden ook de indexwaarden ingelezen en worden de *DataVariable*-instanties via de *SetValue* methode voorzien van de juiste data. Dit laatste is aanzienlijk langzamer, wegens de extra methodcalls en het intern berekenen van de juiste positie in de data array. NULL-waarden uit de database worden vervangen door de defaultwaarde van een variabele.

Wegschrijven data

De *WriteData* methode van de *DataTable* klasse loopt alle bijbehorende DataVariable-instanties af en schrijft de bijbehorende data, indien gewijzigd, naar de bijbehorende databasetabel. Indien voor een bepaalde combinatie van indexwaarden alle variabelen de defaultwaarde hebben, wordt er voor deze combinatie niets naar de database geschreven. Defaultwaarden worden verder weggeschreven naar de database als NULL waarden.

De *WriteData* methode gebruikt standaard INSERT statements om data naar de databasetabel weg te schrijven. Wanneer de bijbehorende variabelen veel indices hebben is dit proces vanwege de grote hoeveelheid data bijzonder traag. Daarom is de mogelijkheid ingebouwd om de data in één keer via XML naar de database te sturen. Het zo versturen van bulkdata gaat vele malen sneller dan via individuele INSERT statements[2]. Het via XML versturen van data is geïmplementeerd met de stored procedure `usp_InsertXML` (Figuur 5-8).

[2] Het .NET Framework versie 1.1 maakt helaas geen gebruik van de bulk insert mogelijkheden van Microsoft SQL Server. In versie 2.0 is een bulk insert operatie mogelijk met behulp van de SqlBulkInsert klasse.

```
dbo.usp_InsertXML
(
	@TableName VARCHAR(254),
	@Data NTEXT,
	@SchemaDeclaration VARCHAR(8000) = NULL
)
```

Figuur 5-8 Signatuur usp_InsertXML stored procedure

De parameters van deze stored procedure zijn als volgt:

Parameter	Omschrijving
@TableName	Naam van de databasetabel
@Data	XML representatie van de tabelinhoud
@SchemaDeclaration	Schema waarmee opgegeven kan worden hoe de diverse velden in de tabel vanuit het XML document worden gevuld. Deze parameter is optioneel.

Figuur 5-9 geeft een voorbeeld van de XML-representatie die door een *DataTable*-object wordt doorgegeven aan de stored procedure.

```
<ROOT>
   <AOWModel_a_g_r_y Age="0" Gender="1" Region="1" Year="0" WorkingPopulation="1.23"
   WAOPopulation="3.21" />
   <AOWModel_a_g_r_y Age="0" Gender="1" Region="1" Year="1" WorkingPopulation="4.56"
   WAOPopulation="6.54" />
   ...
</ROOT>
```

Figuur 5-9 Voorbeeld van een XML-representatie

Zoals uit het voorbeeld blijkt is XML niet het meest efficiënte formaat om data in over te sturen. Met behulp van een schema is de hoeveelheid data al drastisch te verminderen. Het schema uit figuur 5-9 definieert een mapping tussen de velden in de XML-data en de velden in de databasetabel. De attribuutnamen in de XML worden zo kort mogelijk gehouden. Dit levert een aanzienlijke ruimtebesparing op. Vergelijk figuur 5-9 en figuur 5-11 en zie het verschil.

```
(
    Age                 int     '@i0',
    Gender              int     '@i1',
    Region              int     '@i2',
    Year                int     '@i3',
    WorkingPopulation   float   '@v1',
    WAOPopulation       float   '@v2'
)
```

Figuur 5-10 Voorbeeld van een schemadefinitie

```
<ROOT>
   <AOWModel_a_g_r_y i0=”0” i1=”1” i2=”1” i3=”0” v0=”1.23” v1=”3.21” />
   <AOWModel_a_g_r_y i0=”0” i1=”1” i2=”1” i3=”1” v0=”4.56” v1=”6.54” />
   ...
</ROOT>
```

Figuur 5-11 Voorbeeld van een efficiëntere XML-representatie

5.4.5 Databaserepresentatie

Een Murray DataDB-database bevat in- en uitvoerdata voor modellen. Deze wordt gebruikt tijdens:

- het bekijken en aanpassen van modelinvoer;
- het berekenen van een modelh
- het visualiseren van model(data).

Een Murray DataDB-database bevat twee typen tabellen:

Indextabellen

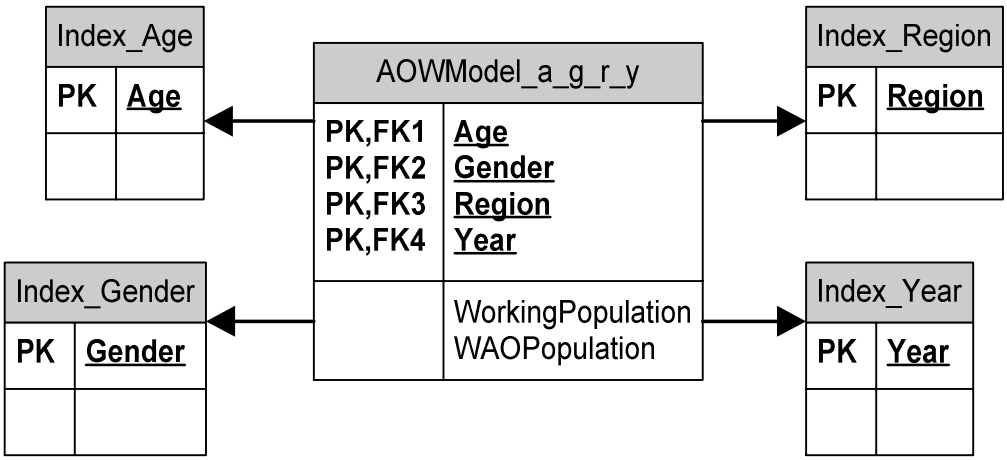
Per index wordt een tabel aangemaakt met daarin (één veld met) de mogelijke waarden van de index. De naam van deze tabel bestaat uit de string “Index” gevolg door een underscore en de naam van de index. Het enige veld in de tabel is tevens primary key van de tabel (de waarden zijn dus zowel uniek als niet NULL). Er mogen geen indexwaarden ontbreken in de tabel; alle indexwaarden tussen de minimum- en maximumwaarde van de index moeten aanwezig zijn.

Tabellen met variabelen

Variabelen in dezelfde groep met gelijke indices komen in dezelfde tabel. De naam van deze tabel bestaat uit de naam van de groep, gevolgd door de korte namen van de bijbehorende indices, op alfabetische volgorde en gescheiden door underscores. Deze tabel bevat ook relaties (foreign keys) met (eventuele) indextabellen. De indexvelden vormen de samengestelde primary key van de tabel. Indexvelden mogen geen NULL waarden bevatten. Variabele velden mogen wel NULL waarden bevatten. Een NULL waarde in de tabel moet worden gelezen als “voor deze indexcombinatie heeft deze variabele de defaultwaarde”. Wanneer een bepaalde indexcombinatie voor elke Variabele uit de tabel de defaultwaarde oplevert (=NULL) hoeft deze combinatie niet in de tabel te worden opgenomen. De veldvolgorde in de tabel is als volgt gedefinieerd: indexvelden (op alfabetische volgorde), variabele velden (op alfabetische volgorde). De veldnamen in de tabel komen overeen met de systeemnamen van de indices en variabelen in de Meta Database (MetaDB).
Een tabel zonder indices heeft geen primary key. Deze tabel bevat maximaal één rij met waarden.

Voorbeeld

Het volgende voorbeeld ERD diagram (figuur 5-12) is afkomstig uit de voor het AOW Model gegenereerde database. De tabel *AOWModel_a_g_r_y* bevat de data voor de variabelen *WorkingPopulation* en *WAOPopulation*. Deze twee variabelen hebben gelijke indices (*Age, Gender, Region* en *Year*) en zijn lid van dezelfde groep (*AOWModel*). Daarom worden deze twee variabelen in dezelfde tabel opgenomen.



Figuur 5-12 DataDB voorbeeld

6. Codegeneratie

6.1 Inleiding

In dit hoofdstuk wordt de Murray codegenerator beschreven. De Murray codegenerator vertaalt MSL-specificaties in MetaDB-formaat naar uitvoerbare broncode, namelijk .NET broncode of T-SQL statements.

De Murray codegenerator is op te splitsen in twee delen: een target-onafhankelijk deel (frontend) en een target-specifiek deel (backend). In het frontend wordt een modelspecificatie ingelezen uit de MetaDB, gecontroleerd op correcte syntax en semantiek en omgezet naar een bruikbare objectrepresentatie die eenvoudige vertaling naar de verschillende targets mogelijk maakt. In het backend wordt de uit het frontend verkregen objectrepresentatie omgezet naar efficiënte broncode voor het door de gebruiker gewenste target.

Paragraaf 6.2 beschrijft het frontend van de Murray codegenerator. In paragraaf 6.3 wordt het backend besproken. Paragrafen 6.4 en 6.5 behandelen het codegenereren voor respectievelijk het .NET Framework en SQL Server 2000.

6.2 Frontend

6.2.1 Inleiding

Het frontend is verantwoordelijk voor het efficiënt en correct omzetten van een MSL-specificatie naar een voor codegeneratie bruikbare objectrepresentatie van deze specificatie. De volgende stappen worden door het frontend uitgevoerd voordat de controle aan het backend wordt overgedragen:

1. inlezen van de gewenste MSL-specificatie uit de MetaDB;
2. het aan de hand van de MetaDB-objectrepresentatie opbouwen van een nieuwe objectrepresentatie waarin formules naar expressiebomen zijn vertaald;
3. het doen van de juiste declaraties voor de target waarvoor code wordt gegenereerd;
4. semantische controle van de objectrepresentatie;
5. bepalen van een juiste uitvoervolgorde van de processen in het model.

In de volgende paragrafen worden bovenstaande stappen beschreven.

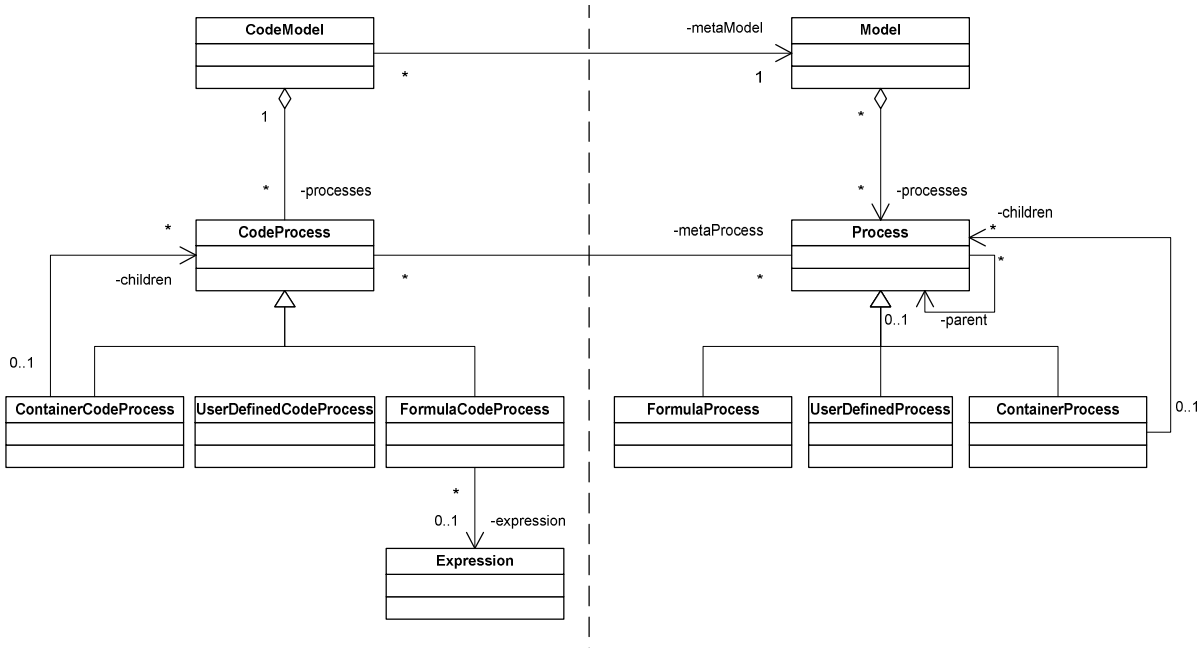
6.2.2 Inlezen MSL-specificatie

Als eerste stap wordt de gewenste MSL-specificatie uit de MetaDB ingelezen. Dit werd reeds beschreven in paragraaf 5.3. Deze stap resulteert in een MetaDB-objectrepresentatie van de gewenste MSL-specificatie.

6.2.3 Opbouwen objectstructuur

De MetaDB-objectrepresentatie bevat niet alle relevante informatie die nodig is voor codegeneratie. Om bijvoorbeeld een formule makkelijk te kunnen vertalen naar broncode moet deze eerst worden omgezet naar een expressieboom. In de MetaDB-objectrepresentatie zijn formules slechts als string beschikbaar. In deze stap wordt een nieuwe objectrepresentatie gecreëerd (opgebouwd uit instanties van CodeGen-klassen). Hierin wordt voor elk MetaDB-object een (soort van) decorator-object aangemaakt dat verwijst naar het betreffende MetaDB-object en dat nieuwe functionaliteit en informatie toevoegt die noodzakelijk zijn voor het codegenereren. Figuur 6-1 toont deze relatie tussen de MetaDB- en CodeGen-klassen.

De CodeGen-klassen bezitten geen functionaliteit om over de objectstructuur te lopen; het zijn feitelijk veredelde datacontainers. Net als bij de MetaDB-klassen is met behulp van visitors het lopen over de objectstructuur losgekoppeld van de structuur zelf. Op deze manier kunnen zaken als semantische analyse, optimalisatie en codegeneratie in aparte klassen (visitors) worden ondergebracht en als het ware als plugin worden toegepast op het codegeneratieproces.

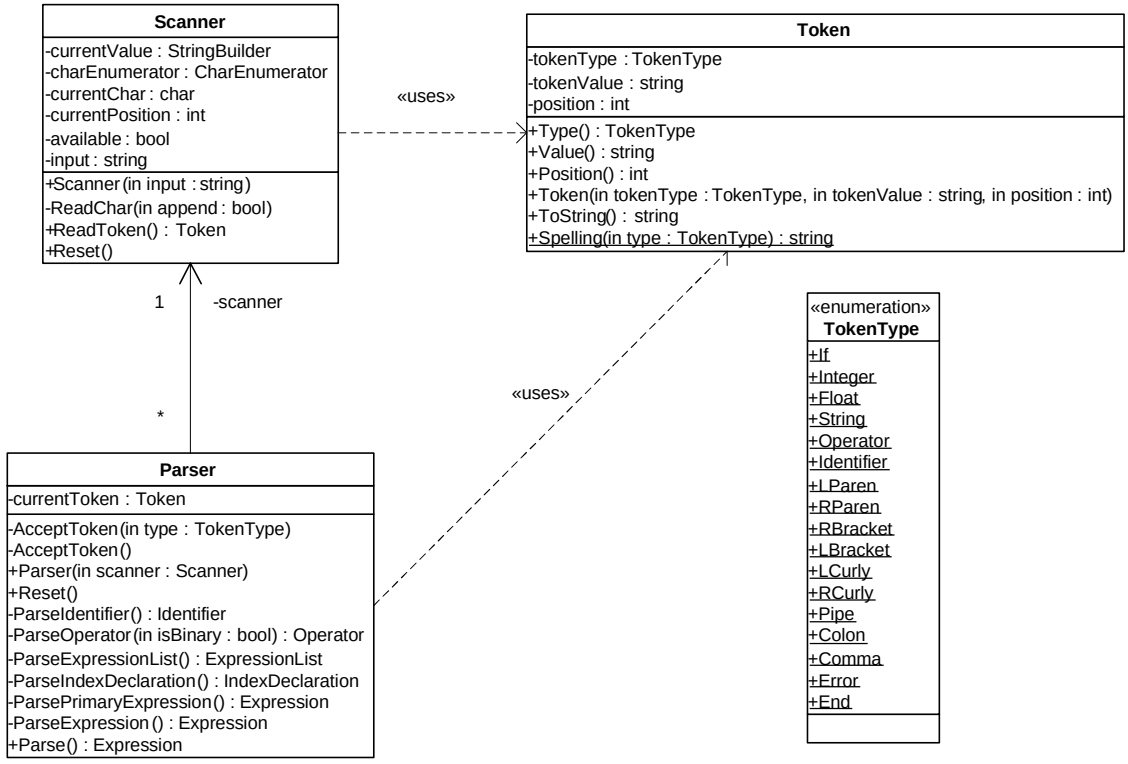


Figuur 6-1 Relatie CodeGen (links) - MetaDB (rechts) klassen

Met behulp van een visitor wordt over de MetaDB-objectstructuur gelopen en wordt voor elk MetaDB-object een corresponderend CodeGen-object aangemaakt. Op het afhandelen van *FormulaProcess*-objecten na is dit proces vrij eenvoudig. De bij een *FormulaProcess*-object horende formulestring (in MEL ASCII-vorm) wordt met behulp van parsingtechnieken omgezet naar een expressieboom.

Figuur 6-2 toont de klassen die verantwoordelijk zijn voor het vertalen van een formulestring naar een expressieboom. De *Scanner*-klasse voert lexicale analyse op de invoer (formulestring) uit. Tijdens de lexicale analysefase worden individuele tekens uit de invoer gegroepeerd en geklassificeerd tot tokens. Zo worden opeenvolgende cijfers als getal herkend en opeenvolgende letters als identifier. De gegenereerde tokens vormen de invoer van de *Parser*-klasse. De Parser-klasse implementeert een *recursive descent* parser waarmee syntactische analyse van MEL-expressies plaatsvindt.

Zowel parser als scanner zijn handmatig gecodeerd; er is geen gebruik gemaakt van lexer- en parsergeneratoren als bijvoorbeeld Lex en Yacc. Hiervoor is gekozen om niet afhankelijk te hoeven zijn van third-party tools. Daarnaast is de grammatica van MEL zodanig opgezet dat deze eenvoudig te parsen is (zie appendix A); het gebruik van dergelijke tools zou de implementatie van de codegenerator alleen maar complexer maken.



Figuur 6-2 Klassen verantwoordelijk voor vertaling naar expressieboom

6.2.4 Targetspecifieke declaraties

De voor een target ondersteunde taalonderdelen zijn door middel van declaraties configureerbaar binnen de Murray codegenerator. Zo is bijvoorbeeld in te stellen welke functies voor een bepaald target worden ondersteund en wat hiervan de signatuur is. De volgende onderdelen zijn door middel van declaraties in te stellen: binaire operatoren, unaire operatoren, aggregatoren, functies en constanten. In deze stap worden de geconfigureerde declaraties toegevoegd aan de symbol table die wordt gebruikt bij het uitvoeren van een semantisch analyse van de objectstructuur.

6.2.5 Semantische analyse

In deze stap wordt de tijdens de vorige stap opgebouwde objectstructuur gecontroleerd op semantische correctheid. Hierbij wordt de objectstructuur met behulp van een visitor recursief doorlopen. Tijdens het doorlopen wordt van elke expressie het type bepaald, gecontroleerd en in de objectstructuur opgeslagen. Gelijktijdig worden expressies onderzocht op semantische correctheid. Tijdens de analyse wordt intensief gebruik gemaakt van een symbol table.

De parser uit de vorige stap zal bijvoorbeeld een expressie in de vorm van *naam(parameter1, parameter2)* herkennen als een functieaanroep met twee parameters. Of er ook daadwerkelijk een dergelijke functie bestaat wordt pas tijdens de semantische analyse gecontroleerd. Hierbij wordt de voor de betreffende functienaam aanwezige verzameling van functiedeclaraties bij de symbol table opgevraagd[3]. In deze verzameling wordt gezocht naar een declaratie met het juiste

[3] Omdat MEL functieoverloading ondersteunt kunnen er meerdere functies met dezelfde naam (maar wel met een andere signatuur) bestaan

aantal parameters van het juiste type. Wanneer er geen juiste declaratie gevonden is, wordt er met behulp van type promotion getracht om toch nog een match te krijgen. Overal waar bijvoorbeeld een float wordt verwacht mag ook een integer worden gebruikt. Het niet vinden van een declaratie resulteert in een fout(melding). De uiteindelijk gevonden declaratie wordt aan de identifier (de naam van de functie in de functie-aanroep) gekoppeld en het type van de functie wordt ingevuld als het resultaattype van de betreffende functie-aanroep.

