Argument editor

Verwer, Kamiel

Award date:
2002

TECHNISCHE UNIVERSITEIT EINDHOVEN

Faculteit Wiskunde en Informatica

AFSTUDEERVERSLAG

Argument Editor

door

Kamiel Verwer

Afstudeerdocent: Prof. Dr. P.M.E. de Bra

Augustus 2002

Indeling

1. Inleiding

Dit is een afstudeerscriptie voor de studie Technische Informatica. Het is niet het resultaat van een theoretisch onderzoek, maar de verslaglegging met betrekking tot het project "Argument Editor". Ik heb aan de Katholieke Universiteit Brabant dit programma ontwikkeld, dat ervoor dient om (rechten)studenten beter te leren redeneren. Met behulp van test-sessies, waar studenten een prototype kregen te zien dat ze mochten beoordelen, heb ik een dialoog-model samengesteld voor de manier waarop het programma de studenten leert redeneren. Dit model heb ik uitgewerkt tot een programma, dat voor medewerkers en studenten van de Universiteit van Tilburg via het internet beschikbaar is.
Toch is deze scriptie zelfstandig te lezen: er wordt in uiteengezet hoe we tot een bepaalde dialoog met de gebruiker komen, en waarom deze aanpak nuttig is voor het ontwikkelen van didactische programmatuur en voor de specifieke vaardigheden van het leren redeneren.
Het deelgebied van mijn afstuderen is User Interfaces, en dat betekent dat ik me voornamelijk bezighoud met die dialoog tussen programma en gebruiker en haar representatie op het scherm.
In deze scriptie beschrijf ik de architectuur van het programma "Argument Editor". In hoofdstuk 1 beschrijf ik de opdracht in haar context. Dat wil zeggen: haar voorgeschiedenis binnen de Universiteit van Tilburg en de plaats die het inneemt tussen andere programma's op de markt. Daarna ontwikkel ik de logische concepten die nodig zijn voor deze doelstellingen. In het bijzonder worden daar leermomenten geformuleerd en gekoppeld aan propositielogische principes (hoofdstuk 2). Hoofdstuk 3 is een globale beschrijving van de meest algemene ontwerpbeslissingen. Er wordt beargumenteerd hoe het systeem in abstracte componenten wordt ingedeeld, en eisen aan het systeem worden geformaliseerd. In hoofdstuk 4 wordt vervolgens een beschrijving gegeven van de architectuur. Hierbij worden de leermomenten uit hoofdstuk 2 betrokken, en aan de hand van User Interface principes uitgewerkt tot een dynamisch dialoogmodel dat zich concentreert op toestandsveranderingen van het systeem en een statisch model dat de structuur van het systeem beschrijft op verschillende abstractie-niveau's. Hoofdstuk 5 beschrijft de implementatie van de Argument Editor op een globaal niveau. Er wordt aan de hand van literatuur verantwoord waarom het Interface het uiterlijk heeft gekregen zoals ze dat heeft.
Voorts zijn er een aantal bijlagen: er is een rapportage van de evaluatie van het aan studenten voorgelegde prototype, en er is een lijst van definities van de gebruikte termen.
Ik dank voor dit project Ivana Ivkovic, die de teksten zorgvuldig heeft geanalyseerd, Bert van Roermund en David Janssens voor hun nuttige adviezen, en de studenten die meegeholpen hebben prototypes van het programma te testen. Verder gaat mijn dank uit naar de organisatie van het symposium "ons product is leren" dat op 27 mei 2002 aan de KUB heeft plaatsgevonden, en waar ik ook mijn programma heb kunnen presenteren aan belangstellenden. Ten slotte bedank ik Paul De Bra voor de begeleiding.

2. De opdracht

Heel in het kort zou de opdracht als volgt geformuleerd kunnen worden:

"Bouw een programma om rechtenstudenten te leren argumenteren."

Argumenteren is een belangrijke vaardigheid, die zich nuttig maakt in vrijwel alle disciplines. Er worden geen aparte cursussen "Argumenteren" gegeven in de standaard curricula van de Nederlandse universiteiten. Argumentatieve vaardigheden moeten vaak in de marge van andere vakken worden bijgebracht. Dat is op zich niet erg, aangezien er op dit gebied veel door zelfstudie is te bereiken.
Om studenten te helpen met het structureel analyseren van teksten, willen we een computerprogramma ontwikkelen. Dit programma moet gevoed worden met een analyse die de docent maakt van de tekst, en in staat zijn om studenten nuttig commentaar te geven als ze pregnante logische redeneerfouten maken, en eventueel als ze fijnzinnigheden over het hoofd zien. De studenten moeten na een korte instructie zelfstandig met het programma kunnen werken. Dat betekent dat ze de teksten die het programma aanbiedt (of die de studenten in het kader van een college reeds op papier in bezit hebben) kunnen analyseren in termen van basale logica. Als ze deze analyse hebben gemaakt (bijvoorbeeld op kladpapier), moeten ze in staat zijn haar te toetsen met behulp van het programma. Het programma moet de mogelijkheid bieden om de onderbouwingen van de student van commentaar te voorzien. Daarbij hoeft de student geen inhoudelijke uitleg te krijgen (bijvoorbeeld over de principes die hij verkeerd interpreteert), maar moet hij uitleg krijgen over *logische* fouten die hij heeft gemaakt (bij voorbeeld als de student een materiële implicatie verkeerd heeft begrepen).
Het programma moet dit met verschillende redeneringen kunnen doen, die op een voor de studenten aantrekkelijke manier worden gepresenteerd.