Tijdens de semantische analyse worden modelspecifieke declaraties (variabelen, indices, enumeraties en indexdeclaraties) op het juiste moment toegevoegd aan en verwijderd uit de symbol table. Zo worden alle variabelen die via een ingaande flow naar een formuleproces stromen toegevoegd aan de symbol table, vlak voordat de bij het formuleproces behorende formule-expressie op semantiek wordt gecontroleerd. Variabelen die niet als input dienen voor een formuleproces worden in diens formule-expressie dus ook niet herkend. Het gebruik hiervan levert dan ook een fout(melding) op.

6.2.6 Bepalen procesvolgorde

Als laatste stap voert het frontend het in paragraaf 4.4 beschreven algoritme uit op de aangemaakte objectrepresentatie. Het resultaat is een geannoteerde objectrepresentatie waarbij per proces (CodeProcess) is opgeslagen over welke indices dit proces itereert. Ook de correcte volgorde van de indices is daarbij opgeslagen. Daarnaast is de door het algoritme bepaalde volgorde van processen opgeslagen in het top-level CodeModel-object. Met behulp van deze informatie is het mogelijk code te genereren waarin de processen en eventuele iteraties in de door het algoritme bepaalde volgorde worden aangeroepen. De verschillende backends zijn zelf verantwoordelijk voor het uitlezen en uitgenereren van deze volgorde.

6.3 Backend

Een Murray codegeneratorbackend bestaat uit een enkele klasse die in ieder geval de IMurrayCodeGenerator interface implementeert. Met behulp van deze interface kan de codegenerator het backend correct aanspreken en code laten genereren. De twee geimplementeerde backends hebben een gezamelijke base-klasse (MurrayCodeGeneratorBase). Deze base-klasse implementeert de genoemde interface en bevat bovendien functionaliteit die beide backends nodig hebben.

Ook bij het codegenereren wordt de objectstructuur met behulp van een visitor afgelopen. Dit aflopen gebeurt voor elk backend op dezelfde manier. De uit te voeren actie per type object is uiteraard per backend verschillend. In de volgende paragrafen wordt hier dieper op ingegaan. Paragraaf 6.4 beschrijft het .NET-backend. In paragraaf 6.5 wordt het T-SQL-backend beschreven.

6.4 .NET codegeneratie

6.4.1 Inleiding

Het .NET backend van de Murray codegenerator kan broncode produceren voor elke door het .NET Framework ondersteunde taal. Dit is mogelijk doordat de codegenerator gebruik maakt van CodeDOM, een taalonafhankelijke representatie van broncode. CodeDOM staat voor Code Document Object Model en is onderdeel van het .NET Framework [13]. Het .NET Framework bevat standaard vertalers van CodeDOM naar C#, VisualBasic.NET, J# en C++[4]. Daarnaast bestaat de mogelijkheid om eenvoudig eigen vertalers te bouwen.

Het .NET codegenerator backend genereert broncode in twee stappen. Allereerst wordt de uit het frontend verkregen objectrepresentatie vertaald naar een CodeDOM-objectboom. Vervolgens wordt deze objectboom met een CodeDOM-vertaler omgezet naar de door de gebruiker gewenste .NET-taal.

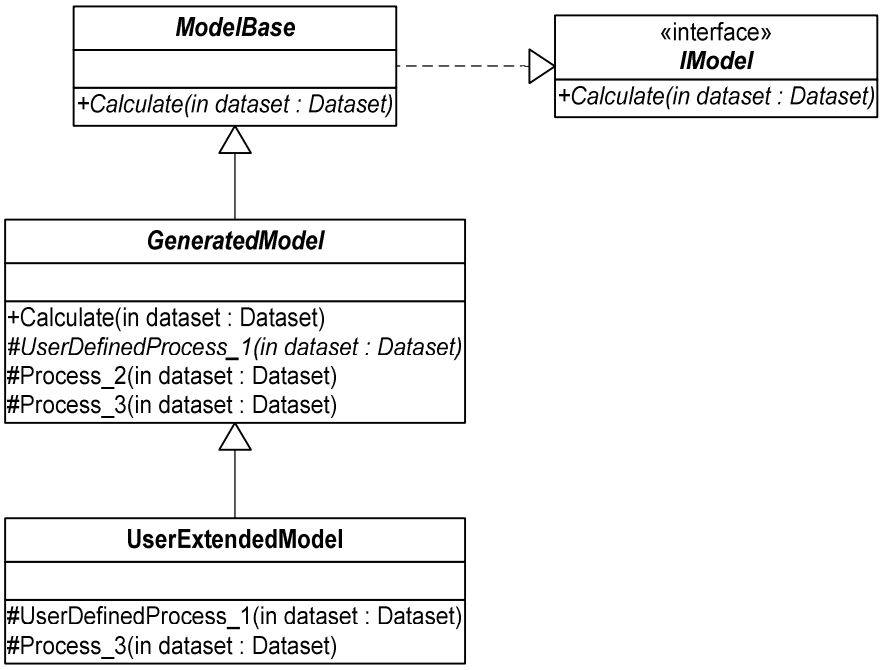
Het gebruik van CodeDOM heeft als groot voordeel dat er niet voor elke .NET-taal apart een backend hoeft te worden geschreven. Het gebruik van CodeDOM kent echter ook nadelen. Zo beschrijft CodeDOM taalconstructies op een abstracter niveau. In CodeDOM wordt een loop gerepresenteerd door een *CodeIterationStatement*. Het is niet mogelijk om de CodeDOM-objectboom te annoteren met aanwijzingen over hoe dit naar de target-taal moet worden vertaald, bijvoorbeeld naar een for-lus of een while-lus. Deze beslissing ligt bij de gekozen CodeDOM-vertaler. De C#-vertaler bijvoorbeeld produceert een for-lus met een lokaal aan de for-lus gedeclareerde loopvariabele, terwijl de VB.NET-vertaler een while-lus genereert met de loopvariabele *buiten* de loop gedeclareerd. Omdat de Murray codegenerator niet weet welke .NET-taal er wordt gegenereerd moet deze defensief te werk gaan. Vanwege mogelijke verschillen in scope-regelgeving zal het bijvoorbeeld niet altijd mogelijk zijn om de naamgeving van modelentiteiten in de gegenereerde code aan te houden.
Vanwege het hogere abstractieniveau zal in de uiteindelijk gegenereerde code slechts een subset van de taalconstructies van de doeltaal worden gebruikt. Dit levert meer code op, die bovendien minder goed leesbaar is vergeleken met code afkomstig van een codegenerator die specifiek voor deze doeltaal is ontwikkeld.

Toch is de gegenereerde code bruikbaar en leesbaar genoeg om te kiezen voor een dergelijke tussenstap in het vertalingsproces.

6.4.2 Opbouw van de gegenereerde code

Figuur 6-3 geeft met behulp van een voorbeeld de structuur van de gegenereerde .NET code weer. Voor het model wordt een klasse gegenereerd die overerft van de klasse *ModelBase* (*GeneratedModel* in het voorbeeld). De *ModelBase* klasse implementeert de *IModel* interface zodat modelimplementaties uniform aangesproken kunnen worden en biedt daarnaast basisfunctionaliteit aan.

[4] Het .NET Framework versie 2.0 bevat een C++-vertaler



Figuur 6-3 Structuur gegenereerde .NET code

Voor alle formule- en userdefined-processen in het model worden methoden gegenereerd: voor ieder formuleproces een methode waarbinnen de bijbehorende uitvoervariabele wordt uitgerekend en voor ieder userdefined-proces een abstracte methode die in een subklasse van de gegenereerde klasse moet worden geïmplementeerd. De voor het model gegenereerde klasse is abstract als het model minimaal één userdefined-proces bevat. De naam van een gegenereerde methode komt overeen met de naam van het betreffende proces in de MetaDB.

Gebruikers hebben de mogelijkheid om gegenereerde code te vervangen door eigen code door middel van het subklassen van de gegenereerde klasse. Alle gegenereerde methoden kunnen worden vervangen, omdat deze als virtual (non-final) zijn gedefinieerd. De met userdefined-processen corresponderende abstracte methoden *moeten* worden vervangen/geïmplementeerd. De *UserExtendedModel* klasse in het voorbeeld implementeert de abstracte methode *UserDefinedProcess_1* en vervangt de methode *Process_3* uit zijn superklasse.

Ook wordt er een *Calculate* methode gegenereerd waarbinnen eerder genoemde procesmethoden in de eerder bepaalde juiste volgorde worden aangeroepen. Met behulp van deze methode kan de gebruiker van de gegenereerde klasse de modelberekening starten. Bij de aanroep van de methode moet een Dataset-object als parameter worden doorgegeven. Met behulp van dit object kunnen modelgegevens (variabelen en indices) worden opgehaald en worden weggeschreven (zie paragraaf 5.4). Het object wordt vervolgens als parameter doorgegeven aan elke procesmethode, samen met eventuele indexwaarden-parameters (in het geval er sprake is van een iteratie). Figuur 6-4 toont een deel van de *Calculate* methode van het demografisch prognosemodel.

```
public virtual void Calculate( DataSet dataSet )
{
    DataIndex Year = dataSet.GetIndex( “Year” );

    for (int Year_value = Year.Min;
         Year_value <= Year.Max; Year_value++)
    {
          // Population
          DemografischModel_1_3_1( dataSet, Year_value );
          // NatPopulation
          DemografischModel_1_4_1( dataSet, Year_value );
          // MigrFacIm
          DemografischModel_1_4_2_2( dataSet, Year_value );
          // RegImmigrants
          DemografischModel_1_4_4( dataSet, Year_value );
          // MigrFacEm
          DemografischModel_1_4_2_1( dataSet, Year_value );
          // RegEmigrants
          DemografischModel_1_4_3( dataSet, Year_value );
          // Death
          DemografischModel_1_1_1( dataSet, Year_value );
          // Birth
          DemografischModel_1_2_1( dataSet, Year_value );
    }
    // PopSelection2020
    DemografischModel_1_3_4_3_3( dataSet );
    ...
}
```

Figuur 6-4 Voorbeeld van gegenereerde Calculate methode

Binnen een formuleprocesmethode worden de voor de berekening benodigde variabelen en indices bij het doorgegeven DataSet-object opgevraagd. Vervolgens wordt de bij het formuleproces behorende formule-expressie uitgerekend, voor alle mogelijke combinaties van indexwaarden. Dit laatste gebeurt met behulp van geneste *for*-lussen. Figuur 6-5 geeft een voorbeeld van code gegenereerd voor een simpel formuleproces uit het AOW-model, waarin twee variabelen van elkaar worden afgetrokken.

```
// Calculate AOW Deficiency
// formula: AOWDeficiency = (AOWDeposition[Year] - AOWPayment[Year])
protected virtual void AOWModel_1_4_2( DataSet dataSet )
{
    DataVariable AOWDeposition = dataSet.GetVariable(“AOWDeposition”);
    DataVariable AOWPayment = dataSet.GetVariable(“AOWPayment”);
    DataVariable AOWDeficiency = dataSet.GetVariable(“AOWDeficiency”);
    DataIndex Year = dataSet.GetIndex(“Year”);

    for (int Year_value = Year.Min; Year_value <= Year.Max;
         Year_value++)
    {
          AOWDeficiency[Year_value] =
                (double)AOWDeposition[Year_value] –
                (double)AOWPayment[Year_value];
    }
}
```

Figuur 6-5 Voorbeeld van gegenereerde formuleprocesmethode

6.4.3 Vertaling naar CodeDOM

Met behulp van een visitor wordt de objectstructuur recursief doorlopen en vertaald naar een CodeDOM-objectboom. In deze paragraaf wordt alleen het vertalen van MEL-expressies naar CodeDOM beschreven.

Voor elk type MEL-expressie heeft de *DotNetCodeGenerator*-klasse een visitmethode gedefinieerd. Elk van deze visitmethoden levert een *DotNetCodeBlock*-instantie op. De *DotNetCodeBlock*-klasse bestaat uit een CodeDOM-expressie (*CodeExpression*), een verzameling CodeDOM-statements (*CodeStatementCollection*) en een verzameling van in de expressie en statements gebruikte indices (*IndexCollection*).

MEL is een taal die alleen expressies kent: statements komen in de taal niet voor. Niet alle MEL-expressies zijn te vertalen naar alleen (een) CodeDOM-expressie(s). Soms zijn er statements nodig om het resultaat te kunnen berekenen. Een voorbeeld hiervan is de MEL conditie-expressie (zie paragraaf 4.3.10). Dit type expressie moet worden vertaald naar een aantal statements: de declaratie van een tijdelijke variabele en één of meer geneste if-then-else-statements waarin deze variabele de juiste waarde krijgt. De tijdelijke variabele kan in de uiteindelijke formule-expressie worden gebruikt, maar wel pas *nadat* de bijbehorende statements zijn uitgevoerd. Om deze reden heeft een *DotNetCodeBlock*-instantie dus niet alleen een CodeDOM-expressie maar ook een eventuele verzameling van CodeDOM-statements.

Figuur 6-6 toont hoe een *BinaryExpression*-object wordt vertaald naar CodeDOM. Allereerst worden de visitors van zowel de linker- als de rechterexpressie afgegaan om zo twee *DotNetCodeBlock*-objecten te verkrijgen. De statementverzamelingen van beide objecten worden toegevoegd aan de (lege) statementverzameling van het resultaat van de *VisitBinaryExpression* methode. De indexverzameling van het resultaat bestaat uit de vereniging van de twee verkregen indexverzamelingen. Verder wordt er als resultaatexpressie een *CodeBinaryOperatorExpression*-object aangemaakt met de naar CodeDOM vertaalde linker- en rechterexpressies als parameters. De MEL-operator wordt vertaald naar de corresponderende CodeDOM-operator met behulp van de *GetBinaryOperatorType* methode .

```
public override object VisitBinaryExpression(
    BinaryExpression expression, object argument)
{
    DotNetCodeBlock result = new DotNetCodeBlock();

    DotNetCodeBlock left = (DotNetCodeBlock)
        expression.LeftExpression.Visit( this, null );
    DotNetCodeBlock right = (DotNetCodeBlock)
        expression.RightExpression.Visit( this, null );

    result.Statements.AddRange( left.Statements );
    result.Indices.Merge( left.Indices );
    result.Statements.AddRange( right.Statements );
    result.Indices.Merge( right.Indices );

    result.Expression = new CodeBinaryOperatorExpression(
        left.Expression,
        GetBinaryOperatorType(expression.Operator),
        right.Expression);

    return result;
}
```

Figuur 6-6 Vertalen van een binaire expressie naar CodeDOM

6.4.4 Van CodeDOM naar broncode

In de tweede stap van het codegeneratieproces wordt de ontstane CodeDOM-objectboom omgezet naar de door de gebruiker gewenste .NET-taal. Met behulp van een factory is een instantie van de gewenste vertalerklasse te krijgen die de vertaalslag uitvoert. Elke CodeDOM-vertaler implementeert de *ICodeGenerator* interface die functionaliteit biedt om een CodeDOM-objectboom naar broncode te vertalen. Voor een verdere beschrijving van CodeDOM-vertalers wordt verwezen naar [13].

6.5 T-SQL codegeneratie

6.5.1 Inleiding

Het T-SQL-backend van de Murray codegenerator genereert T-SQL statements voor Microsoft SQL Server, een database management system (DBMS). T-SQL is het SQL dialect dat in SQL Server wordt gebruikt en staat voor Transact-SQL.

De voor een model gegenereerde statements moeten worden uitgevoerd op een database met in- en uitvoergegevens van het model. De opbouw van de tabellen in deze database moeten voldoen aan de in paragraaf 5.4.5 gestelde eisen. Na uitvoering van de statements bevat de database een aantal stored procedures[5] waarin de modelberekeningen plaatsvinden. Door deze stored procedures aan te roepen kan de gebruiker (delen van) het model berekenen. Berekeningen vinden op deze manier direct in de database plaats; er vindt geen dataoverdracht plaats tussen database en rekenmodelimplementatie.

6.5.2 Opbouw van de gegenereerde code

Voor elk formuleproces wordt een stored procedure gegenereerd waarin de bij het proces horende uitvoervariabele wordt uitgerekend. Voor het model zelf wordt één stored procedure gegenereerd, waarin de stored procedures van de formuleprocessen in de juiste volgorde worden aangeroepen. Hierbij wordt ook rekening gehouden met iteraties; de iteratie-indexwaarden worden aan de stored procedures als parameter meegegeven. Figuur 6-7 toont een deel van de voor het demografisch prognosemodel gegenereerde stored procedure. Hierin worden enkele processen geïtereerd over de jaarindex. De minimum- en maximumwaarden van deze index worden uit de jaarindextabel opgehaald.
Voor userdefined-processen wordt alleen de aanroep in de stored procedure van het model gegenereerd. De gebruiker dient zelf de betreffende stored procedure aan te maken.

[5] Een stored procedure bestaat uit een in SQL Server opgeslagen reeks (eventueel geparameteriseerde) T-SQL statements. Het gebruik van stored procedures levert een behoorlijke snelheidswinst op; statements hoeven niet telkens opnieuw vanuit een applicatie naar de database-server te worden gestuurd en daar te worden gecompileerd. Dit compileren gebeurt eenmalig bij het aanmaken van de stored procedure.

```
CREATE PROCEDURE dbo.usp_DemografischModel
AS
DECLARE @Year_value INT
DECLARE @Year_max_value INT
SET @Year_value = (SELECT MIN(Year) FROM Index_Y)
SET @Year_max_value = (SELECT MAX(Year) FROM Index_Y)
WHILE @Year_value <= @Year_max_value
BEGIN
    EXEC usp_DemografischModel_1_3_1 @Year_value -- Population
    EXEC usp_DemografischModel_1_4_1 @Year_value -- NatPopulation
    EXEC usp_DemografischModel_1_4_2_2 @Year_value -- MigrFacIm
    EXEC usp_DemografischModel_1_4_4 @Year_value -- RegImmigrants
    EXEC usp_DemografischModel_1_4_2_1 @Year_value -- MigrFacEm
    EXEC usp_DemografischModel_1_4_3 @Year_value -- RegEmigrants
    EXEC usp_DemografischModel_1_1_1 @Year_value -- Death
    EXEC usp_DemografischModel_1_2_1 @Year_value -- Birth
    SET @Year_value = @Year_value + 1
END
EXEC usp_DemografischModel_1_3_4_3_3 -- PopSelection2020
...
GO
```

Figuur 6-7 Voorbeeld van "model" stored procedure

Een voor een formuleproces gegenereerde stored procedure doorloopt de volgende stappen:

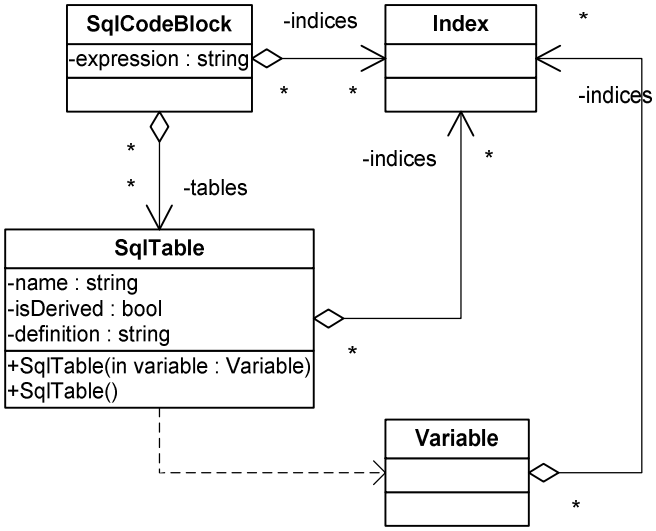
1. het aanmaken van een tijdelijke tabel om berekeningen in op te slaan. Deze tabel is op dezelfde manier opgebouwd als de ‘vaste’ tabellen binnen de DataDB (per index een veld, per variabele een veld, zie 5.4.5) met het verschil dat er in deze tijdelijke tabel de waarden van slechts één variabele komen te staan (de uitvoervariabele);
2. het vullen van deze tijdelijke tabel door middel van een query die de waarde(n) van de uitvoervariabele berekent, voor elke combinatie van diens indexwaarden;
3. het bijwerken van de bij de uitvoervariabele horende tabel met de nieuwe waarden uit de tijdelijke tabel (met behulp van insert en update statements);
4. het verwijderen van de tijdelijke tabel uit de database.