2.1 Geschiedenis

Deze paragraaf beschrijft de ontwikkeling van programma's zoals de Argument Editor zoals die überhaupt heeft plaatsgevonden (2.1.1), en het ontwikkelingstraject van de Argument Editor als onderdeel van het project Leren op maat II, aan de Universiteit van Tilburg zelf (2.1.2).

2.1.1 Overige programma's op de markt

Via het internet is toegang te krijgen tot een aantal programma's die logisch redeneren willen ondersteunen. Het bekendste voorbeeld is het programma van Jon Barwise en John Etchemendy[1]. Zij ontdekten een incongruentie tussen de manier waarop men met logica omgaat in software en de alledaagse manier van redeneren. "By 1988 we had collected and studied many examples of valid reasoning that did not fit within the confines of logic as it is normally understood." (Barwise1998, blz. 12). De standaard behandeling van logica is beperkter dan de logisch correcte manieren om te redeneren. "[...] modern logic [...] is based on the paradigm of mathematical communication: the rigorous proofs (or disproofs) by which one mathematician communicates results to another. This is why the theory, as important as it may be, yields neither a practical tool nor an accurate model of the real-life process of reasoning." ([Barwise1998], blz. 16). Hun idee was om logisch redeneren uit te breiden zodat alle correcte vormen van redeneren uit het alledaagse taalgebruik erin kunnen worden weergegeven. Daarom hebben ze de taal van de logica veranderd in driedimensionale figuren. De relaties van die figuren tot elkaar moeten logische relaties uitdrukken. Onze doelgroep, rechtenstudenten in de eerste fase van hun studie, zijn volgens ons niet gebaat bij een dergelijke aanpak: die wil vooral *terug* van een wiskundig-logische benadering naar een meer expressieve benadering. Maar studenten met weinig logica ervaring zullen juist de basisprincipes van de formele logica moeten leren kennen. Daarnaast is een presentatie door middel van bijvoorbeeld geometrische figuren verwarrend: de metaforen die we kiezen moeten zoveel mogelijk aansluiten bij de praktijk van de student.
Aan de andere kant van het spectrum, dus heel specifiek op de juridische praktijk gericht, bestaan er programma's die proberen specifiek 'legal reasoning' te ondersteunen. Voorbeelden daarvan zijn de programma's Dialaw van Arno Lodder en Argue! van Bart Verheij (zie [Lodder1999]). Deze systemen gaan uit van de praktijk van het redeneren in een juridische omgeving. Ze dienen ter ondersteuning van advocaten "by mediating the process in which they draft and generate arguments: the system can administer and supervise the argument processs by keeping track of the reasons adduced and the conclusions drawn, and by checking whether the users of the system obey the pertaining rules of argument, e.g. those related to the division of burden of proof." ([Lodder1999], blz. 1). Zij baseren zich op onderzoek naar *Legal Logic*. Strategieën die in het juridisch redeneren een rol spelen proberen ze om te zetten in logica. Het gaat om het in kaart brengen van uitzonderingen, het afwegen van conflicterende argumenten, de toepasbaarheid van regels, en de strategie van

[1] zie [Barwise1998].

opdeling van de bewijslast. Hun stelling is dat "legal logic has more to offer that is readily useful for law school than classical logic, since the argument form that can be analysed by legal logic, occur regularly in actual legal argument." (blz. 3). Daarnaast zijn er programma's die niet zozeer zelf deelnemen in de argumentatie, maar de argumentatie tussen studenten structureren. Een voorbeeld daarvan is ArgueTrack van Anders Bouwer[2].
Aangezien ons primaire doel het aanleren van *logisch redeneren* is, willen we niet teveel toegeven aan de concrete argumenteer-situatie uit de juridische praktijk. Door studenten het idee te geven dat *zij* zich moeten aanpassen aan het programma, leren we op een elementair niveau de omgang met logische structuren aan[3] en daarmee de vaardigheid om argumenten op waarde te schatten. Een systeem dat de specifieke nuances van juridisch redeneren wil aanleren, kan *daarna* worden aangeboden.
Onze ervaring is dus dat een programma dat op een aantrekkelijke en intuïtieve manier de omgang met de basale logische structuur van argumentaties aan kan leren, een behoefte is van het onderwijs waarin nog niet echt wordt voorzien. Niet door de eerste soort programma's omdat die de aandacht afleiden van de argumentatieve aspecten, en niet door de tweede soort, omdat die de aandacht afleiden van de basale logische structuur van argumentaties. Het doel van de Argument Editor is om een soort middenweg tussen de twee voornoemde programma's te vinden.