In de volgende paragraaf wordt het genereren van de query waarmee de waarde(n) van de uitvoervariabele zijn te berekenen beschreven.

6.5.3 Querygeneratie

Een query haalt data op uit tabellen. Bij dit ophalen worden de data mogelijk geaggregeerd, gefilterd en / of bewerkt. Het resultaat van een query is wederom een tabel; deze kan daarom door een nieuwe query worden gebruikt als brontabel. Het is dus mogelijk om queries te nesten. De T-SQL codegenerator maakt dankbaar gebruik van dit principe.

In figuur 6-8 zijn de voor de querygeneratie gebruikte hulpklassen en hun onderlinge relaties afgebeeld. Per klasse zijn de relevante methoden en velden aangegeven. De getter-/setter-methoden die de private fields beschikbaar maken zijn niet afgebeeld.



Figuur 6-8 hulpklassen SQL-codegeneratie

Een visitor loopt de bij een formuleproces horende expressieboom recursief af. Ieder bezoek aan een object in de expressieboom levert een *SqlCodeBlock*-instantie op, bestaande uit:

- een (deel van een) SQL-expressie (*expression*);
- een verzameling van de in deze SQL-expressie gebruikte tabellen (*SqlTable**);
- een verzameling van de indices gebruikt in zowel deze SQL-expressie als in de daarin gebruikte tabellen (*Index**).

De vulling van de teruggegeven *SQLCodeBlock*-instantie verschilt per type object. Zo levert een *IntegerExpression*-object het bijbehorende getal op als SQL-expressie. De twee verzamelingen zijn in dit geval leeg.
Bij het bezoeken van een *BinaryExpression*-object worden eerst diens linker en rechter expressie-objecten bezocht. De resulterende SQL-expressie bestaat uit de linker en rechter SQL-expressies, gescheiden door de juiste operator. De resulterende verzameling tabellen bestaat uit de vereniging van de linker en rechter tabelverzamelingen. Hetzelfde geldt voor de verzameling indices.

Een in de query gebruikte tabel is ofwel een vaste tabel uit de DataDB ofwel een virtuele tabel (geneste query). De *SqlTable*-klasse kan beiden representeren. Aan de hand van een *Variable*-instantie kan de constructor van de klasse de naam en indices van diens DataDB-tabel afleiden. Het gebruik van de constructor zonder parameters heeft een virtuele tabel tot gevolg. Een virtuele tabel heeft een unieke, gegenereerde naam (*_derived_<volgnummer>*). *AggregatorExpression*-objecten en *VariableExpression*-objecten waarbij op bepaalde indices wordt gefilterd, leveren virtuele tabellen op. Een virtuele tabel heeft naast eventuele indexvelden een veld met unieke naam (*_result_<volgnummer>*) waarin het resultaat van de (tussen)berekening staat. De query (*definition*) wordt in de Visitmethoden van de objecten gegenereerd.

Met de informatie verkregen uit een *SqlCodeBlock*-instantie kan een query worden opgebouwd. De opbouw van zowel de uiteindelijke formulequery als eventueel daardoor gebruikte geneste queries is bijna identiek. Figuur 6-9 toont deze opbouw.

```
SELECT          <index 1 table>.<index 1 field>,
                ...
                <index n table>.<index n field>,
                <expressie>
FROM            <index 1 table>
                ...
CROSS JOIN      <index n table>
LEFT OUTER JOIN <table 1> ON <table 1 join conditions>
                ...
LEFT OUTER JOIN <table n> ON <table n join conditions>
WHERE           <filter condities>
GROUP BY        <index 1 table>.<index 1 field>,
                ...
                <index n table>.<index n field>
```

Figuur 6-9 Queryopbouw

- *SELECT gedeelte*

  In dit gedeelte worden alle gebruikte indices opgesomd, gevolgd door de opgebouwde expressie. Elke index heeft een eigen tabel.

- *FROM gedeelte*

  Hier worden alle gebruikte index- en andere (mogelijk virtuele) tabellen aan elkaar "gejoind". Indextabellen worden met behulp van een CROSS JOIN aan elkaar verbonden. Dit houdt in dat *alle* mogelijke indexwaardencombinaties worden gegenereerd. Vervolgens worden de overige gebruikte tabellen aan dit resultaat verbonden met behulp van LEFT OUTER JOIN operaties op basis van de indexvelden.

  Wanneer in een gebruikte tabel een bepaalde indexwaardencombinatie niet voorkomt, zal voor een variable uit die tabel door de LEFT OUTER JOIN operatie een NULL waarde worden gegenereerd. Een NULL waarde wordt in een gegenereerde expressie vervangen door de defaultwaarde van een variabele (met behulp van de ISNULL functie).

  Variabelen zonder indices worden vóór het uitvoeren van de query waarin ze worden gebruikt in een constante ingelezen. De tabel waarin zij zich bevinden wordt dus niet met behulp van een join in het FROM gedeelte opgenomen.

- *WHERE gedeelte*

  Hier worden eventuele indexfiltercondities opgenomen. Bij het itereren van een formuleproces over een bepaalde iteratie-index wordt op deze plaats gefilterd op de geldende waarde van de iteratie-index. Bij een aggregatie wordt er hier gefilterd op de filterexpressie van de deelindex waarover wordt geaggregeerd.

- *GROUP BY gedeelte*

  Dit gedeelte wordt alleen in virtuele tabellen gebruikt. Bij aggregaties moet worden gegroepeerd op de indexcombinaties.

6.5.4 Voorbeelden

1.2.1 Calculate working population

$$wp_{a,g,r,y} = pop_{a,g,r,y} * pp_{a,g} * (1 + t_y * pf_{a,g})$$

pf	**ProgFactWorkPop**
pop	**Population**
pp	**ParticipantPerc**
wp	**WorkingPopulation**
t	**YearIndex**

Figuur 6-10 Berekening "Working Population"

De formule in figuur 6-10 is afkomstig uit het AOW Model. Voor formuleproces 1.2.1 wordt een stored procedure gegenereerd waarin de variabele WorkingPopulation wordt uitgerekend.

Voordat de query kan worden uitgevoerd moeten eerst de defaultwaarden van de in de query gebruikte variabelen gedeclareerd en van een waarde voorzien worden. Deze waarden worden eenmalig uit de MetaDB gelezen en in de stored procedure geplaatst (Figuur 6-11).

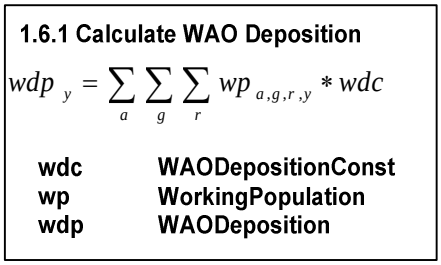
```
DECLARE @Population_Default FLOAT
SET @Population_Default = 0
DECLARE @ParticipantPerc_Default FLOAT
SET @ParticipantPerc_Default = 0
DECLARE @YearIndex_Default INT
SET @YearIndex_Default = 0
DECLARE @ProgFactWorkPop_Default FLOAT
SET @ProgFactWorkPop_Default = 0
```

Figuur 6-11 Declaratie defaultwaarden

Figuur 6-12 toont de voor de formule uit figuur 6-10 gegenereerde query.

```
SELECT      Index_a.Age AS Age,
            Index_g.Gender AS Gender,
            Index_r.Region AS Region,
            Index_y.Year AS Year,
((ISNULL(DemografischModel_a_g_r_y.Population,@Population_Default) *
ISNULL(Input_a_g.ParticipantPerc,@ParticipantPerc_Default)) *
(1 + (ISNULL(Input_y.YearIndex,@YearIndex_Default) *
ISNULL(AOWModel_a_g.ProgFactWorkPop,@ProgFactWorkPop_Default)))
) AS WorkingPopulation
FROM  Index_a
CROSS JOIN  Index_g
CROSS JOIN  Index_r
CROSS JOIN  Index_y
LEFT OUTER JOIN DemografischModel_a_g_r_y ON
            DemografischModel_a_g_r_y.Age = Index_a.Age AND
            DemografischModel_a_g_r_y.Gender = Index_g.Gender AND
            DemografischModel_a_g_r_y.Region = Index_r.Region AND
            DemografischModel_a_g_r_y.Year = Index_y.Year
LEFT OUTER JOIN Input_a_g ON
            Input_a_g.Age = Index_a.Age AND
            Input_a_g.Gender = Index_g.Gender
LEFT OUTER JOIN Input_y ON
            Input_y.Year = Index_y.Year
LEFT OUTER JOIN AOWModel_a_g ON
            AOWModel_a_g.Age = Index_a.Age AND
            AOWModel_a_g.Gender = Index_g.Gender
```

Figuur 6-12 Gegenereerde query voor berekening WorkingPoulation



Figuur 6-13 Voorbeeld van aggregatie

De sommatie over drie indices in de formule uit figuur 6-13 wordt in de gegenereerde code in drie stappen gedaan. Hoewel een dergelijke sommatie in SQL in één keer kan worden uitgevoerd, wordt op het moment door de SQL codegenerator per indexsommatie een virtuele tabel gegenereerd. Voor deze formule worden dus in totaal drie geneste virtuele tabellen geproduceerd. Figuur 6-14 toont de sommatie over de regionindex.

```
(
SELECT          Index_a.Age AS Age,
                Index_g.Gender AS Gender,
                Index_y.Year AS Year,
                SUM((ISNULL(AOWModel_a_g_r_y.WorkingPopulation,
                @WorkingPopulation_Default) *
                @WAODepositionConst)) AS _result_3
FROM            Index_a
CROSS JOIN      Index_g
CROSS JOIN      Index_r
CROSS JOIN      Index_y
LEFT OUTER JOIN AOWModel_a_g_r_y ON
                  AOWModel_a_g_r_y.Age = Index_a.Age AND
                  AOWModel_a_g_r_y.Gender = Index_g.Gender AND
                  AOWModel_a_g_r_y.Region = Index_r.Region AND
                  AOWModel_a_g_r_y.Year = Index_y.Year
GROUP BY        Index_a.Age,
                Index_g.Gender,
                Index_y.Year
) AS _derived_3
```

Figuur 6-14 Sommatie over regionindex

De gegenereerde virtuele tabel krijgt een unieke naam (hier _derived_3), evenals het veld met daarin de uitgerekende waarde (hier _result_3). De virtuele tabel kan vervolgens als “gewone” tabel in een verdere query worden gebruikt.

De in de formule gebruikte variabele *WAODepositionConst* heeft geen indices. Daarom wordt de waarde van deze variabele voor het uitvoeren van de query in een constante ingelezen. Deze constante wordt vervolgens in de query gebruikt. Figuur 6-15 toont hoe de waarde wordt ingelezen.

```
SET @WAODepositionConst =
      ISNULL((SELECT WAODepositionConst FROM Input),
      @WAODepositionConst_Default)
```

Figuur 6-15 Inlezen variabele zonder indices

7. Resultaten, conclusie en aanbevelingen

7.1 Resultaten

Het resultaat van dit afstudeerwerk is een prototype van het Murray framework, waarmee specificaties in MSL-formaat geautomatiseerd omgezet kunnen worden naar broncode geschreven in zowel T-SQL als een willekeurige .NET-taal. Met behulp van deze broncode en de runtime onderdelen van het Murray framework is het mogelijk om snel en efficiënt een rekenmodel te bouwen.

Het prototype kan de specificatie van het demografisch prognosemodel (zie appendix B) succesvol omzetten naar broncode. Dit model is representatief voor het soort modellen waar Murray voor is ontworpen. Bovendien is van deze specificatie een handgecodeerde referentie-implementatie aanwezig die gebruik maakt van de LionFish bibliotheken. De uitkomsten van de Murray en LionFish implementaties[6] zijn met elkaar vergeleken en identiek bevonden.

Het bepalen van een juiste procesvolgorde bleek lastiger dan in de eerste instantie gedacht. Er is een algoritme ontwikkeld waarmee een juiste volgorde van uitrekenen van processen binnen een model kan worden bepaald, rekening houdend met iteraties. De tijdcomplexiteit van het algoritme is exponentieel in het aantal verschillende iteratie-indices dat in het model wordt gebruikt. Dit vormt geen obstakel aangezien het aantal indices waarover in een model wordt geïtereerd normaliter beperkt blijft.

De performance van de gegenereerde .NET code is vergelijkbaar met die van de referentie-implementatie. Een exacte vergelijking is lastig omdat de LionFish-server zijn gegevens pas na enige inactiviteit in de achtergrond naar de database wegschrijft.

Het wegschrijven van de uitkomsten van een rekenmodel naar een database kost erg veel tijd in vergelijking met het inlezen van de benodigde gegevens en het uitrekenen van het model. In het geval van het demografisch prognosemodel is er 2% van de tijd nodig voor het inlezen, 2% voor het uitrekenen en 96% voor het wegschrijven. Hierbij is het wegschrijven al enigszins geoptimaliseerd door gebruik te maken van een XML-structuur (zie 5.4.4).

De performance en bruikbaarheid van de gegenereerde T-SQL code is duidelijk minder dan die van de gegenereerde .NET code. In het geval van het demografisch prognosemodel is de .NET-implementatie tweemaal zo snel. SQL Server en zijn T-SQL taal zijn meer geschikt voor het verwerken van grote hoeveelheden data dan voor het uitvoeren van ingewikkelde berekeningen. Wel kan een combinatie van .NET code en T-SQL code een snelheidsverbetering opleveren. Hierbij kan worden gedacht aan het uitvoeren van pre- en postprocessing van het model in SQL Server terwijl het echte rekenwerk in .NET code wordt gedaan.

[6] Omdat de manier van uitrekenen in de T-SQL-implementatie nogal verschilt met die van de .NET- en referentie-implementaties, zijn er minimale afrondingsverschillen in uitkomsten waarneembaar tussen de eerstgenoemde implementatie en de andere implementaties.

Het gebouwde prototype voldoet aan de functionele eisen die in paragraaf 3.3 zijn gesteld, met uitzondering van de volgende (sub)eisen:

FE-4 *optimaliseren modelspecificatie (lage prioriteit)*

Omdat het wegschrijven van de uitkomsten van een rekenmodel naar een database *veel* meer tijd kost dan het uitrekenen van het rekenmodel is er voor gekozen om dit onderdeel niet uit te voeren. In plaats hiervan is er gewerkt aan het zo optimaal mogelijk kunnen wegschrijven van modeluitkomsten naar een database.

FE-5.8 *genereren stubcode*

De .NET codegenerator voldoet aan de gestelde eis. De T-SQL codegenerator genereert geen stubcode voor userdefined-processen, omdat deze stubcode de implementatie van de gebruiker zou kunnen overschrijven. Zie ook *FE-5.14*.

FE-5.12 *debug-uitvoer*

De gegenereerde code geeft slechts debug-informatie over de gebruikte rekentijd. Deze informatie is niet met levels te filteren.

FE-5.14 *wijzigingen behouden na opnieuw codegenereren*

De .NET codegenerator voldoet aan de gestelde eis wanneer de gebruiker wijzigingen doorvoert door middel van het *overriden* van een process-methode in een subklasse van de gegenereerde modelklasse.
De T-SQL codegenerator voldoet niet aan deze eis; aanpassingen aan gegenereerde stored procedures gaan verloren bij het opnieuw genereren.

FE-5.15 *mogelijkheid tot starten, pauzeren en stoppen van berekeningen (lage prioriteit)*

Aan deze eis is niet voldaan wegens tijdgebrek.

FE-5.16 *tonen van voortgang van de berekeningen (lage prioriteit)*

Aan deze eis is niet voldaan wegens tijdgebrek.

7.2 Conclusie

Met behulp van het gerealiseerde prototype kunnen rekenmodellen aanzienlijk sneller gebouwd worden dan voorheen. Daarnaast zal de geproduceerde rekenmodelimplementatie naar alle waarschijnlijkheid minder fouten bevatten dan wanneer deze met de hand zou zijn gemaakt, zoals ook al is beschreven in paragraaf 2.2.1.

De vraag of het mogelijk is om met behulp van automatisering het huidige proces van het ontwikkelen van rekenmodellen te versnellen en de foutgevoeligheid van de ontwikkeling van rekenmodellen te verminderen, moet in het licht van het hierboven beschrevene dus bevestigend worden beantwoord.

7.3 Aanbevelingen

Diverse ideeën voor verbetering en uitbreiding die buiten de scope van deze afstudeeropdracht vielen zijn hieronder kort beschreven als aanbevelingen voor doorontwikkeling.

7.3.1 Synchronisatie modelspecificatie met MetaDB

Op dit moment worden Murray specificaties gemaakt met behulp van Microsoft Visio. Vervolgens worden deze specificaties met de hand vertaald naar MSL/MEL en in de Murray MetaDB gezet. Deze vertaalslag kost veel tijd en is bovendien foutgevoelig. Een aanpassing aan de specificatie heeft (uiteraard) tot gevolg dat ook de MetaDB aangepast moet worden. Wanneer dit niet (of niet goed) gebeurt, ontstaan er verschillen. Het grote voordeel van codegeneratie wordt zo enigszins teniet gedaan.

De mogelijkheid van het koppelen van de MetaDB aan een grafische designer, zodat de specificatie slechts op één plek hoeft te worden bijgehouden, lijkt een interessante uitbreiding voor het Murray framework. Hierbij kan bijvoorbeeld worden gedacht aan een plugin voor Microsoft Visio of aan een eigen implementatie van een dergelijke designer met behulp van de Domain-Specific Language Tools[7] van Microsoft.

7.3.2 Uitbreidingen MSL

In deze paragraaf worden enkele mogelijke uitbreidingen van MSL besproken. Met behulp hiervan wordt MSL een nog krachtigere rekenmodelspecificatietaal, zodat nog meer (soorten) rekenmodellen in MSL kunnen worden geschreven en zodoende met behulp van het Murray framework worden doorgerekend.

Iteraties met stopcriterium

Op dit moment worden binnen MSL alleen *for-iteraties* ondersteund. Dit houdt in dat er wordt geïtereerd over één of meerdere indices, in een *vast* aantal iteratiestappen (namelijk het aantal verschillende combinaties van indexwaarden).

MSL ondersteunt geen *while-iteraties*. Hierbij is het aantal iteratiestappen *variabel*; er wordt geïtereerd zolang er niet aan een bepaald stopcriterium is voldaan. Hiermee kunnen berekeningen worden uitgevoerd die in een aantal iteratiestappen convergeren naar de gewenste waarde (door steeds een betere benadering te geven).

Een variabele krijgt zijn waarden slechts éénmaal toegekend (zie paragraaf 4.2.4). Om het hierboven genoemde convergeren te kunnen ondersteunen moeten de betrokken variabelen dus worden voorzien van een extra iteratie-index. Voor elke iteratiestap worden de waarden van de

[7] Met deze tools is het mogelijk om een custom grafische designer te creëren die gebruik maakt van zelf gedefinieerde domein-speficieke diagramnotatie. De met DSL-tools gemaakte grafische designer opereert binnen Microsoft Visual Studio en genereert XML-bestanden die met behulp van templates te vertalen zijn naar bijvoorbeeld broncode. Op het moment van schrijven is er van de DSL-software nog slechts een betaversie beschikbaar. Zie ook [14].

variabele bewaard. Dit kan geheugenproblemen veroorzaken bij variabelen met veel indices en/of veel iteratie-stappen. Daarnaast hebben indices op dit moment een vast aantal indexwaarden, dus zal er automatisch een maximum aantal iteratiestappen gelden. Verder moet er na het itereren ook bekend zijn bij welke iteratiestap de iteratie is beëindigd, om zo de juiste waarden van de variabelen te kunnen verkrijgen.

Een alternatief zou zijn om in geval van dit type iteraties *wel* de mogelijkheid te bieden om aan variabelen meerdere malen een waarde toe te kennen. De iteratieindex staat dan los van de variabele.

Dynamische indices

Het bereik van een index is gedefinieerd in de MetaDB; in een aangeleverde dataset moet voor het gehele indexbereik data aanwezig zijn. Als de gebruiker een kleinere dataset wil doorrekenen met bijvoorbeeld de gegevens van een beperkt aantal jaren, dan moet hij het bereik van de jaarindex in de MetaDB aanpassen. Doet hij dat niet, dan rekent het systeem voor de missende indexwaarden (jaren) met defaultwaarden (resulterend in verkeerde uitkomsten) en duurt het doorrekenen van een dergelijke kleine dataset dus even lang als het doorrekenen van een volledige dataset.

Het zou een nuttige uitbreiding zijn om in een dataset te kunnen aangeven voor welk indexbereik data aanwezig zijn. Zo kunnen verschillende datasets (van verschillende grootte) eenvoudiger doorgerekend worden.

Afhankelijke indices

Indexbereiken zijn in MSL onderling niet afhankelijk, dat wil zeggen dat mogelijke indexwaarden van de ene index niet afhangen van de op dat moment gekozen waarden van andere indices. De data van een variabele zijn dus volledig multidimensionaal.

Stel, men wil voor iedere combinatie herkomst-bestemming het aantal routesegmenten vastleggen, en per segment bijvoorbeeld de kosten. In MSL zou men dan een variabele met drie indices aanmaken: een herkomst-, een bestemmings- en een routesegmentindex. Deze laatste index verschilt per combinatie van de eerste twee indices, aangezien niet elke “route” een zelfde aantal segmenten heeft. Daarom is het aantal elementen van deze index het maximum aantal routesegmenten dat voor kan komen. Wanneer de routelengtes erg van elkaar verschillen gaat dit gepaard met onnodig hoog geheugengebruik. Beter zou zijn om een manier te vinden om het aantal indexelementen afhankelijk te laten zijn van gekozen indexwaarden van andere indices. In dit geval wordt de volgorde van indices van belang: een index is afhankelijk van de *eerdere* indices.

Ondersteuning subtypes

Op dit moment kent MSL drie types: integer (32-bit), float (64-bit) en bool. Soms past het bereik van een variabele eenvoudig in een kleiner type integer (8- of 16-bit). Dit is bijvoorbeeld het geval bij variabelen met een enumeratie. Ook is het denkbaar dat formule-uitkomsten

minder precies hoeven te worden opgeslagen (32-bits floatgetallen in plaats van 64-bits floatgetallen).

Wanneer een variabele veel indices heeft kan het nemen van een kleiner type veel schelen in zowel geheugengebruik als benodigde opslagruimte.

Met behulp van type-promotion[8] kunnen formules zo precies mogelijk worden uitgerekend. Na afloop kan het resultaat worden geconverteerd naar het (mogelijk kleinere) gewenste type. Het is aan de gebruiker om ervoor te zorgen dat de formule-uitkomsten in het gekozen datatype passen.

7.3.3 Verbeteringen .NET codegeneratie

De op dit moment door Murray gegenereerde .NET code is redelijk efficiënt. De bewerkingen zijn vrij eenvoudig en daarom kunnen de compilers in het .NET Framework deze code goed optimaliseren. Toch zijn er verbeteringen en uitbreidingen mogelijk die de implementatie van rekenmodellen efficiënter maken. De in deze paragraaf voorgestelde verbeteringen hebben zowel betrekking op de .NET codegenerator als op het DataDB gedeelte van het Murray framework.

Statements buiten for lussen halen

Wanneer een expressie onderdeel uitmaakt van een aggregatie over een bepaalde index en de expressie zelf niet afhankelijk is van deze index, is het mogelijk om deze expressie buiten de aggregatie te halen, en dus buiten de voor de aggregatie gegenereerde for-lus. Dit houdt in dat de betreffende expressie slechts éénmaal wordt uitgevoerd in plaats van voor iedere indexwaarde één keer.

Simpele aggregaties over een variabele door de datalaag laten afhandelen

Aggregaties worden door de codegenerator omgezet naar een constructie met geneste for-lussen. Beperkende condities worden door middel van if-statements geïmplementeerd. In het geval van simpele aggregaties zou dit efficiënter in de datalaag uitgevoerd kunnen worden. Denk hierbij bijvoorbeeld aan het bepalen van totalen van een variabele, door middel van het wegsommeren van indices. De DataVariabele klasse heeft de waarden van een variabele opgeslagen in een platgeslagen array. Het optellen van (een deel van) de waarden in deze array is vele malen efficiënter dan het één voor één ophalen van waarden met behulp van een method-call. Niet alleen is een method-call op zich al een dure operatie, maar ook is er nog eens extra sprake van een box/unbox operatie van een valuetype[9].

[8] Het optellen van twee 8-bits waarden resulteert bijvoorbeeld in een 16-bits waarde.

[9] Inkapselen van een waarde in een object. Om de methoden op de DataVariabele klasse generiek te houden is het nodig dat deze waarden van het type object retourneert. Intern houdt de klasse wel een array bij van het originele valuetype. In .NET 2.0 is dit probleem overigens op te lossen door gebruik te maken van generics.

Geheugenoptimalisaties

In de DataDB-laag worden alle waarden van alle variabelen tijdens het uitrekenen van een model in het geheugen bewaard. Dit houdt in dat er voor elke mogelijke combinatie van indexwaarden voor een variabele geheugenruimte is gereserveerd. Dit kan problematisch worden wanneer er sprake is van veel indices omdat het aantal mogelijke combinaties van indexwaarden snel stijgt wanneer het aantal indices toeneemt.
Om grote datasets te kunnen doorrekenen zal er enige vorm van geheugenoptimalisatie toegepast moeten worden. Mogelijkheden zijn:

- niet-persistente variabelen[10] uit het geheugen verwijderen wanneer deze niet meer nodig zijn; dit kan verder worden geoptimaliseerd door wijzigingen aan te brengen in de volgorde van het uitrekenen van variabelen, zodat variabelen zo snel mogelijk weer uit het geheugen verwijderd kunnen worden. Uiteraard moet er wel rekening gehouden worden met de geldende afhankelijkheden tussen variabelen;
- alleen die variabelen in het geheugen houden die betrokken zijn bij de huidige berekening. Dit kan echter wel de rekenperformance nadelig beïnvloeden, aangezien dezelfde variabelen mogelijk meerdere malen in het geheugen ingelezen moeten worden;
- extreem grote variabelen per index of beperkte combinatie van indices in het geheugen houden en uitrekenen. Dit is alleen mogelijk wanneer de volgorde van de indices bij het uitrekenen irrelevant is;
- niet voor alle mogelijke combinaties van indexwaarden geheugenruimte reserveren, maar slechts voor gedefinieerde combinaties. Hiermee kunnen variabelen met veel indices maar met weinig vulling (*sparse matrices*) efficiënter worden verwerkt. Op dit moment wordt aan de hand van de indexwaarden de positie in het geheugen van de bijbehorende waarde bepaald. Deze positie zou in het geval van sparse matrices gebruikt kunnen worden als key in bijvoorbeeld een hashtable. Zo worden alleen nog maar de gebruikte waarden in het geheugen bewaard.

7.3.4 Verbeteringen SQL codegeneratie

Deze paragraaf beschrijft enkele verbeteringen die, wanneer doorgevoerd, de inzetbaarheid en performance van de gegenereerde SQL code zullen doen toenemen.

Aggregatie Optimalisaties

Een sommatie over *n* indices gebeurt in *n* slagen, terwijl dit in SQL eenvoudig in 1 slag kan (zie bijvoorbeeld figuur 6-13 in paragraaf 6.5.4). Er moet dan wel rekening worden gehouden met het feit dat indices (of hun restricties) van elkaar afhankelijk kunnen zijn.

Wanneer de driemaal geneste query die voor de formule uit figuur 6-13 is gegenereerd herschreven wordt, zodat deze de betreffende sommatie over drie indices tegelijk uitvoert, resulteert dit in een performance verbetering van 10% (volgens de “Estimation Execution

[10] Een niet-persistente variabele is een doorvoervariabele die niet naar de uitvoer wordt weggeschreven (een tussenresultaat).

Plan”). Omdat deze query met een beperkt aantal records werkt is het lastig om dit ook echt na te meten.

N.B. Omdat er gebruik gemaakt wordt van floating-point getallen is het mogelijk dat resultaten door optimalisaties wijzigen!

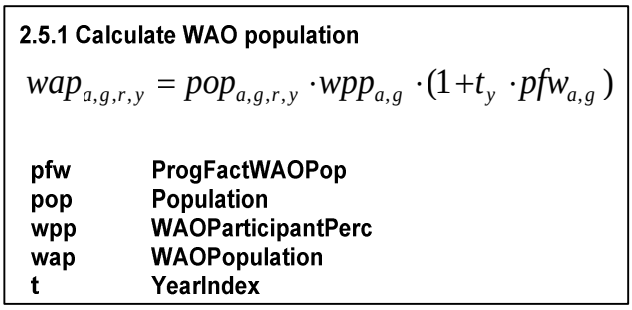
Gebruik van tijdelijke tabellen

Een gegenereerde query kan vrij complex worden door alle subqueries/derived tables. Wanneer een subquery niet afhankelijk is van de (hoofd) query kan het resultaat ervan in een tijdelijke tabel opgeslagen worden. Dit heeft een aantal voordelen:

- de query wordt minder complex;
- het resultaat van de subquery kan worden hergebruikt.

Combineren van Queries

De SQL codegenerator produceert voor elke berekening van een variabele een query. De formules uit figuur 6-10 en figuur 7-1 worden berekend aan de hand van data uit dezelfde tabellen. Bovendien komen de twee uitgerekende variabelen uiteindelijk ook in dezelfde tabel terecht.



Figuur 7-1 Berekening "WAO Population"

Het samenvoegen van de twee gegenereerde queries heeft de volgende positieve gevolgen op de performance:

- de benodigde data (indexcombinaties, waarden van gebruikte variabelen bij die indexcombinaties) hoeven slechts éénmaal te worden opgevraagd;
- de insert en updatequeries hoeven nog slechts éénmaal te worden uitgevoerd (in plaats van tweemaal);
- er hoeft nog maar één keer een tijdelijke tabel te worden aangemaakt en gevuld. Deze tabel is minder dan tweemaal zo groot als de twee ‘originele’ tabellen gecombineerd.

Berekeningen uitvoeren in code

SQL Server is er niet op ontworpen om ingewikkelde expressies snel te kunnen uitrekenen. SQL Server 2005 biedt de mogelijkheid om zogenaamde *User Defined Functions* (UDF) te schrijven in bijvoorbeeld C#. Een dergelijke functie is vervolgens weer in een SQL query te gebruiken. In de context van Murray lijkt het nuttig om dergelijke functies te kunnen genereren voor zowel ingewikkelde expressies als berekeningen die niet door SQL Server / Transact-SQL worden ondersteund.

Hieronder volgt een voorbeeld van een dergelijke UDF. De broncode in figuur 7-2 is een implementatie van de berekening uit figuur 6-10. Het gebruik van deze functie heeft tot gevolg dat de query een aantal malen langzamer is dan wanneer de formule direct in SQL wordt uitgevoerd.
Vanwege de overhead (function call, doorgeven parameters en omzetten van parameters van SQL datatype naar .NET datatype) zullen alleen complexe berekeningen een verbetering in performance tot gevolg hebben. Dit simpele voorbeeld is daarom ook niet representatief (maar zuiver illustratief).

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
{
    [Microsoft.SqlServer.Server.SqlFunction(
        IsDeterministic=true,IsPrecise=false)]
    public static SqlDouble WorkingPopulation(
        double population,
        double participantPerc,
        double yearIndex,
        double progFactWorkPop)
    {
        return population * participantPerc *
            (1 + yearIndex * progFactWorkPop);
    }
};
```

Figuur 7-2 Berekeningen uitvoeren in code in plaats van in SQL

Link met MetaDB

De SQL codegenerator gaat ervan uit dat de inhoud van de MetaDB na het code genereren niet meer wijzigt. De op het moment van genereren geldende defaultwaarden van variabelen worden bijvoorbeeld “hardcoded” in de SQL code geplaatst. Ook minima, maxima en counts van indices worden als getal in de code gezet in plaats van dynamisch uitgerekend.

Er moet een oplossing bedacht worden om deze getallen instelbaar te kunnen maken. Dit zou bijvoorbeeld kunnen door het uitlezen uit de MetaDB, doorgeven met behulp van parameters of uitlezen uit een extra tabel in de DataDB.

Ondersteuning meerdere databasesystemen

Momenteel wordt alleen SQL Server ondersteund door de SQL codegenerator. De code zou aangepast kunnen worden door de databasesysteem-afhankelijke stukken te verplaatsen naar abstracte methodes die door een concrete implementatie voor een bepaald databasesysteem geïmplementeerd moeten worden.

Verklarende Woordenlijst

Afkorting	Omschrijving
CLI	Command Line Interface
DSL	Domain Specific Language
EBNF	Extended Backus-Naur Form
MEL	Murray Expression Language
MRL	Murray Runtime Library
MSL	Murray Specification Language
SQL	Structured Query Language
SRS	Software Requirements Specification
T-SQL	Transact SQL, het SQL-dialect dat in Microsoft SQL Server wordt gebruikt.
UML	Unified Modeling Language
ERD	Entity Relationship Diagram

Bibliografie

[1] G.F. Zegwaard. Formele Specificatie Richtlijnen. Memo 282, QQQ Delft, 2 mei 2003

[2] Karl E. Wiegers. Software Requirements. Second Edition, Microsoft Press, 2003

[3] Microsoft Corporation. Microsoft Visual C#: Language Reference. Microsoft Press, 2002

[4] A. Aho, R. Sethi en J. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986

[5] E. Gamma, R. Helm, R. Johnson en J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[6] Microsoft Corporation. SQL Server 2000 Books Online. January 2004 Update. Microsoft Corporation, Redmond, WA, 2004

[7] G.F. Zegwaard. Richtlijnen Codering. Rapport 270, QQQ Delft, 14 juni 2005

[8] David A. Watt. Programming Language Syntax and Semantics. Prentice Hall International, 1991

[9] David A. Watt en Deryck F. Brown. Programming Language Processors in Java: Compilers and Interpreters. Prentice Hall, 2000

[10] Thomas H. Cormen et al. Introduction to Algorithms. Second Edition, MIT Press, 2001

[11] Esko Nuutila en Eljas Soisalon-Soininen. On Finding the Strongly Connected Components in a Directed Graph. Information Processing Letters, 49(1):9-14, 1994

[12] Esko Nuutila. An Efficient Transitive Closure Algorithm for Cyclic Digraphs. Information Processing Letters, 52(4):207-213, 1994

[13] Microsoft Corporation. Microsoft .NET Framework 1.1 Class Library Reference Volumes 1-7. Microsoft Press, 2003

[14] Microsoft Visual Studio Developer Center: Domain-Specific Language Tools. http://msdn.microsoft.com/vstudio/DSLTools/

A Murray Expression Language

Hieronder volgt de grammatica van MEL in EBNF. Eind symbolen zijn vet weergegeven.

Concrete Syntax (EBNF)

```
Formula ::= VariableExpression := Expression

Expression ::= PrimaryExpression
             | Expression Operator PrimaryExpression

PrimaryExpression ::= Integer-Literal
                    | Float-Literal
                    | Operator Primary-Expression
                    | ConditionalExpression
                    | AggregatorExpression
                    | CallExpression
                    | VariableExpression
                    | NamedExpression
                    | EnumerationExpression
                    | ( Expression )