2.1.2 Ontwikkeling binnen de Universiteit van Tilburg zelf: stage "COO: leren redeneren met behulp van het internet"

In de laatste maanden van 1999 heb ik een stage gelopen aan de Universiteit van Tilburg. In het kader van het project "leren op maat II" heb ik gewerkt aan een programma om rechten-studenten te leren redeneren. De belangrijkste doelstelling van de stage was om de computer op zodanige wijze in te zetten in het onderwijs, dat hij een meerwaarde heeft boven het 'schoolbordperspectief'.
In de applicaties waarmee de studenten gewend zijn te werken, wordt de computer gebruikt als een schoolbord: de studenten krijgen informatie aangeboden en verwerken die passief. De vaardigheid van het leren redeneren veronderstelt een actieve instelling van de student. Om leren om te gaan met logische structuren moet je zelf in staat zijn om die structuren op elkaar te betrekken, ze kritisch te analyseren. Daarom was het project *leren op maat II* en in het bijzonder mijn deelproject COO

---

[2] "Preliminary evaluation with ten postgraduate students and staff members has shown that ArgueTrack allows the interactive construction of arguments, with possibilities to support or clarify them, ask for clarification, express uncertainty and disagreement. Diagrams were automatically created using heuristic knowledge acquisition rules which combine the propositional content of the discussion with information from the dialogue model. The quality of the resulting dialogue text is adequate, but the diagrams resulting from the evaluation sessions were not very good." ([Bouwer1999], blz. 7). Het valt op dat er geëvalueerd is met *postgraduate* studenten voor wie een formele presentatie van een dialoogmodel mogelijk is, terwijl ons doel juist is om de kunst van het logisch redeneren aan te leren met zo min mogelijk overhead, zo min mogelijk nieuwe termen. We concentreren ons daarom op de *propositional content* en bekommeren ons niet om een specifiek dialoog-model.
[3] Uit de testsessies bleek dat de studenten meer leren van de tekst die ze analyseren als ze gedwongen worden om in bepaalde structuren te denken. Het gaat er ons om, deze structuren eenvoudig te houden, zoals in de klassieke propositie-logica het geval is.

(Computer Ondersteund Onderwijs) in het leven geroepen. De werkzaamheid van de stage blijkt uit de toenmalige probleemstelling:

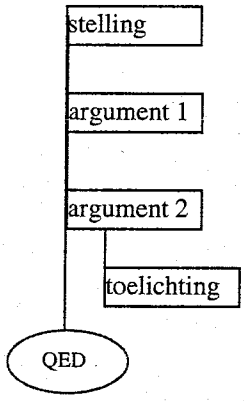
> "De stage houdt in het ontwikkelen van de argumenteren-applicatie. Rechtenstudenten die het vak rechtsfilosofie A volgen hebben meestal weinige vaardigheden in het redeneren. Zij krijgen een tekst met rechtsfilosofische argumentatie[4] die ze moeten analyseren en in een schema onderbrengen De argumenteren-applicatie waarvan ze gebruik maken bevat een vooraf uitgewerkte versie van zo'n analyse waaraan ze hun eigen analyse op verschillende manieren kunnen toetsen. Het belangrijkste leermoment is het omgaan met argumentatie-structuren en vooral het verwerven van inzicht in verschillende niveau's van de tekst. Het algemene doel van de stage is concrete taalelementen rangschikken op een manier die recht doet *aan retorica en de onderliggende logica.*"

Het ging in de stage dus vooral om het getrouw volgen van de 'structuur' van de tekst. Daarvoor heb ik gebruik gemaakt van een argumentatie-model van de gewone taal, gebaseerd op het model van Stephen Toulmin[5]. Het op de retorica georiënteerde model van de stage zet ik hier kort uiteen. Tevens zal ik het van commentaar voorzien. Dit dient er in de eerste plaats voor om de ontwikkelingsgang van het project in kaart te brengen, en verder bestaat er een mogelijkheid om in een later stadium extra semantiek toe te voegen aan de analyses van de Argument Editor (dat wil zeggen: de retorische aspecten, de wijze waarop argumenten *gebruikt* worden, bij de structuur te betrekken).
Een redenering, in het argumentatie-model van de stage, is een aaneenschakeling van elementen die door links aan elkaar zijn gekoppeld. Een element heeft een bepaald type (bijvoorbeeld 'stelling' of 'toelichting' of 'argument')[6]. Dit type is in een redenering voor het grootste deel afhankelijk van een context, en hoort daarom bij een link[7]. Voor zover een element intrinsiek een type heeft, kan het volgende onderscheid worden gemaakt: elk element dat een uitspraak doet over feitelijke standen van zaken in de wereld is een feit en al het andere is geen feit.
Elementen kunnen op twee manieren door links met elkaar worden verbonden: op hetzelfde niveau of op een dieper niveau. In de argumentatietheorie[8] is het belangrijkste onderscheid dat tussen onderschikkende elementen en nevenschikkende

---

[4] Het betreft de tekst van Lon Fuller, [Fuller1949]. Dit project baseert zich in beginsel ook op deze tekst, al is de opzet volledig generiek.
[5] Het boek "Argumentatieleer" ([vanEemeren1981]) is mij daarbij van nut geweest. In het afstudeerproject heb ik niet teruggegrepen naar analyses van de structuur van retoriek. In plaats van zo'n "top-down" benadering van de logische structuur van een tekst, heb ik voor een "bottom-up" benadering gekozen. De teksten worden door de docent geherformuleerd in een degelijk en simpel logisch kader. De reden daarvoor is dat het leren omgaan met basale propositie-logica een voorwaarde is om met retorische fijnzinnigheden kennis te maken. Aan deze voorwaarde bleek bij te weinig studenten voldaan, dus heeft ze de prioriteit. De keuze voor mijn benadering onderbouw ik verder in hoofdstuk 3.
[6] In het huidige project is dit onderscheid niet primair. Studenten moeten in de eerste plaats leren de argumentatie-structuur als een propositie-logische structuur te zien, en dus door de retorische gebruikswijze heen te kijken. Primair is nu het afleren om te snelle conclusies te trekken en het aanleren om de benodigde aannames te expliciteren. Zie verderop de didactische eisen Di-1 en Di-2 in § 3.1.
[7] Als we extra semantische informatie gaan toevoegen aan de Argument Editor, dan zou die gaan toebehoren aan een *formule*.
[8] zie bv. [vanEemeren1984].

elementen. Dit onderscheid komt hier tot uitdrukking door de twee soorten links: links op een gelijk niveau voor nevenschikkende argumenten, en links op een dieper niveau voor onderschikkende argumenten.
Een redenering krijgt als gevolg van dit onderscheid een boomstructuur die er bijvoorbeeld zo uit zou kunnen zien:



Deze boomstructuur is in principe geschikt voor alle soorten redeneringen in de gewone taal. Om meer logische 'hardheid' in de begripsanalyse te krijgen is het goed om Aristotelische syllogismen in deze structuur te representeren[9]. Een syllogisme heeft, zoals bekend, de vorm major-minor-conclusie. Elke van de onderdelen kan verder worden onderbouwd, eveneens syllogistisch. De retorische concepten onderschikking en nevenschikking zijn dan aanwezig en dus kan in principe een redenering worden geanalyseerd in deze termen. Om echter de flexibiliteit van het systeem in de zin van overeenkomst met gewone taal en in de zin van aansluitend op de gedachtewereld van de studenten te handhaven, is een compromis gesloten. De stap van retorische analyse (die de volgorde van een betoog aanhoudt[10]) naar syllogistische analyse is tweeledig:

1. Elementen kunnen worden toegevoegd, de "verzwegen vooronderstellingen", om het syllogisme compleet te maken[11];
2. Elementen kunnen binnen hun eigen niveau worden verschoven om de volgorde major-minor-conclusie te verkrijgen[12].

---

[9] Deze manier bleek niet voldoende om studenten de logische 'hardheid' van redeneringen te laten ontdekken.

[10] Om de structuur van de tekst nauwgezet te volgen, heb ik in het stage-project bij de analyse van redeneringen de volgorde van de tekst aangehouden. De *logica* wordt daardoor echter vaak op de achtergrond gesteld. De volgorde van de tekst is vaak niet de logische volgorde: veel retorische middelen bestaan er precies is vooruit te grijpen op argumenten of de conclusie vooraf te vermelden. In dit project zal de tekstuele volgorde niet belangrijk zijn: de student moet in zijn analyse abstractie kunnen maken van de tekst en dus ook van haar volgorde. Alleen zo kan hij zich bewust zijn van de *analogie* tussen tekst en structuur (dit zal verderop eis Di-3 heten), en de tekst niet met de structuur *identificeren*.

[11] Dit zal bij ons terugkomen als de didactische eis Di-1 en Di-2. De student moet in zijn analyse leren om zijn reconstructie logisch kloppend te krijgen. Daarvoor moet de docent in de aangeboden analyses de verzwegen vooronderstellingen nauwkeurig formuleren. Aan de student is de taak op die vooronderstelling op de juiste plek in de redenering in te passen.