ConditionalExpression ::= if ( ( Expression, Expression, )+ Expression )
AggregatorExpression ::= Identifier ( Index-Declaration (, Expression)? )
Index-Declaration ::= { Identifier ( | Identifier ( : Expression )? )? }
CallExpression ::= Identifier ( Expression-List )
VariableExpression ::= Identifier [ Expression-List ]
Expression-List ::= ( Expression ( , Expression )* )?
NamedExpression ::= Identifier
EnumerationExpression ::= Identifier : String-Literal
Identifier ::= Letter (Letter | Digit)*
Operator ::= + | - | * | / | = | != | % | < | > | <= | >= | and | or | not
Integer-Literal ::= Digit+
Float-Literal ::= Digit+ . Digit+
String-Literal ::= ‘ Character* ‘
Letter ::= a..z | A..Z
Digit ::= 0..9
Character ::= any-ascii-character
```

Ondersteunde operatoren

Onderstaande tabel bevat een opsomming van door MEL ondersteunde operatoren, geordend naar prioriteit (van hoog naar laag). De associativiteit van de operatoren geeft aan in welke volgorde operatoren met gelijke prioriteit worden uitgevoerd.

Operator (ASCII)	Operator (Grafisch)	Omschrijving	Associativiteit
-	−	Unaire Minus	Rechts naar links
not	¬	Logisch niet	
*	·	Vermenigvuldiging	Links naar rechts
/	−	Deling	
%	mod	Modulo	
+	+	Plus	Links naar rechts
-	-	Minus	
<	<	Kleiner dan	Links naar rechts
<=	≤	Kleiner of gelijk aan	
>	>	Groter dan	
>=	≥	Groter of gelijk aan	
=	=	Gelijkheid	Links naar rechts
!=	≠	Ongelijkheid	
and	∧	Logisch en	Links naar rechts
or	∨	Logisch of	

Onderstaande tabel geeft de in MEL ondersteunde operator signaturen. Wanneer een gevraagde signatuur niet is gedefinieerd zal met behulp van type promotion de juiste signatuur opgezocht worden. Een vermenigvuldiging van een integer type en een float type bijvoorbeeld zal als resultaat een float opleveren. De combinatie is niet in onderstaande tabel te vinden. Wanneer het integere type naar een float type wordt gepromoveerd is er wel een entry in onderstaande tabel gedefinieerd en is het resultaat type dus float.

Operator (ASCII)	Signatuur
- *(unair)*	$int \rightarrow int$
	$float \rightarrow float$
not	$bool \rightarrow bool$
*,%,+,-	$int \times int \rightarrow int$
	$float \times float \rightarrow float$
/	$int \times int \rightarrow float$
	$float \times float \rightarrow float$
<,<=,>,>=	$int \times int \rightarrow bool$
	$float \times float \rightarrow bool$
=,!=	$int \times int \rightarrow bool$
	$float \times float \rightarrow bool$
	$bool \times bool \rightarrow bool$
And, or	$bool \times bool \rightarrow bool$

Ondersteunde functies

Naam	Signatuur	Omschrijving
Min	$int \times int \rightarrow int$ $float \times float \rightarrow float$	Het minimum van twee getallen.
Max	$int \times int \rightarrow int$ $float \times float \rightarrow float$	Het maximum van twee getallen.
Abs	$int \rightarrow int$ $float \rightarrow float$	De absolute van een getal.
Floor	$float \rightarrow float$	Het grootste gehele getal dat kleiner of gelijk is aan het gegeven getal.
Ceiling	$float \rightarrow float$	Het kleinste gehele getal dat groter of gelijk is aan het gegeven getal.
Round	$float \rightarrow float$	Het gehele getal dat het dichtst bij het gegeven getal ligt.
Convert	$float \rightarrow int$	Het aan het gegeven float getal equivalente integere getal. Wanneer het gegeven getal geen geheel getal is wordt het getal afgerond naar het dichtstbijzijnde gehele getal.

Ondersteunde Named Constanten

MEL (ASCII)	MEL (Grafisch)	Type	Waarde	Omschrijving
True	true	Bool	True	Logisch Waar
False	false	Bool	False	Logisch Niet waar
Pi	π	Float	3.14…	(Benadering van) π
E	e	Float	2.71…	(Benadering van) e

B Demografisch prognosemodel

In deze bijlage wordt een beschrijving en een volledige specificatie gegeven van het demografisch prognosemodel. Deze modelspecificatie is als case gebruikt voor het Murray framework.

Het demografisch prognosemodel bestaat uit twee submodellen: een demografisch submodel dat de bevolkingsontwikkelingen voorspelt en een arbeidsmarkt-submodel dat de uitwerking van de bevolkingsontwikkelingen op de AOW en de WAO berekent.

Door het demografisch submodel wordt voor de periode van 2000 tot 2020 voorspeld hoe de bevolking zich ontwikkelt. Hiervoor worden geboorten, immigratie, emigratie, sterften en veroudering berekend. De parameters van dit submodel zijn beschikbaar gesteld door het CBS en komen overeen met die van de nationale bevolkingsprognose voor 1999. De demografische rekenregels zijn beschikbaar gesteld door de Rijksplanologische Dienst.

Ook voor het arbeidsmarkt-submodel, dat de consequenties van de bevolkingsontwikkeling voor de AOW en de WAO berekent, worden de gebruikte parameters beschikbaar gesteld door het CBS. In het arbeidsmarkt-submodel wordt berekend hoeveel AOW en WAO-premies worden ingelegd en hoeveel WAO-ers en AOW-ers aanspraak op een uitkering zullen maken.

De specificatie van het demografisch prognosemodel is om de volgende redenen als case gebruikt voor Murray:

1. deze specificatie bevat de meeste door Murray ondersteunde expressietypen, waardoor deze specificatie bijzonder geschikt is voor gebruik bij black-box testen;
2. van deze specificatie is een referentie-implementatie aanwezig. Deze referentie-implementatie maakt gebruik van de LionFish bibliotheken en is handmatig gecodeerd. Met behulp van deze referentie-implementatie kan zowel de performance als juistheid van de door de Murray codegenerator geproduceerde code worden beoordeeld;
3. de specificatie is zonder veel uitleg te begrijpen, omdat het achterliggende concept van bevolkingsgroei eenvoudig en voor iedereen duidelijk is;
4. er is geen sprake van grote hoeveelheden data, waardoor de nadruk kan worden gelegd op codegeneratie in plaats van data-organisatie;
5. het demografisch prognosemodel bestaat uit twee delen die ook los van elkaar uitgerekend kunnen worden. Het arbeidsmarkt-submodel kent slechts eenvoudige bewerkingen waardoor het ideaal is voor een eerste basisimplementatie van het Murray framework. Het demografisch submodel daarentegen bevat een aantal moeilijkheden die evenwel later aan bod kunnen komen:
   a) de processen kunnen niet worden uitgerekend in de gespecificeerde volgorde. De juiste uitrekenvolgorde moet eerst worden bepaald. Bovendien bevat het submodel iteratieve berekeningen;
   b) in dit submodel worden zogenaamde deelindices gedefinieerd en gebruikt; dit type indices wordt gebruikt bij complexe aggregaties;

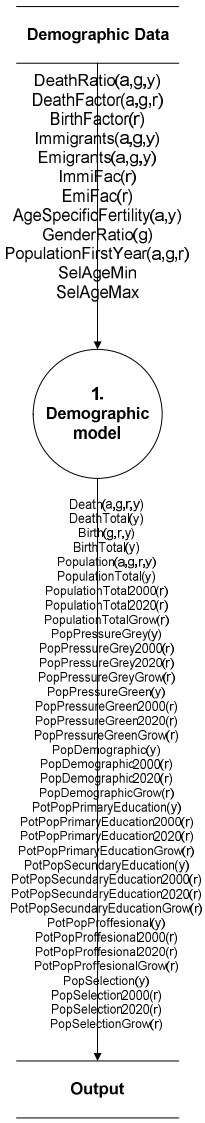
c) de berekeningen in dit submodel zijn complexer en uitgebreider dan die uit het arbeidsmarkt-submodel.

De volgende tabel geeft een opsomming van de in de modelspecificatie gebruikte indices.

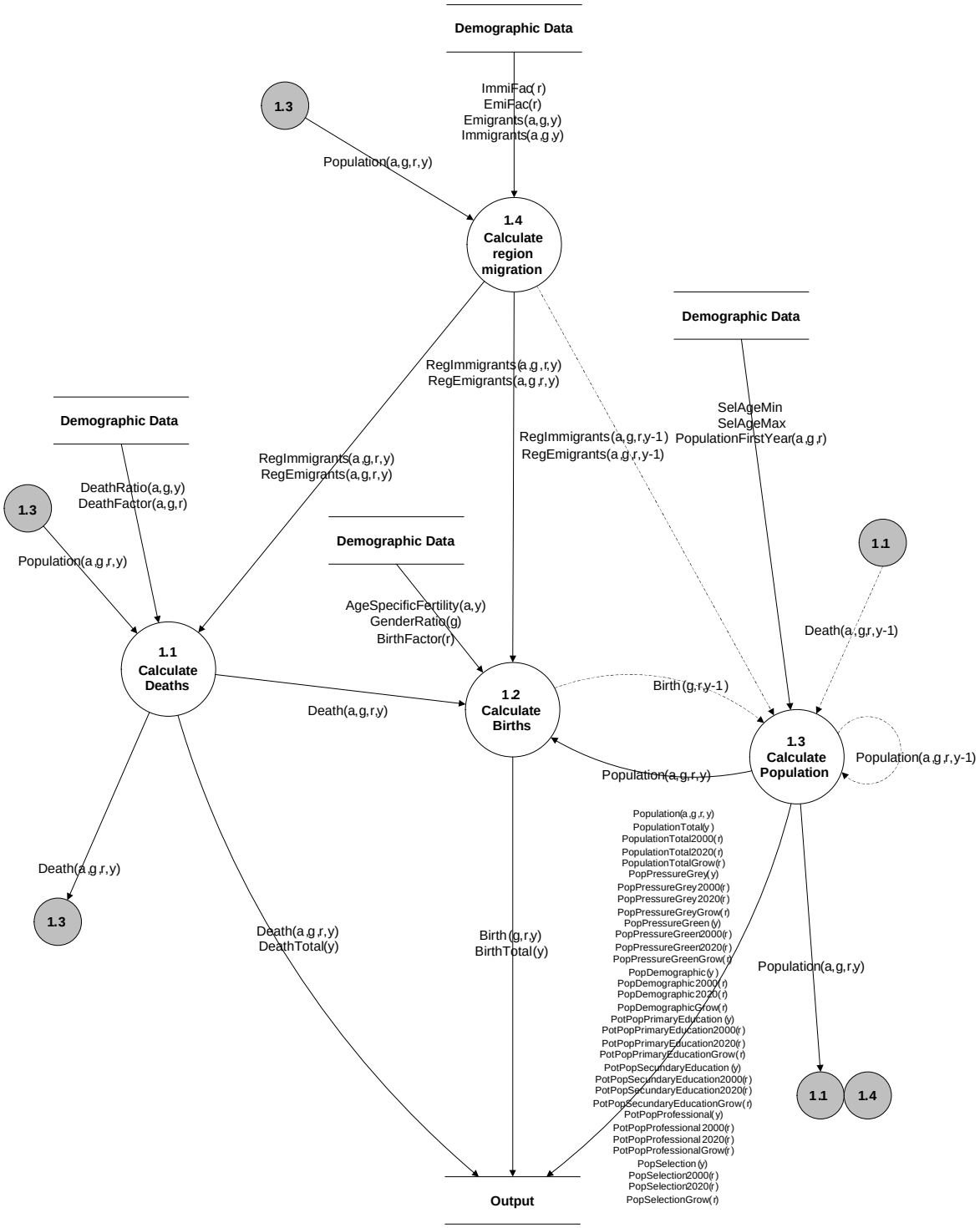
Index	**Afkorting**	**Bereik**	**Enumeratie**
Age	a	0-99	0=0, …, 99-99
Gender	g	1-2	1=male, 2=female
Region	r	1-40	*enumeratie wordt niet gebruikt*
Year	y	0-20	0=2000, …, 20=2020

De volgende bladzijden geven de volledige specificatie van het demografisch prognosemodel.

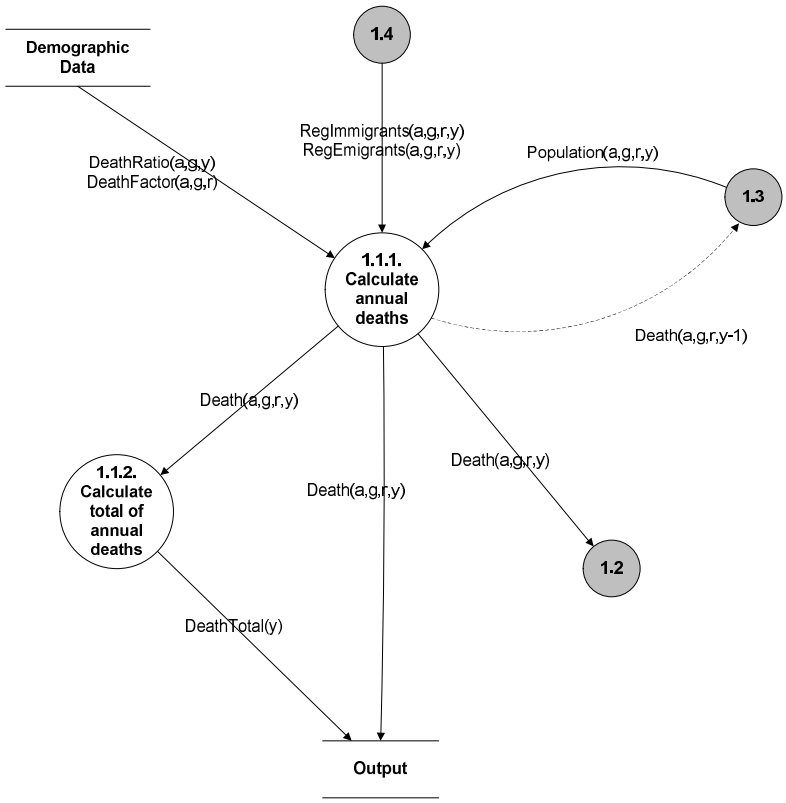
1. Demographic model - Global View



1. Demographic model



1.1. Calculate Deaths



1.1.1. Calculate annual deaths

$$dea_{a,g,r,y} = dra_{a,g,y} \cdot \left(pop_{a,g,r,y} + \frac{rim_{a,g,r,y} - rem_{a,g,r,y}}{2} \right) \cdot dfa_{a,g,r}$$

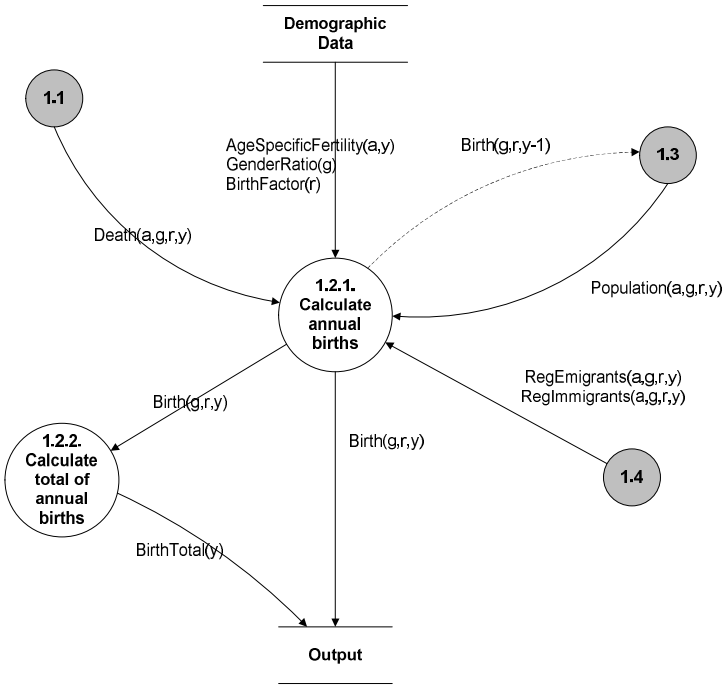
dea	**Death**
dra	**DeathRatio**
rim	**RegImmigrants**
rem	**RegEmigrants**
pop	**Population**
dfa	**DeathFactor**

1.1.2. Calculate total of annual deaths

$$dto_y = \sum_a \sum_g \sum_r dea_{a,g,r,y}$$

dto	**DeathTotal**
dea	**Death**

1.2 Calculate Births



1.2.1. Calculate annual births

$$\mathbf{a1} = \{ a1 \mid a1 \in a.(a1 \geq 15 \wedge a1 \leq 49) \}$$

$$bir_{g,r,y} = gra_g \cdot \sum_{a1} \left(asf_{a1,y} \cdot \left(pop_{a1,g='\text{female}',r,y} + \frac{rim_{a1,g='\text{female}',r,y} - rem_{a1,g='\text{female}',r,y} - dea_{a1,g='\text{female}',r,y}}{2} \right) \right) \cdot bfa_r$$

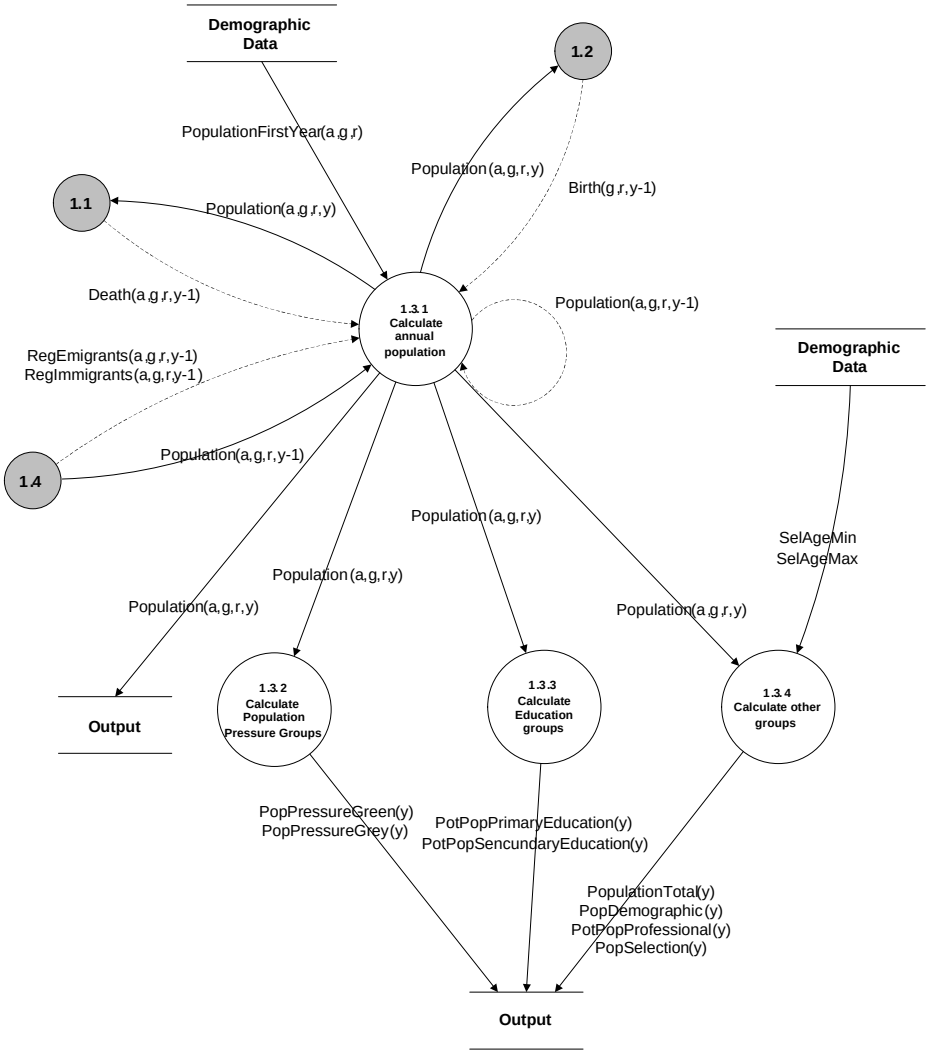
asf	**AgeSpecificFertility**
bir	**Birth**
dea	**Death**
rem	**RegEmigrants**
gra	**GenderRatio**
rim	**RegImmigrants**
pop	**Population**
bfa	**BirthFactor**

1.2.2 Calculate total of annual births

$$bto_y = \sum_g \sum_r bir_{g,r,y}$$

bto	**BirthTotal**
bir	**Birth**

1.3 Calculate Populationgroups

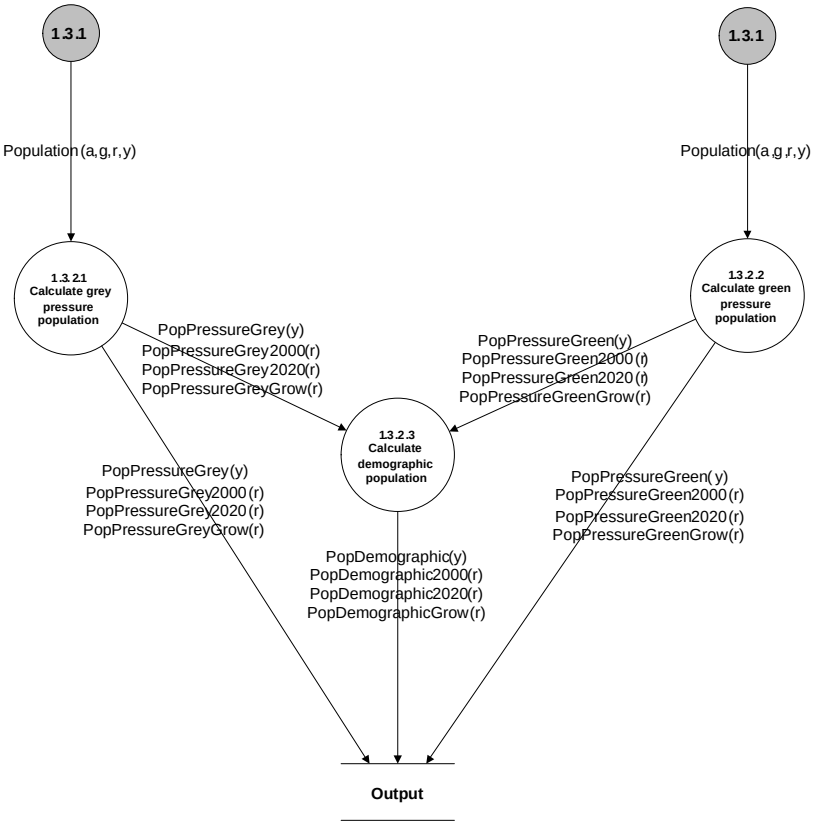


1.3.1. Calculate annual population

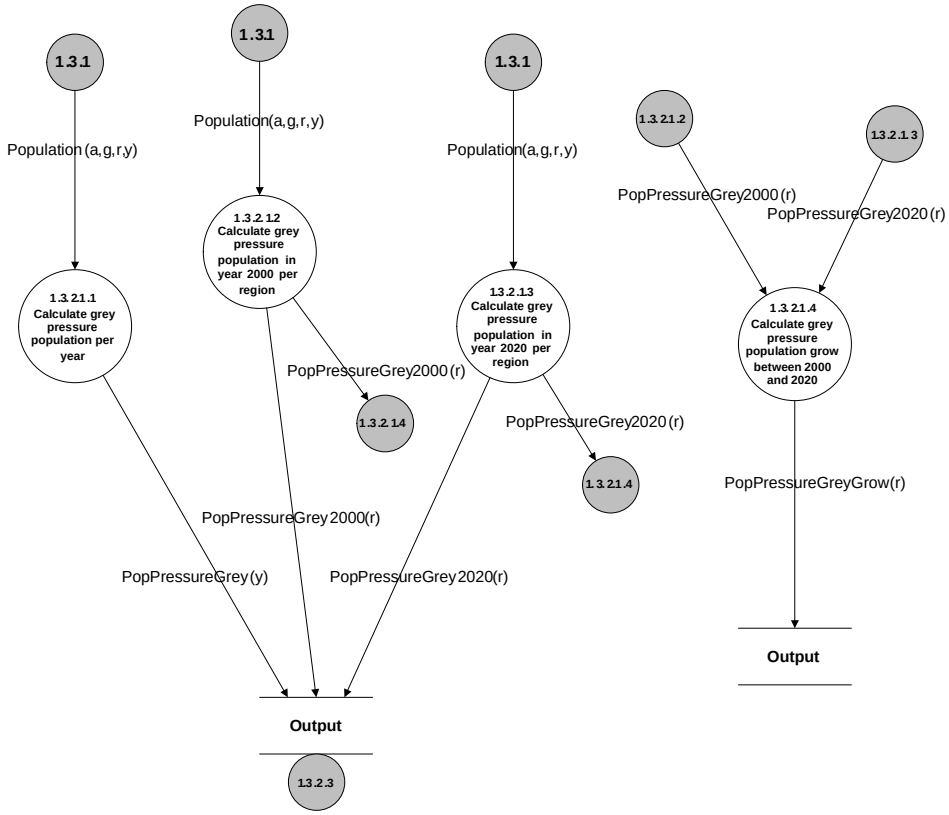
$$pop_{a,g,y,r} = \begin{cases} pop0_{a,g,r} & y = \min(\mathbf{y}) \\ bir_{g,r,y-1} & y > \min(\mathbf{y}) \wedge a = \min(\mathbf{a}) \\ pop_{a-1,g,r,y-1} - dea_{a-1,g,r,y-1} + rim_{a-1,g,r,y-1} - rem_{a-1,g,r,y-1} & \end{cases}$$

dea	**Death**
bir	**Birth**
rem	**RegEmigrants**
rim	**RegImmigrants**
pop	**Population**
pop0	**PopulationFirstYear**

1.3.2 Calculate Population Pressure groups



1.3.2.1 Calculate grey pressure population



1.3.2.1.1 Calculate grey pressure population per year

$$\mathbf{a1} = \{ a1 \mid a1 \in \mathbf{a}.(a1 \geq 65 \wedge a1 \leq 99) \}$$

$$ppgrey_y = \sum_{a1} \sum_{g} \sum_{r} pop_{a1,g,r,y}$$

ppgrey PopPressureGrey
pop Population

1.3.2.1.3 Calculate grey pressure population in year 2020 per region

$$\mathbf{a1} = \{ a1 \mid a1 \in \mathbf{a}.(a1 \geq 65 \wedge a1 \leq 99) \}$$

$$ppgrey_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2020}$$

ppgrey PopPressureGrey2020
pop Population

1.3.2.1.2 Calculate grey pressure population in year 2000 per region

$$\mathbf{a1} = \{ a1 \mid a1 \in \mathbf{a}.(a1 \geq 65 \wedge a1 \leq 99) \}$$

$$ppgrey_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2000}$$

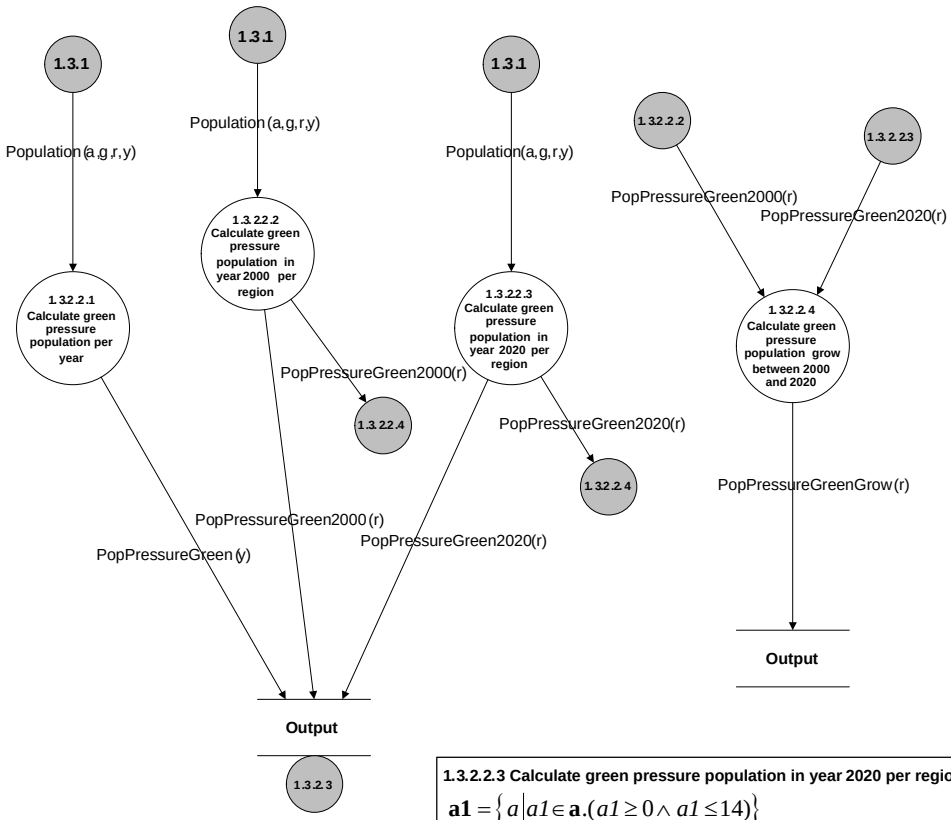
ppgrey PopPressureGrey2000
pop Population

1.3.2.1.4 Calculate grey pressure population grow

$$ppgrow_r = ppgrey2020_r / ppgrey2000_r$$

ppgrey2000 PopPressureGrey2000
ppgrey2020 PopPressureGrey2020
ppgrow PopPressureGreyGrow

1.3.2.2 Calculate green pressure population



1.3.2.2.1 Calculate green pressure population per year

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}.(a1 \geq 0 \wedge a1 \leq 14)\}$$

$$ppgreen_y = \sum_{a1} \sum_{g} \sum_{r} pop_{a1,g,r,y}$$