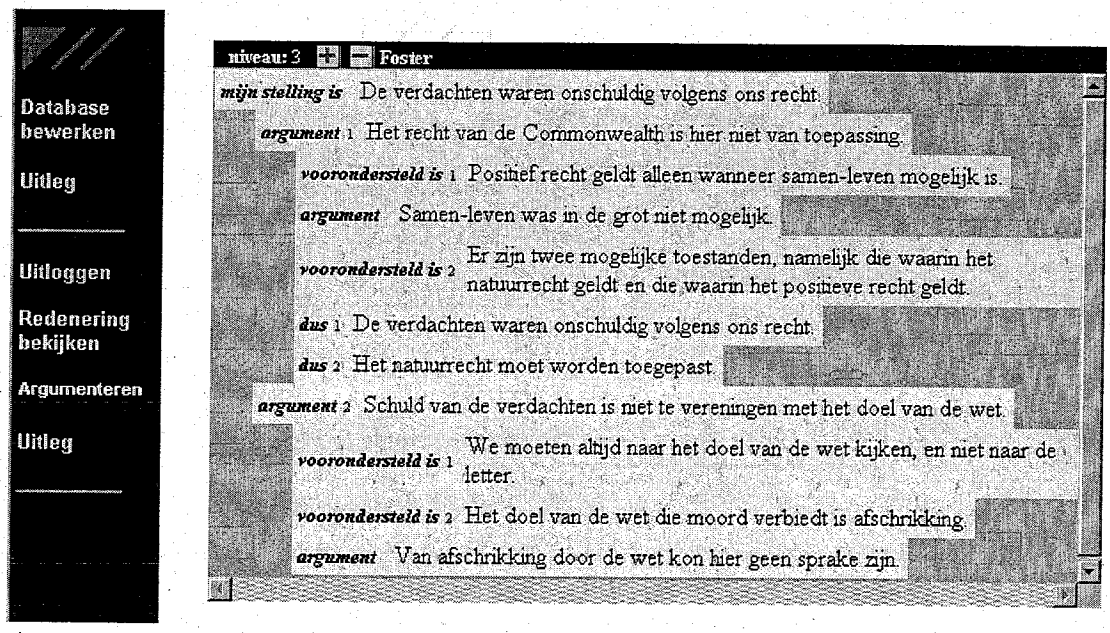
[12] Dit zal bij ons terugkomen als de didactische eis Di-2. De student moet leren om de aannames die hij maakt te expliciteren. Daarvoor is echter de *volgorde* van de tekst niet belangrijk. De student moet zelf de analyse maken waarin er niet zozeer sprake is van volgorde, maar van een boomstructuur (zie hoofdstuk 4).

De student moet, om te kunnen omgaan met het programma, de volgende concepten beheersen[13]:

1. Niveau van een redenering. Het niveau waarop een element in een redenering is aangebracht geeft zijn relevantie voor de oorspronkelijke stelling aan. Studenten hebben de neiging om in teksten het merendeel van de zinnen voor belangrijk aan te zien (en die allemaal te markeren) waardoor ze het overzicht kwijtraken. Het beheersen van het concept *niveau* zorgt er tevens voor dat het onderscheid tussen onderschikking en nevenschikking helder wordt;
2. Gewone taal-connectieven. Voor de typering van de koppelingen is gekozen voor connectieven uit de gewone spreektaal, om geen onnodige begrippen aan te hoeven leren. Het gaat om woorden als 'dus', 'want', zie', 'voorondersteld is', 'verondersteld is', 'stel dat', enz.

De implementatie van het stage-project

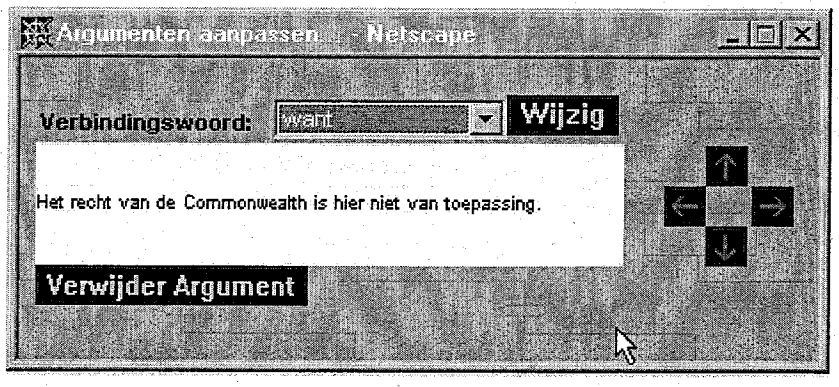
Het stage-project is geïmplementeerd door een Microsoft Access- database die via ASP-routines werd aangesproken. Ik ga hier niet in op de details van de implementatie. Een impressie van hoe het belangrijkste venster van het programma eruit zag moet voldoende zijn om vanuit het perspectief van *User Interfaces* de historische ontwikkeling van de Argument Editor te overzien:



De student kon de redenering op een gewenst niveau bekijken, door op "+" en "-" te drukken. In een ander venster moest hij de gegevens zoals in bovenstaande illustratie te zien zijn zelf aanvoeren. Hij moest het 'connectief' kiezen en een zin, beiden uit een meerkeuze-lijst van verschillende opties:

[13] Voor het werken met de Argument Editor moet de student de basale propositie-logische principes in ieder geval impliciet beheersen (hij hoeft ze niet te kunnen benoemen). De Argument Editor vooronderstelt *geen* intuïtie van alledaagse taalconcepten, maar leert de student redeneren met een *minimaal* idioom om de logische relatie tussen argumenten te benoemen. De term die de student goed zal moeten kennen zijn 'argument' (of 'onderbouwing'), en 'structuur' (zie de lijst van begrippen).

Als de student een argument in de context wil verplaatsen, doet hij dat via deze Interface:



Door op de pijlen te klikken kan de student een argument in de context verplaatsen. Daarmee wordt duidelijk dat het heel wat uitmaakt *waar* een argument in de redenering staat (als nevenschikking of als onderschikking).
In het hoofdstuk over implementatie (hoofdstuk 6) kom ik terug op het User Interface-technieken die ik voor de Argument Editor heb gebruikt, en beschrijf ik op welke punten ik dingen van het stage-project heb kunnen overnemen.

Het uitbreidingsperspectief van het stage-project

Ik bespreek nu kort wat de status was na afloop van het stage-project. Dit om de huidige context (zie 2.2) inzichtelijker te maken vanuit het perspectief van haar voorgeschiedenis. De aandachtspunten aan het eind van de stage zijn hier van commentaar voorzien; in 2.2 zijn ze meer structureel uitgewerkt. Aan het eind van het stage-project waren er de volgende opties voor uitbreiding:

1. *Meer rechtsfilosofische en andere teksten moeten kunnen worden geanalyseerd en aan studenten aangeboden;*
   De opzet van de Argument Editor is volledig generiek. Om toch een vertrouwde omgeving te maken voor studenten die met één bepaald vak bezig zijn en daarvoor het programma als hulpmiddel gebruiken, kan er een meer specifieke versie uit worden afgeleid, waar de metaforen op het scherm bijvoorbeeld

betrekking hebben op dat vak, of op die casus. De werkrichting is in die zin omgekeerd, dat ik nu ben uitgegaan van een generieke module, en er op Interface-technisch niveau kan worden toegespitst op één bepaalde casus;

2. *Het programma kan interactiever worden gemaakt in de zin dat studenten zelf voor een groot deel kunnen bepalen welke argumentatie-analyse er verschijnt, of in die zin dat het programma die argumentatie laat oefenen waarmee de student moeite heeft;*
   De interactiviteit van de Argument Editor is gewaarborgd doordat studenten geacht worden de teksten die ze krijgen aangeboden zelf eerst volledig te analyseren. Dit heeft ermee te maken dat de logische structuur die we aan de teksten 'opleggen' nu niet meer zo natuurgetrouw is: de student zal moeite moeten doen om het verband te zien tussen tekst en structuur, maar daar door het programma wel steeds toe gedwongen worden. In zekere zin oefent de student sowieso al bij het zelfstandig bestuderen van de tekst waarmee hij moeite heeft.

2.2 Huidige context

Het stage-project bleek niet echt bruikbaar voor een grote groep studenten, omdat er naast het genoemde probleem dat studenten hebben met basale logica teveel mondelinge uitleg bij nodig was. De student kon er niet zelf mee aan de slag. De context heeft zich dus iets veranderd: de nadruk is komen te liggen op de logische structuur 'in' de redenering.
We hebben de redeneringen uit [Fuller1949] nogmaals geanalyseerd[14], nu met meer nadruk op de eenvoudige propositie-logica. Het resultaat van deze analyse was vooral dat het mogelijk bleek om de teksten in een bepaalde boomstructuur te formuleren. Het programma zal de studenten vertrouwd maken met deze boomstructuur en haar analogie met de oorspronkelijke tekst. In hoofdstuk 3, waar de eisen aan het programma uiteengezet worden, zullen de eisen waaraan de logische component moet voldoen worden afgeleid uit de eisen van de didactiek. Hier wordt aangegeven dat deze benadering van didactiek en logica vruchtbaar is, omdat de teksten in de vorm van een boomstructuur kunnen worden weergegeven. We kunnen, aan de hand van de teksten die we geanalyseerd hebben, de studenten dus leren wat we hen willen leren.

[14] Ivana Ivkovic heeft de teksten uitgeplozen, er er op een nette manier logische structuren in ontdekt.

3. De eisen aan logica, didactiek en Interface

Er zijn drie abstracte componenten te onderscheiden in de opdracht: een component die de logica voor zijn rekening neemt, een component die de didactiek ondersteunt en een component die de didactische dialoog omzet in concrete representatie op het scherm. Deze drie abstracte componenten hoeven niet noodzakelijk te leiden tot drie gescheiden componenten in de implementatie, maar ze vormen een perspectief, een *view* op die implementatie.
In dit hoofdstuk beschrijf ik de ontwikkeling die heeft geleid tot deze driedeling, en identificeer ik de eisen die worden gesteld aan deze drie componenten. Eerst zijn de didactische eisen uitgewerkt, omdat die het meest fundamenteel zijn. De eisen die aan de logica-component worden gesteld zijn daaruit af te leiden, en de eisen aan het Interface voor een deel ook. De eisen aan het Interface vinden hun oorsprong daarnaast in boeken over User Interfaces en het resultaat van testsessies[15]. De eisen die betrekking hebben op algemene zaken als de stabiliteit of de uitbreidbaarheid van het programma, heb ik niet expliciet gesteld. In plaats daarvan heb ik de opdracht zodanig geformuleerd (zie hoofdstuk 2.3) dat er wat dat betreft geen ambiguïteiten meer in de beschrijving staan.