ppgreen	**PopPressureGreen**
pop	**Population**

1.3.2.2.2 Calculate green pressure population in year 2000 per region

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}.(a1 \geq 0 \wedge a1 \leq 14)\}$$

$$ppgreen_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2000}$$

ppgreen	**PopPressureGreen2000**
pop	**Population**

1.3.2.2.3 Calculate green pressure population in year 2020 per region

$$\mathbf{a1} = \{a \mid a1 \in \mathbf{a}.(a1 \geq 0 \wedge a1 \leq 14)\}$$

$$ppgreen_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2020}$$

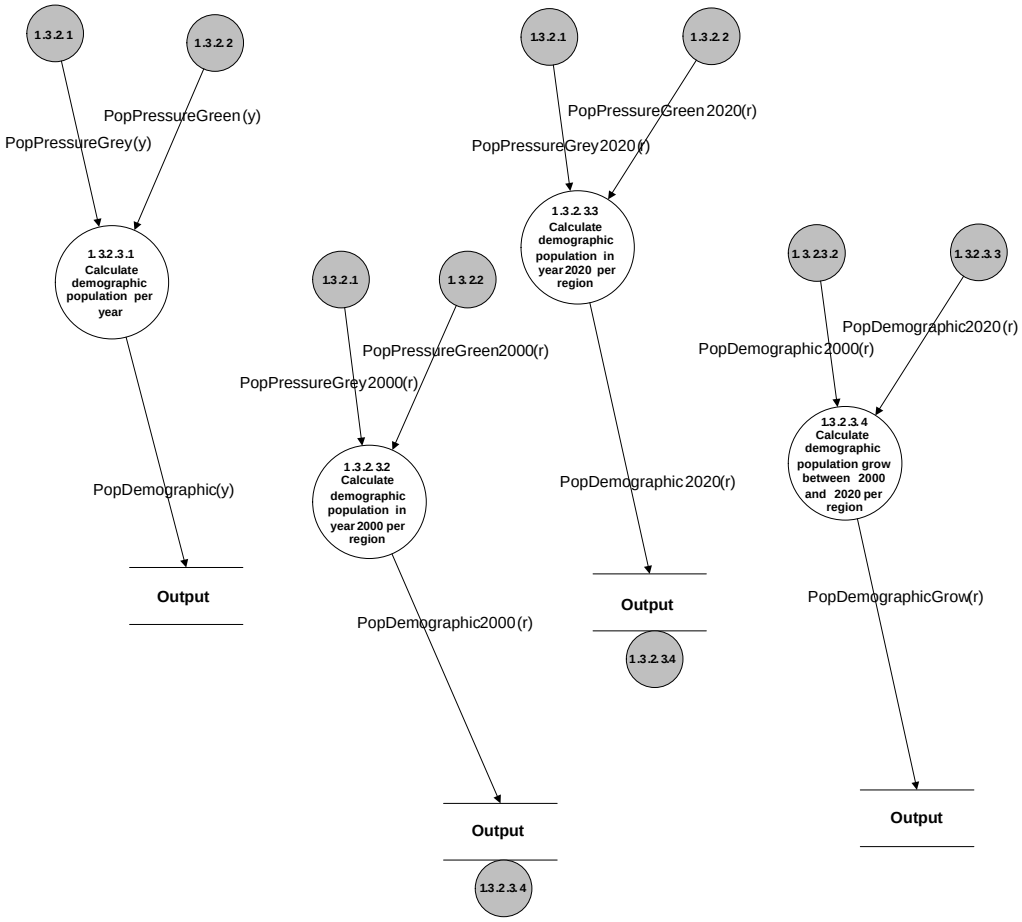
ppgreen	**PopPressureGreen2020**
pop	**Population**

1.3.2.2.4 Calculate green pressure population grow

$$ppgrow_r = ppgreen2020_r / ppgreen2000_r$$

ppgreen2000	**PopPressureGreen2000**
ppgreen2020	**PopPressureGreen2020**
ppgrow	**PopPressureGreenGrow**

1.3.2.3 Calculate demographic population



1.3.2.3.1 Calculate demographic population per year

$$pdem_y = ppgrey_y + ppgreen_y$$

pdem PopDemographic
ppgrey PopPressureGreen
ppgrey PopPressureGrey

1.3.2.3.3 Calculate demographic population in year 2020 per region

$$pdem_y = ppgrey_y + ppgreen_y$$

pdem PopDemographic2020
ppgrey PopPressureGreen2020
ppgrey PopPressureGrey2020

1.3.2.3.1 Calculate demographic population in year 2000 per region

$$pdem_y = ppgrey_y + ppgreen_y$$

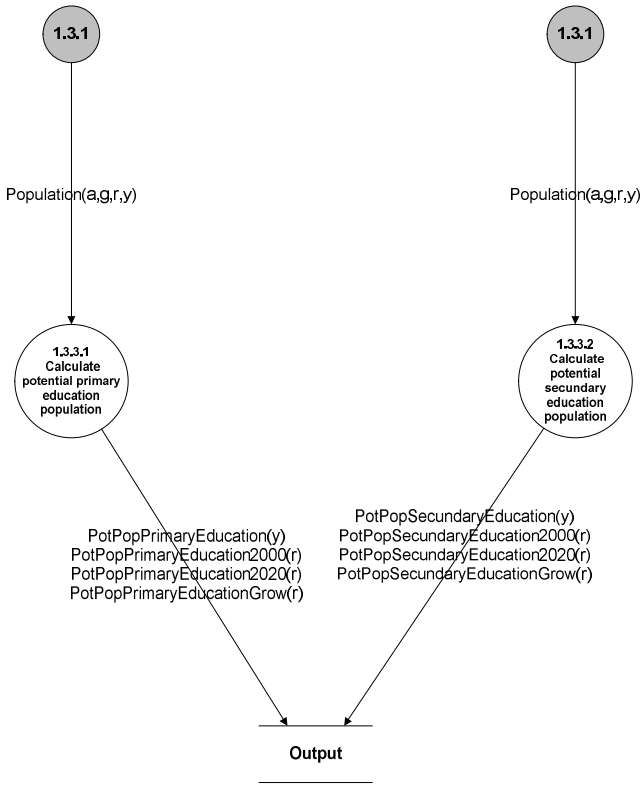
pdem PopDemographic2000
ppgrey PopPressureGreen2000
ppgrey PopPressureGrey2000

1.3.2.3.4 Calculate demographic population Grow

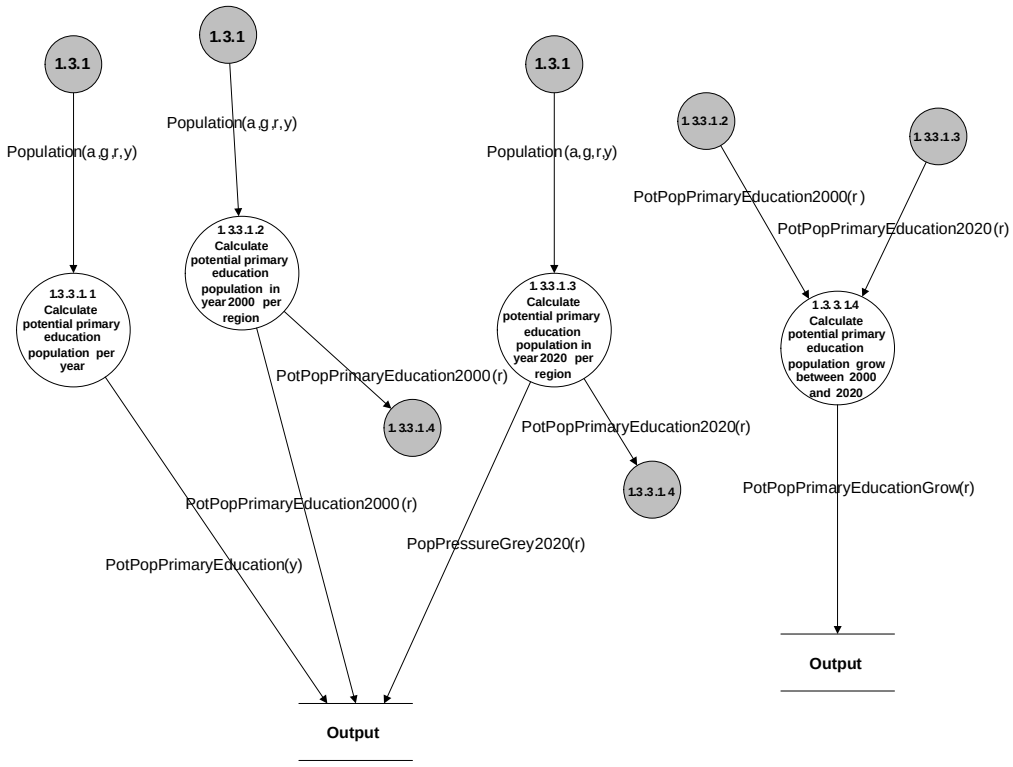
$$ppgrow_r = pdem2020_r / pdem2000_r$$

pdem2000	**PopDemographic2000**
pdem2020	**PopDemographic2020**
ppgrow	**PopDemographicGrow**

1.3.3 Calculate education groups



1.3.3.1 Calculate potential primary education population



1.3.3.1.1 Calculate potential primary education population per year

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}.(a1 \geq 4 \wedge a1 \leq 12)\}$$

$$pppedu_y = \sum_{a1} \sum_{g} \sum_{r} pop_{a1,g,r,y}$$

pppedu	**PotPopPrimaryEducation**
pop	**Population**

1.3.3.1.3 Calculate potential primary education population in year 2020 per region

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}.(a1 \geq 4 \wedge a1 \leq 12)\}$$

$$pppedu_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2020}$$

pppedu	**PotPopPrimaryEducation2020**
pop	**Population**

1.3.3.1.2 Calculate potential primary education population in year 2000 per region

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}.(a1 \geq 4 \wedge a1 \leq 12)\}$$

$$pppedu_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2000}$$

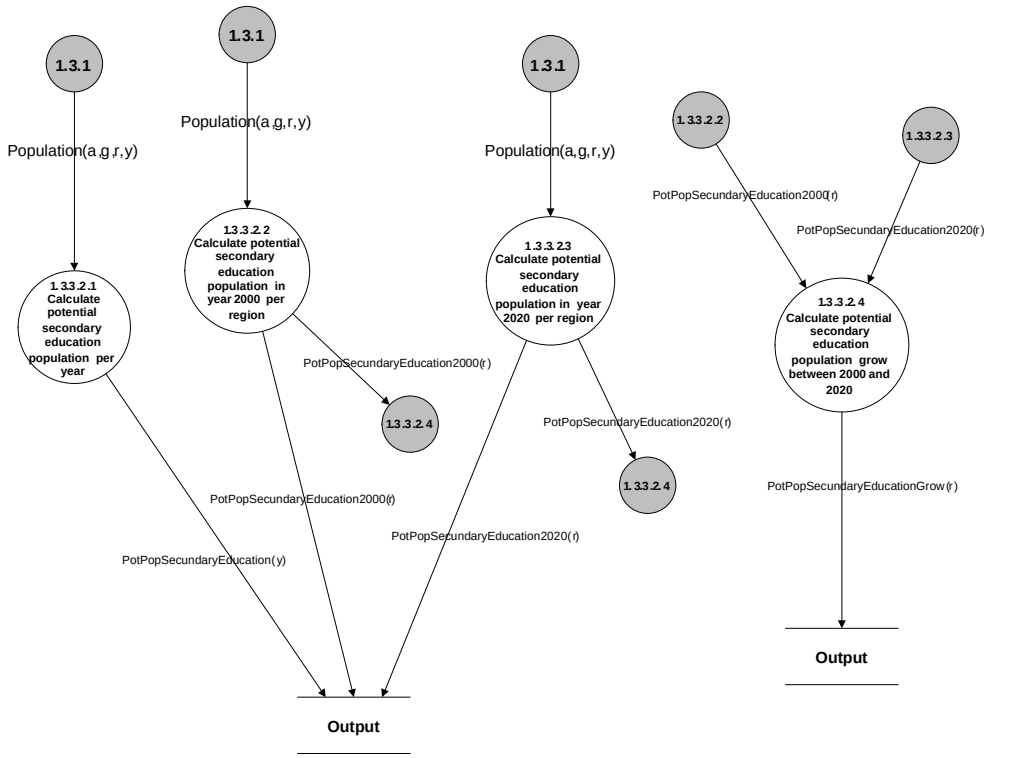
pppedu	**PotPopPrimaryEducation2000**
pop	**Population**

1.3.3.1.4 Calculate potential primary education population grow

$$ppgrow_r = pppedu2020_r / pppedu2000_r$$

pppedu2000	**PotPopPrimaryEducation2000**
pppedu2020	**PotPopPrimaryEducation2020**
ppgrow	**PotPopPrimaryEducationGrow**

1.3.3.2 Calculate potential secundary education population



1.3.3.2.1 Calculate potential secundary education population per year

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}.(a1 \geq 13 \wedge a1 \leq 18)\}$$

$$ppsedu_y = \sum_{a1} \sum_{g} \sum_{r} pop_{a1,g,r,y}$$

ppsedu	**PotPopSecundaryEducation**
pop	**Population**

1.3.3.2.3 Calculate potential secundary education population in year 2020 per region

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}.(a1 \geq 13 \wedge a1 \leq 18)\}$$

$$ppsedu_r = \sum_{a1} \sum_{g} \sum_{y1} pop_{a1,g,r,y=2020}$$

ppsedu	**PotPopSecundaryEducation2020**
pop	**Population**

1.3.3.2.2 Calculate potential secundary education population in year 2000 per region

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}.(a1 \geq 13 \wedge a1 \leq 18)\}$$

$$ppsedu_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2000}$$

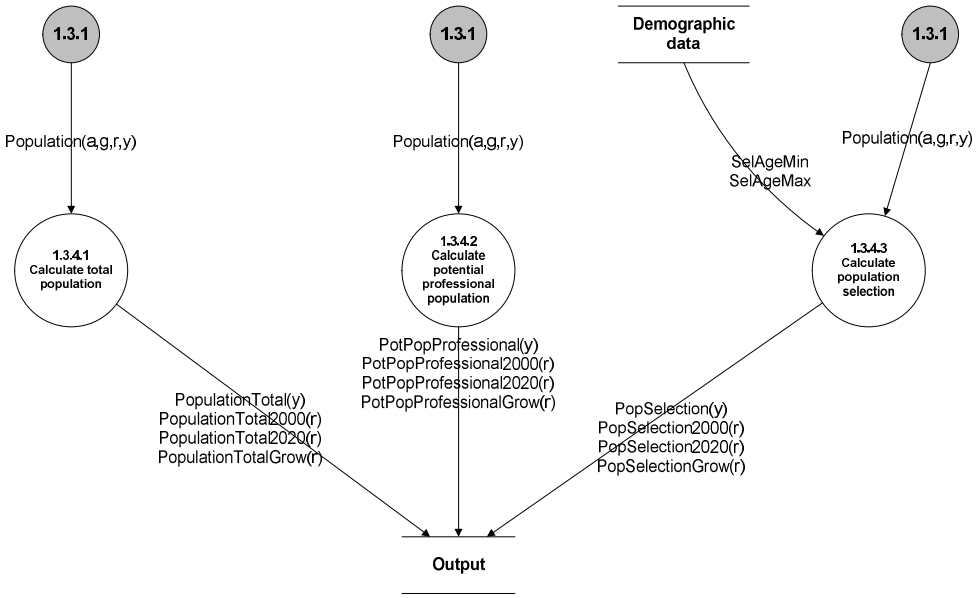
ppsedu	**PotPopSecundaryEducation2000**
pop	**Population**

1.3.3.2.4 Calculate potential secundary education population grow

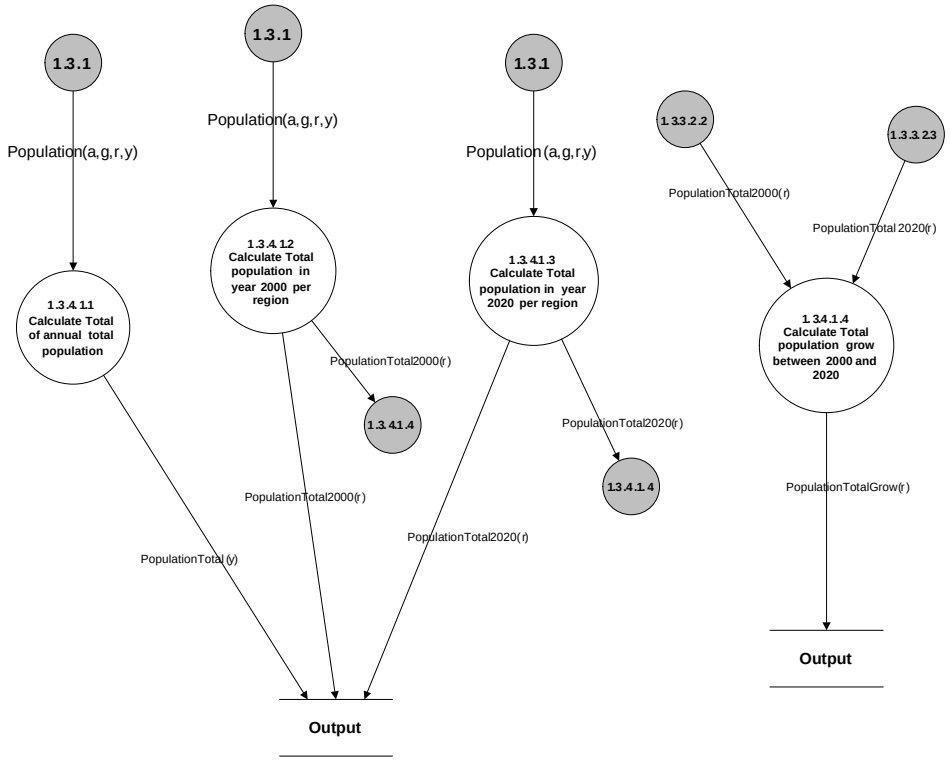
$$ppgrow_r = ppsedu2020_r / ppsedu2000_r$$

ppsedu2000	**PotPopSecundaryEducation2000**
ppsedu2020	**PotPopSecundaryEducation2020**
ppgrow	**PotPopSecundaryEducationGrow**

1.3.4 Calculate other groups



1.3.4.1 Calculate Total population



1.3.4.1.1 Calculate total of annual population

$$\mathbf{a1} = \{ a1 \mid a1 \in \mathbf{a}.(a1 \geq 0 \wedge a1 \leq 99) \}$$

$$pto_y = \sum_{a1} \sum_{g} \sum_{r} pop_{a1,g,r,y}$$

pto **PopulationTotal**
pop **Population**

1.3.4.1.3 Calculate total population in year 2020 per region

$$\mathbf{a1} = \{ a1 \mid a1 \in \mathbf{a}.(a1 \geq 0 \wedge a1 \leq 99) \}$$

$$pto_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2020}$$

pto **PopulationTotal2020**
pop **Population**

1.3.4.1.2 Calculate total population in year 2000 per region

$$\mathbf{a1} = \{ a1 \mid a1 \in \mathbf{a}.(a1 \geq 0 \wedge a1 \leq 99) \}$$

$$pto_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2000}$$

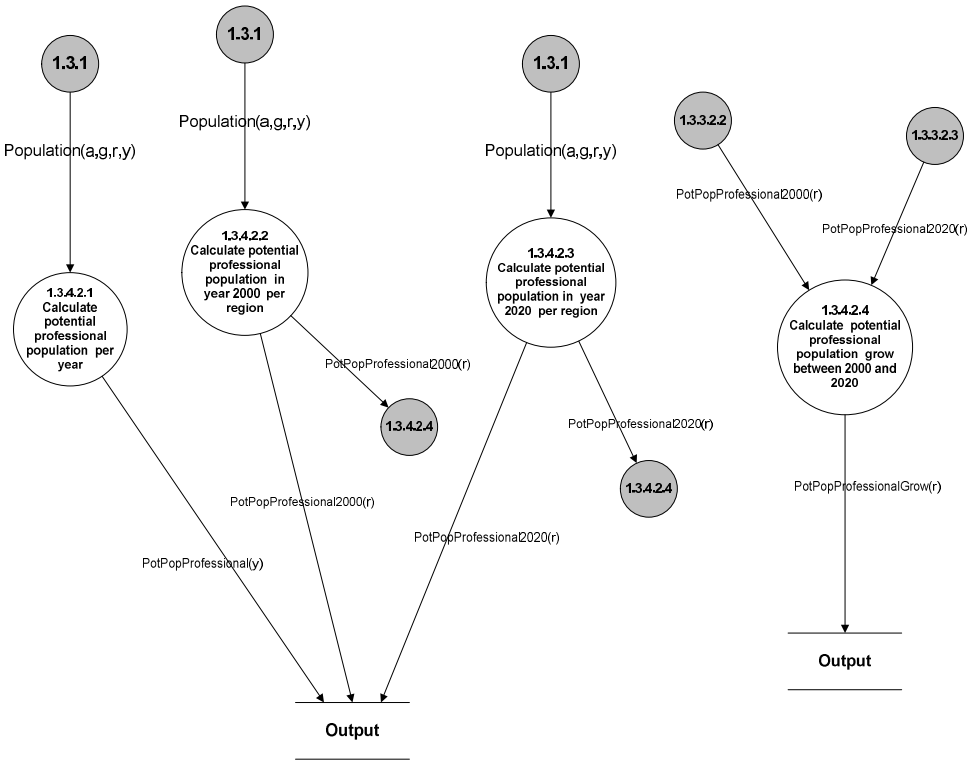