3.1 Didactiek: wat zijn de leereenheden, of leermomenten die een rol spelen bij het leren redeneren?

Voor het bepalen van de eisen die aan de didactiek worden gesteld heb ik me niet gebaseerd op literatuur over didactiek. Een reden daarvoor is dat didactiek over het algemeen betekent de kunde van het leren aan kinderen, terwijl het bij de doelgroep studenten in de eerste plaats van belang is ons op de *inhoud*, namelijk het redeneren, te concenteren. Het is dus contraproductief om deze inhoud op een standaard (schoolse) manier te presenteren. Hoe moet hij dan worden gepresenteerd? Hiervoor ben ik voornamelijk uitgegaan van de onderwijservaring van de docenten die de cursus geven waarbij het programma wordt ingezet. Dat betekent dat ik in een redenering die aspecten de meeste relevantie toeken die in de colleges ook belicht worden. Op deze manier laten zich een aantal eisen onderscheiden. De status van deze eisen is dat er in de dialoog van de Argument Editor momenten aan te wijzen moeten zijn die duidelijk al deze leermomenten representeren.

De didactische eisen zijn onder te verdelen in twee groepen: de *minimale* eisen die gesteld worden aan het programma zoals dat door alle studenten moet kunnen worden gebruikt. De omgang met logische structuren is hier *informeel*. Daarnaast zijn er *extra* eisen die betrekking hebben op de uitbreiding van het programma voor studenten die *expliciet* met de logische structuren kunnen omgaan door formule-manipulatie. De minimale eisen [Di-1] tot en met [Di-3] volgen uit de opdracht. Ze hebben de hoogste prioriteit De eisen [Di-4] tot en met [Di-6] hebben betrekking op de formule-

[15] Boeken: vooral [Preece1994]. Testsessies: zie bijlage 1.

manipulatie, die niet per se door de student wordt uitgevoerd, maar achter de schermen kan verdwijnen.
De eisen [Di-7] en [Di-8] komen respectievelijk voort uit [Lodder1999] en mijn stage-project (zie § 2.1.2). Ze hebben betrekking op de docent, die in de docenten-editor formules invoert en daarna deze mogelijkheden ter controle moet hebben (zie § 4.3.F).

[Di-1] De student moet afleren om te snelle conclusies uit argumenten te trekken, dus leren de nodige tussenstappen te maken en te expliciteren.

[Di-2] De student moet leren om voor het trekken van een conclusie de aannames die hij nodig heeft te expliciteren.

[Di-3] De student moet een overzicht hebben van de logica waarmee hij werkt, en zich van de analogie van de tekst met de structuur bewust zijn.

[Di-4] De student moet zich bewust zijn van de *contextuele status* van argumenten; bijvoorbeeld 'nevenschikking' of 'onderschikking'.

[Di-5] De student moet zich bewust zijn van de *argumentatieve status* van de argumenten; bijvoorbeeld 'feit' of 'norm'.

[Di-6] De student moet logische formules (zie bijlage 2 voor definitie) kunnen laten representeren, toevoegen aan een logica, aanpassen, en verwijderen uit een logica.

[Di-7] De docent moet logische formules kunnen controleren op syntactische geldigheid.

[Di-8] De docent moet twee of meer logische formules met elkaar op equivalentie en wederzijdse implicatie kunnen controleren.

3.2 Logica: welke logische bewerkingen moet het programma kunnen uitvoeren?

Het programma zal een kern bevatten die 'om moet kunnen gaan met logica's'. Hoe die component er precies uitziet, is beschreven in hoofdstuk 4. De eisen die aan de logica worden gesteld volgen uit de didactische eisen uit 3.1. Er is bij elke logica-eis aangegeven welke didactiek-eis(en) en welke logica-eisen die logica-eis heeft opgeleverd. In het geval dat een logica-eis uit een aantal andere logica-eisen is afgeleid, kan het zijn dat die de andere eisen vervangt. Ook dat is aangegeven; de oorspronkelijke eisen zijn wel gehandhaafd, omdat in sommige contexten deze eisen relevant zijn als deel-eisen van de vervangende eis, en voor de duidelijkheid van dit document. De logica-eisen worden dus stap voor stap uit de didactische eisen afgeleid.

[Lo-1] tot en met [Lo-4] zijn vertalingen van de didactische eisen [Di-1] tot en met [Di-3]. [Lo5] tot en met [Lo-7] zijn vertalingen van de didactische eisen [Di-4] tot en met [Di-6]. De laatste logische eis volgt uit [Di-7] en [Di-8].

[Lo-1] De logica-component moet ten minste twee verschillende redeneringen tegelijkertijd kunnen representeren, die ze met elkaar kan vergelijken. Volgt uit [Di-1] en [Di-2].

[Lo-2] De logica-component moet redeneringen met tussenstappen kunnen representeren, en herkennen wanneer die tussenstappen worden overgeslagen en welke. Volgt uit [Di-1].

[Lo-3] De logica-component moet redeneringen kunnen representeren waarin conclusies een rol spelen die uit *meerdere* aannames volgen, en kunnen herkennen wanneer er teveel of te weinig aannames zijn gemaakt en welke. Volgt uit [Di-2].