pto **PopulationTotal2000**
pop **Population**

1.3.4.1.4 Calculate total population grow

$$ptgrow_r = pto2020_r \; / \; pto2000_r$$

pto2000 **PopulationTotal2000**
pto2020 **PopulationTotal2020**
ptgrow **PopulationTotalGrow**

1.3.4.2 Calculate potential professional population



1.3.4.2.1 Calculate potential professional population per year

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}.(a1 \geq 15 \wedge a1 \leq 64)\}$$

$$pppro_y = \sum_{a1} \sum_{g} \sum_{r} pop_{a1,g,r,y}$$

pppro PotPopProfessional
pop Population

1.3.4.2.3 Calculate potential professional population in year 2020 per region

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}.(a1 \geq 15 \wedge a1 \leq 64)\}$$

$$pppro_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2020}$$

pppro PotPopProfessional2020
pop Population

1.3.4.2.2 Calculate potential professional population in year 2000 per region

$$\mathbf{a1} = \{a1 \mid a1 \in \mathbf{a}.(a1 \geq 15 \wedge a1 \leq 64)\}$$

$$pppro_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2000}$$

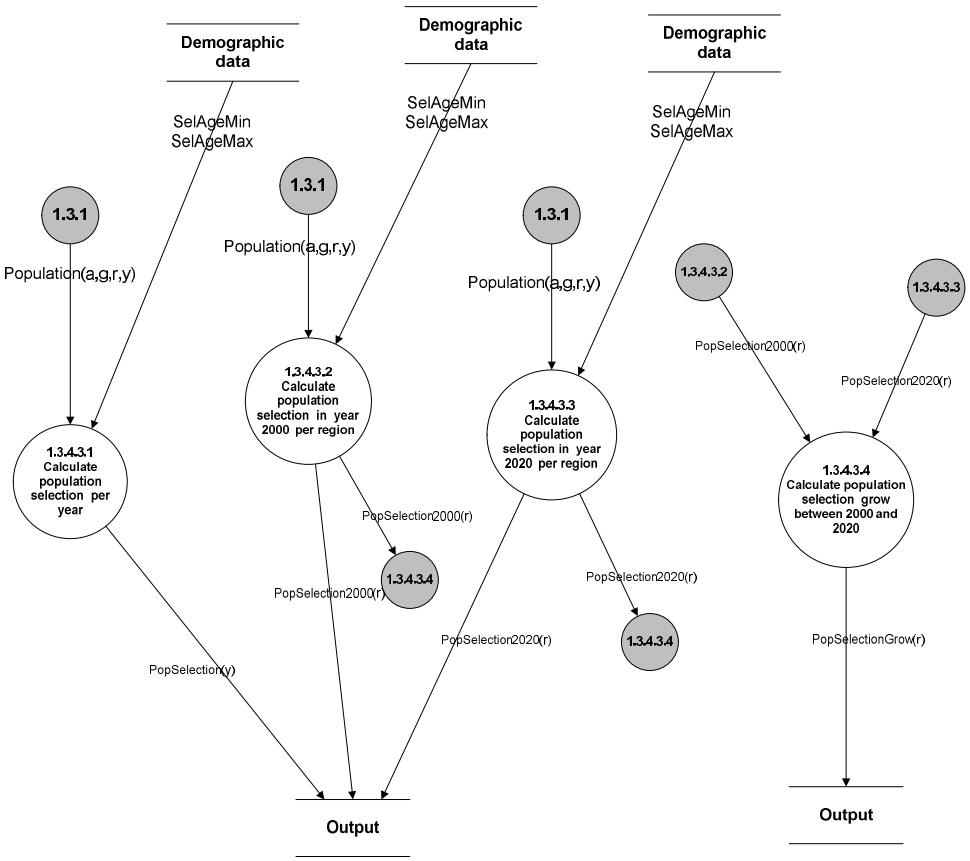
pppro PotPopProfessional2000
pop Population

1.3.4.2.4 Calculate potential professional population grow

$$ppgrow_r = pppro2020_r / pppro2000_r$$

pppro2000 PotPopProfessional2000
pppro2020 PotPopProfessional2020
ppgrow PotPopProfessionalGrow

1.3.4.3 Calculate population selection



1.3.4.3.1 Calculate Population Selection per year

$$\mathbf{a1} = \left\{ a1 \middle| a1 \in \mathbf{a}.(a1 \geq smi \wedge a1 \leq sma) \right\}$$

$$ps_y = \sum_{a1} \sum_{g} \sum_{r} pop_{a1,g,r,y}$$

ps	**PopSelection**
pop	**Population**
smi	**SelAgeMin**
sma	**SelAgeMax**

1.3.4.3.3 Calculate Population Selection in year 2020 per region

$$\mathbf{a1} = \left\{ a1 \middle| a1 \in \mathbf{a}.(a1 \geq smi \wedge a1 \leq sma) \right\}$$

$$ps_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2020}$$

ps	**PopSelection2020**
pop	**Population**
smi	**SelAgeMin**
sma	**SelAgeMax**

1.3.4.3.2 Calculate Population Selection in year 2000 per region

$$\mathbf{a1} = \left\{ a1 \middle| a1 \in \mathbf{a}.(a1 \geq smi \wedge a1 \leq sma) \right\}$$

$$ps_r = \sum_{a1} \sum_{g} pop_{a1,g,r,y=2000}$$

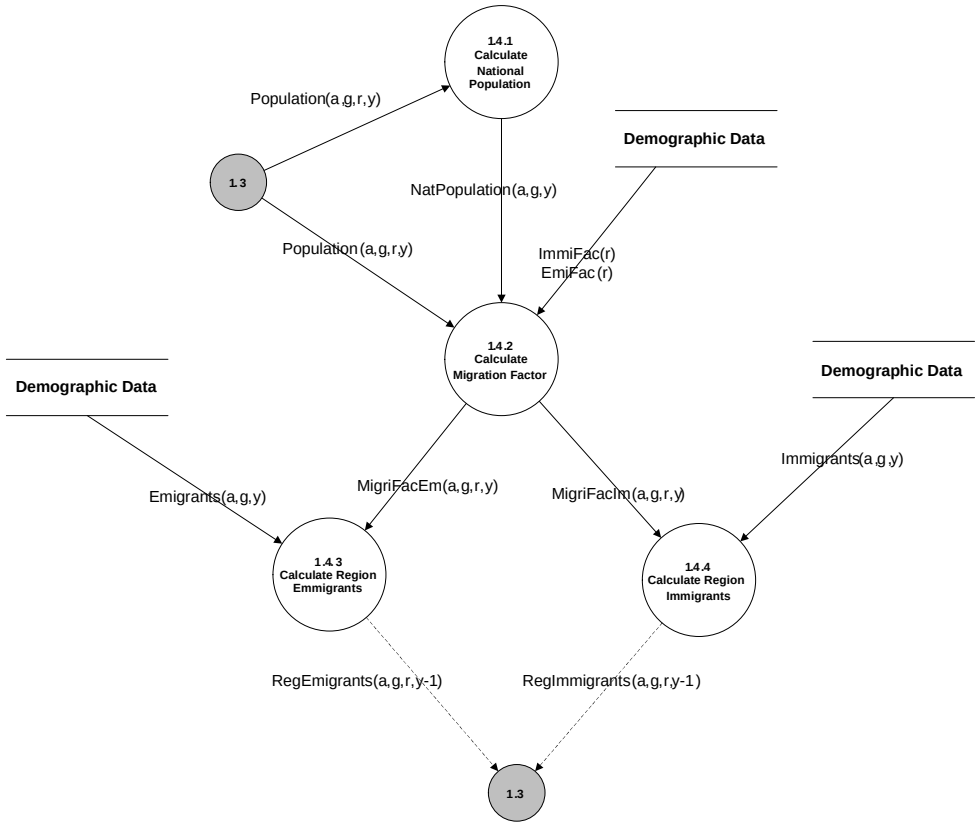
ps	**PopSelection2000**
pop	**Population**
smi	**SelAgeMin**
sma	**SelAgeMax**

1.3.4.3.4 Calculate Population Selection grow

$$ppgrow_r = pppro2020_r \,/\, pppro2000_r$$

ps2000	**PopSelection2000**
ps2020	**PopSelection2020**
psgrow	**PopSelectionGrow**

1.4 Calculate region migration



1.4.1 Calculate national population

$$npo_{a,g,y} = \sum_{r} pop_{a,g,r,y}$$

npo	**NatPopulation**
pop	**Population**

1.4.3 Calculate region emmigrants

$$rem_{a,g,r,y} = emg_{a,g,y} \cdot mfe_{a,g,r,y}$$

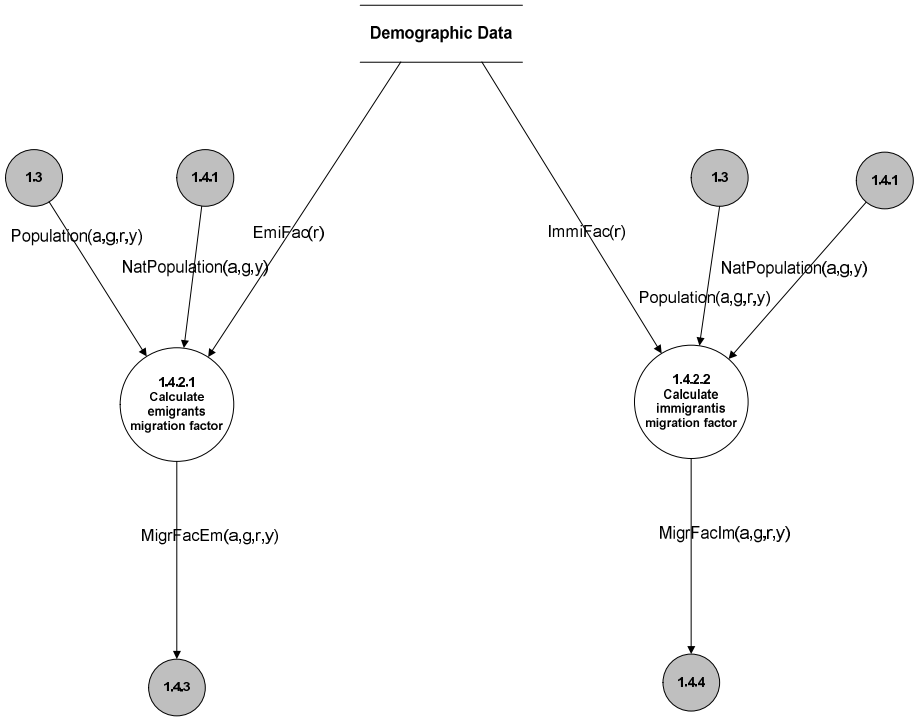
rem	**RegEmigrants**
emg	**Emigrants**
mfe	**MigrFacEm**

1.4.4 Calculate region immigrants

$$rim_{a,g,r,y} = img_{a,g,y} \cdot mfi_{a,g,r,y}$$

rim	**RegImmigrants**
img	**Immigrants**
mfi	**MigrFacIm**

1.4.2 Calculate region migration factor



1.4.2.1 Calculate emigrants migration factor

$$mfe_{a,a,r,v} = \frac{pop_{a,g,r,y}}{npo_{a,g,y}} \cdot efa_r$$

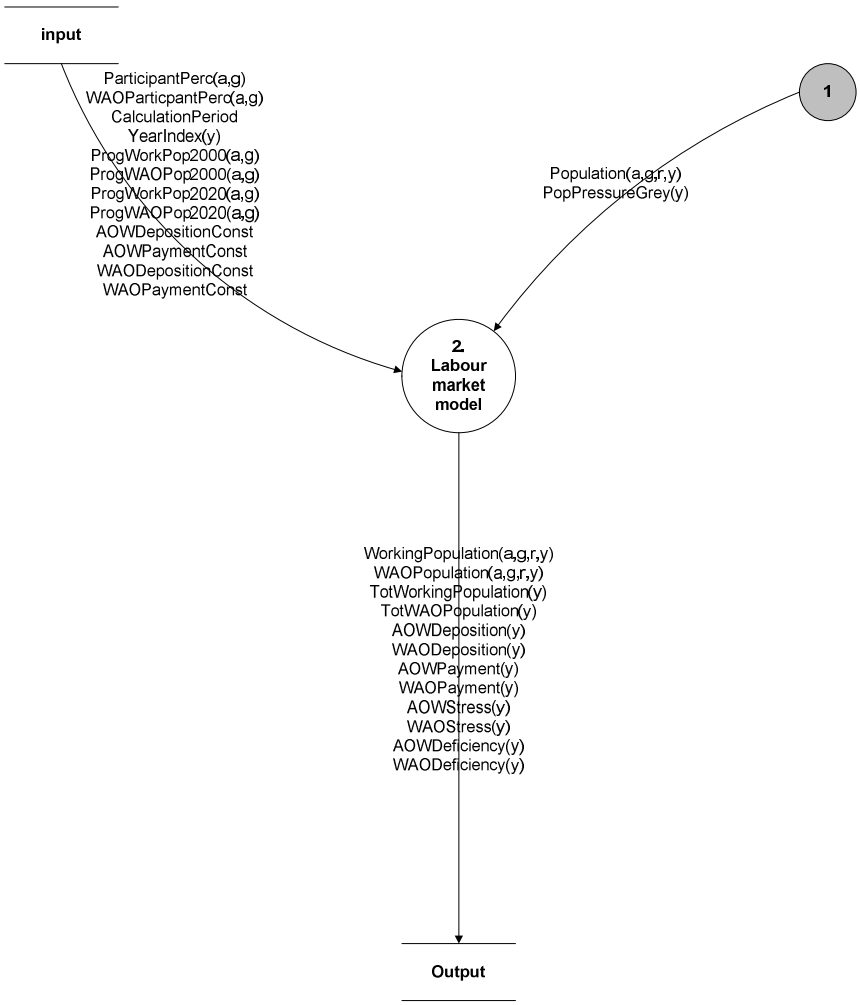
npo	**NatPopulation**
pop	**Population**
mfe	**MigrFacEm**
efa	**EmiFactor**

1.4.2.1 Calculate immgants migration factor

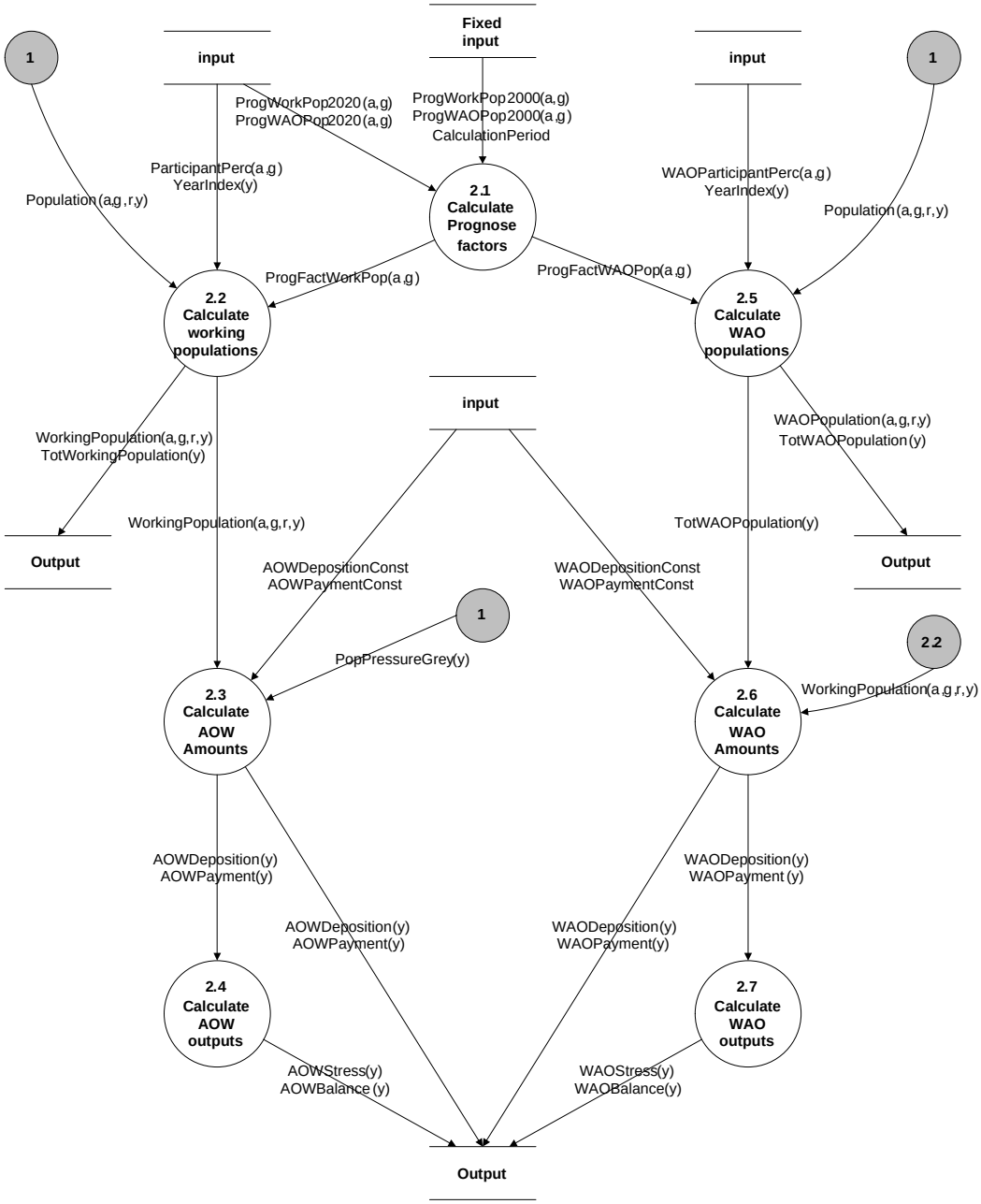
$$mfi_{a,g,r,y} = \frac{pop_{a,g,r,y}}{npo_{a,g,y}} \cdot ifa_r$$

npo	**NatPopulation**
pop	**Population**
mfi	**MigFacIm**
ifa	**ImmiFactor**

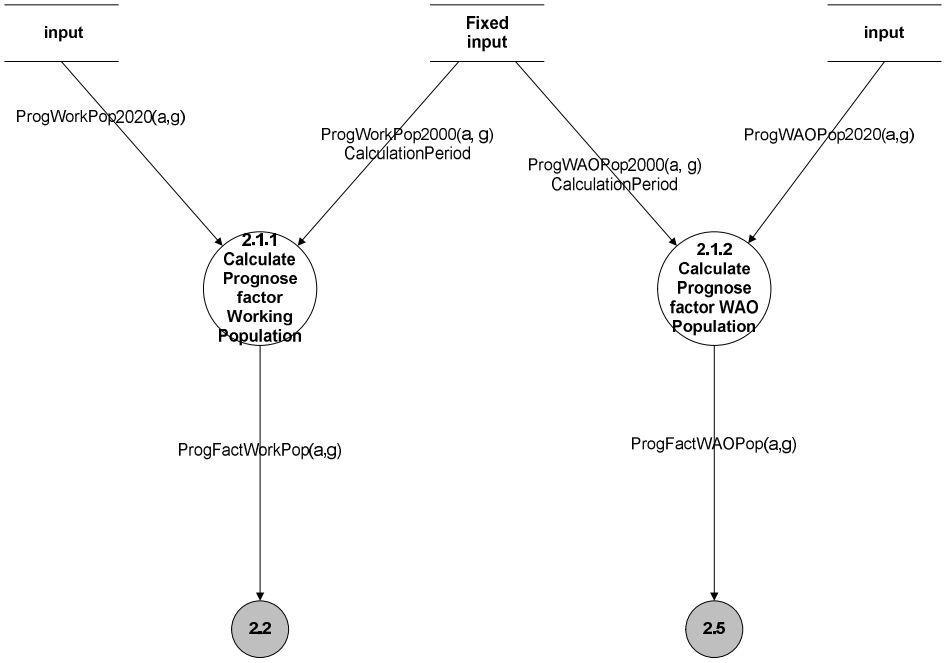
2. Labourmarket model - Global view



2. Labourmarket model



2.1 Calculate prognose factors



2.1.1 Calculate Prognose factor Working Population

$$pfwp_{a,g} = \frac{\left(\left(pwp20_{a,g} / pwp00_{a,g} \right) - 1\right)}{cp}$$

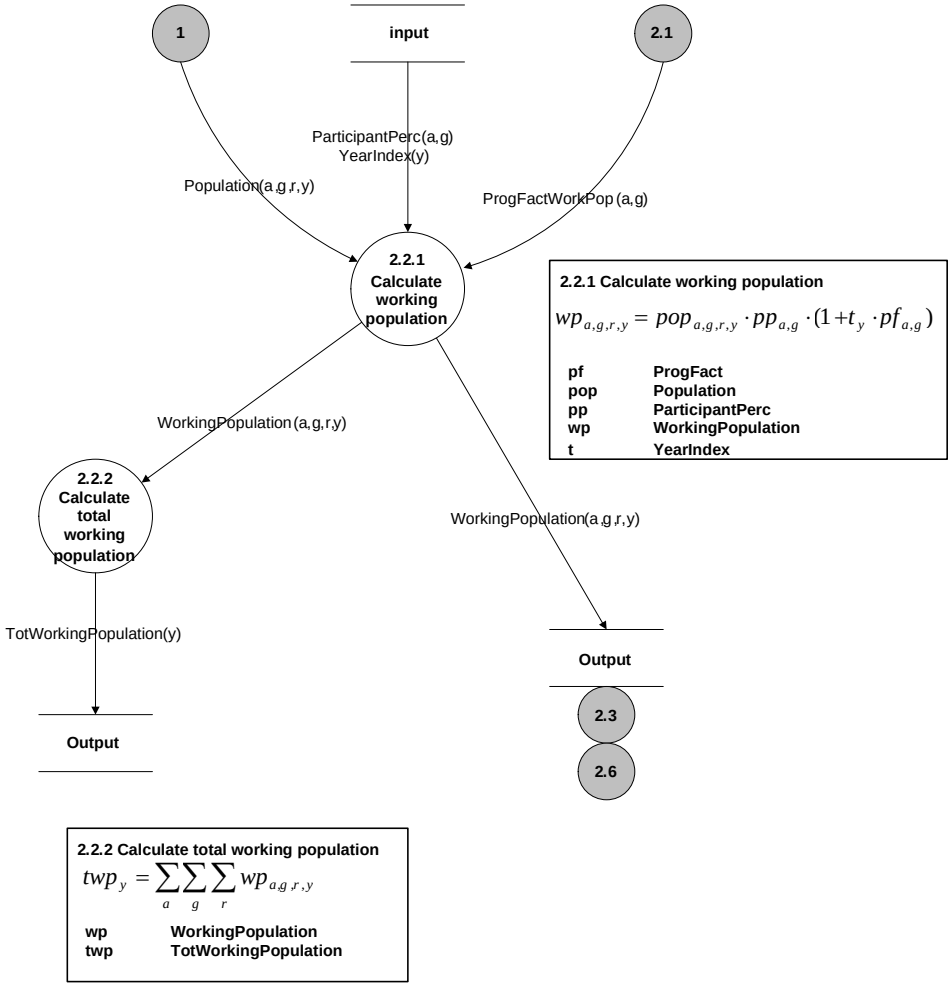
pwp00	ProgWorkPop2000
pwp10	ProgWorkPop2010
cp	CalculationPeriod
pfwp	ProgFactWorkPop

2.1.2 Calculate Prognose factor WAO Population

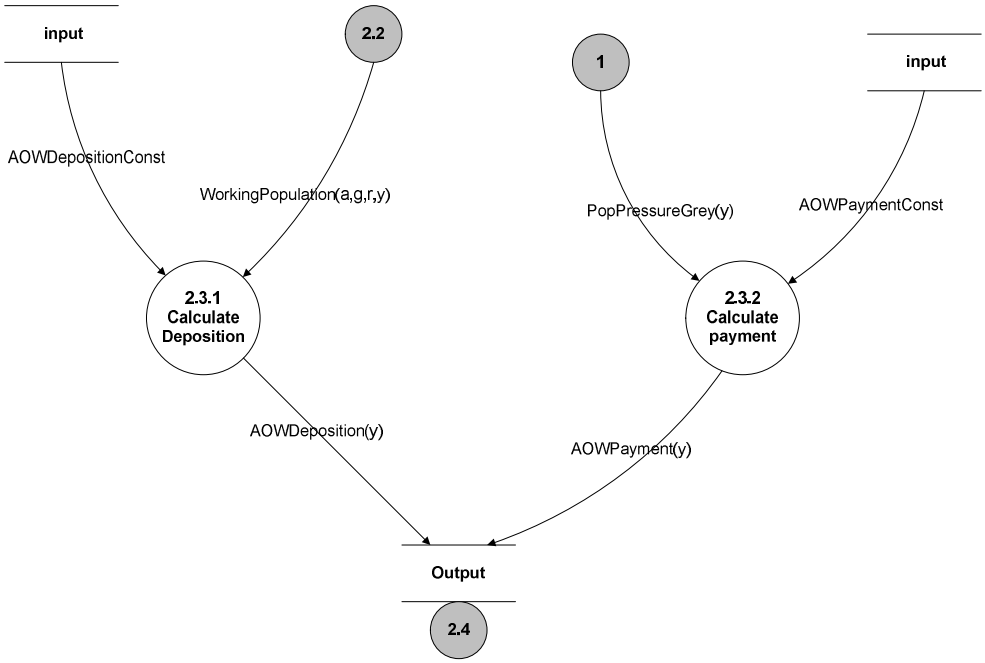
$$pfap_{a,g} = \frac{\left(\left(pap20_{a,g} / pap00_{a,g} \right) - 1\right)}{cp}$$

pap00	ProgWAOPop2000
pap10	ProgWAOPop2010
cp	CalculationPeriod
pfap	ProgFactWAOPop

2.2 Calculate working populations



2.3 Calculate AOW amounts



2.3.1 Calculate AOW Deposition

$$dp_y = \sum_a \sum_g \sum_r wp_{a\ g\ r\ y} \cdot dc$$

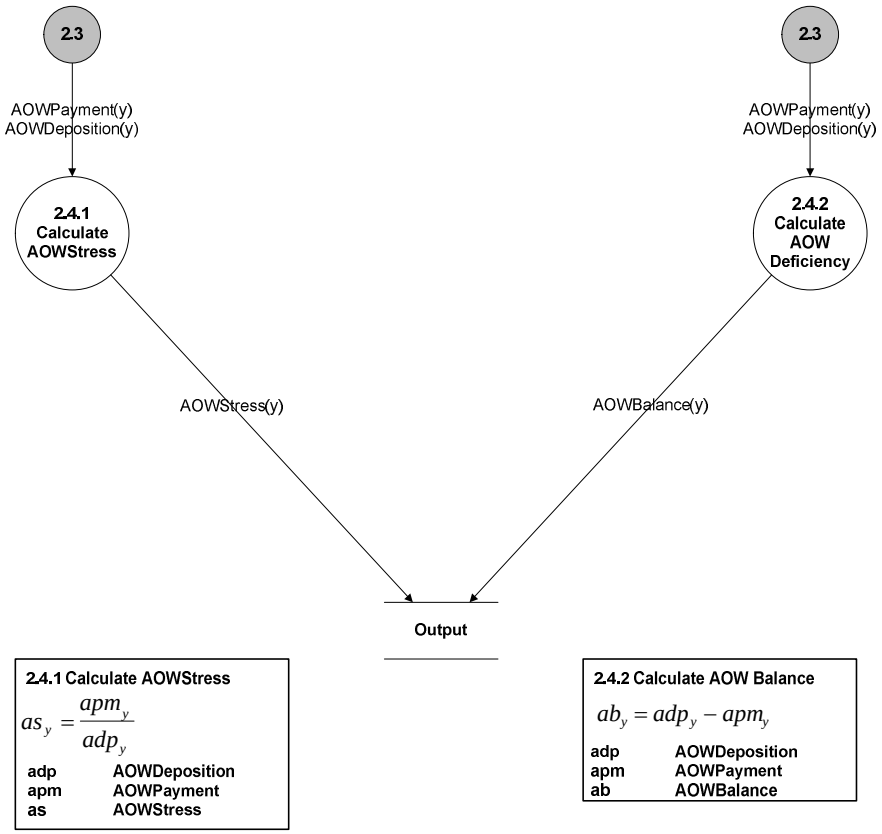
adc	AOWDepositionConst
wp	WorkingPopulation
adp	AOWDeposition

2.3.2 Calculate AOW Payment

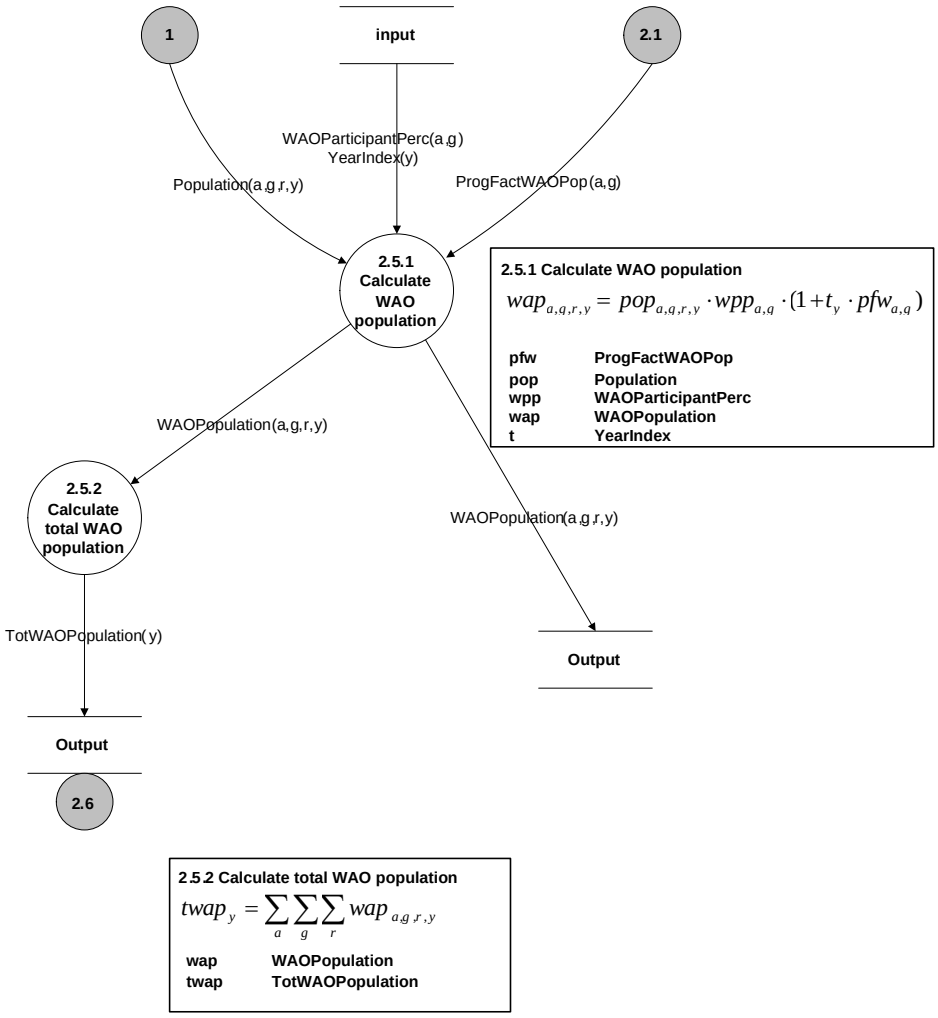
$$apm_y = ppg_y \cdot apc$$

apc	AOWPaymentConst
ppg	PopPressureGrey
apm	AOWPayment

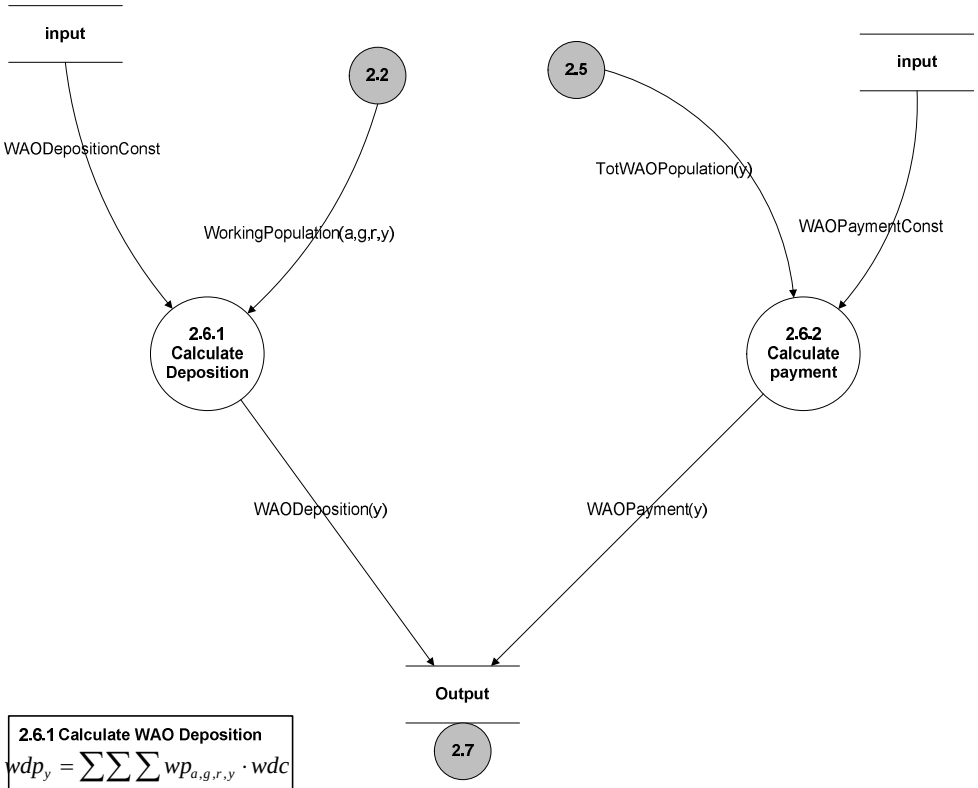
2.4 Calculate AOW outputs



2.5 Calculate WAO populations



2.6 Calculate WAO amounts



2.6.1 Calculate WAO Deposition

$$wdp_y = \sum_a \sum_g \sum_r wp_{a,g,r,y} \cdot wdc$$

wdc	WAODepositionCons t
wp	WorkingPopulation
wdp	WAODeposition

2.6.2 Calculate WAO Payment

$$wpm_y = twp_y \cdot wpc$$

wpc	WAOPaymentConst
twp	TotWAOPopulation
wpm	WAOPayment

2.7 Calculate WAO outputs