[Lo-4] De logica-component moet ten minste twee redeneringen tegelijkertijd kunnen representeren die de vorm hebben van een *boom* (zie lijst van definities). Ze moet de verschillen tussen die bomen kunnen berekenen.
Vervangt [Lo-1, Lo-2 en Lo-3]

[Lo-5] De logica-component moet logische formules (zie lijst van definities) kunnen representeren, toevoegen aan een logica, aanpassen en verwijderen uit een logica. Volgt uit [Di-4].

[Lo-6] De logica-component moet logische formules kunnen controleren op syntactische geldigheid. Volgt uit [Di-5].

[Lo-7] De logica-component moet twee of meer logische formules met elkaar op equivalentie en wederzijdse implicatie kunnen controleren. Volgt uit [Di-6].

[Lo-8] De logica-component moet de mogelijkheid bieden om extra semantiek aan argumenten toe te voegen; bijvoorbeeld de argumentatieve of contextuele status van het argument. Volgt uit [Di-7] en [Di-8].

3.3 *Interface: waar moet het uiterlijk van het programma in ieder geval aan voldoen?*

Het User Interface van het programma moet zodanig worden opgezet dat rechtenstudenten zich erin thuisvoelen; het moet dus overeenstemmen met internet-pagina's en programma's die ze kennen van hun studie (en die ze associëren met "de computer als studie-hulpmiddel"). De programma's die studenten (en met name de primaire doelgroep van de Argument Editor, rechtenstudenten in de eerste fase van hun studie op de Universiteit van Tilburg) gebruiken zijn:

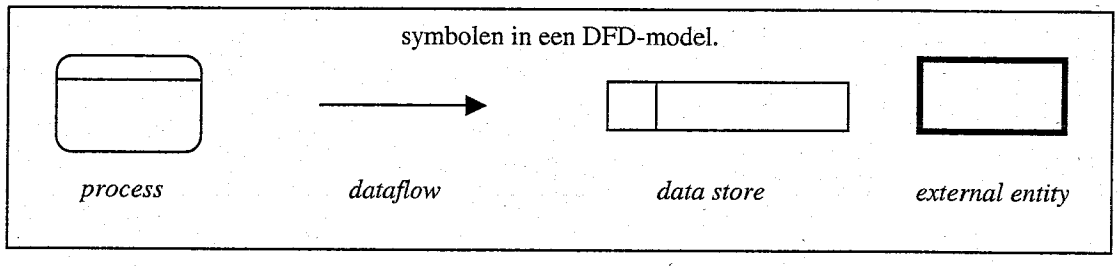
- Blackboard CourseInfo, een programma om collegestof interactief op het internet te publiceren. http://www.blackboard.com/
- Practicum Rechtsinformatica, een programma om studenten te helpen met praktisch computergebruik. http://rechten.kub.nl/prak/menu/index.htm
- De electronische studiegids. http://studiegids.kub.nl/
- De homepage van de Universiteit van Tilburg. http://www.kub.nl/

Het uiterlijk van de Argument Editor is afgestemd op het uiterlijk van deze pagina's. Naast computer-programma's zijn er nog andere elementen uit de leefwereld van studenten die kunnen terugkomen in het programma om het vertrouwder en dus aantrekkelijker te maken. Te denken valt aan metaforen als een toga, een stapel papieren, of iets dergelijks. Bij de keuze voor het gebruik van metaforen is de overweging wat het teweeg brengt bij de gebruiker van groot belang: "An important consideration when searching for Interface metaphors is the [...] subjective and emotional impact that different graphical representations can convey" ([Preece1994], blz. 149).
Bij het ontwikkelen van de grafische Interface, heb ik een holistische benadering gekozen. Daar geldt dat "[...] decisions about the way an Interface should look are made in relation to how this will be physically communicated to users." ([Preece1994], blz. 452). Het is belangrijk om een algemene metafoor te hebben voor het geheel van het Interface. Het Interface van de Argument Editor is een Interface als *tool* en een Interface als *media.* ([Preece1994, blz. 461). Dat betekent dat de student het programma moet kunnen gebruiken voor taken die hij *onafhankelijk* van het programma kan identificeren: zo is het programma een werktuig of *tool* zoals een hamer en een beitel dat zijn. Omdat echter datgene wat met het tool wordt bewerkt ook in het Interface gerepresenteerd moet zijn, is voor het programma ook de metafoor van het Interface als *media* van toepassing: de student heeft in zijn leerproces een kijk op de didactische inhoud die *bemiddeld* is door het Interface.
Deze benadering van het Interface leidt tot een aantal vuistregels met betrekking tot de presentatie van informatie-elementen. Deze regels hebben echter niet meer het karakter van eisen, en daarom heb ik ze in het hoofdstuk over de implementatie van het Interface besproken (zie hoofdstuk 5).

4. Architectuur

Dit hoofdstuk is een beschrijving van de architectuur van de Argument Editor. Het is opgebouwd uit een algemeen deel, waar een globaal overzicht wordt gegeven van het systeem, en een specifiek deel, waarin de architectuur van verschillende onderdelen van het programma uiteen wordt gezet. Eerst zal ik echter beargumenteren waarom ik voor de gebruikte *Architecture Description Language* heb gekozen.

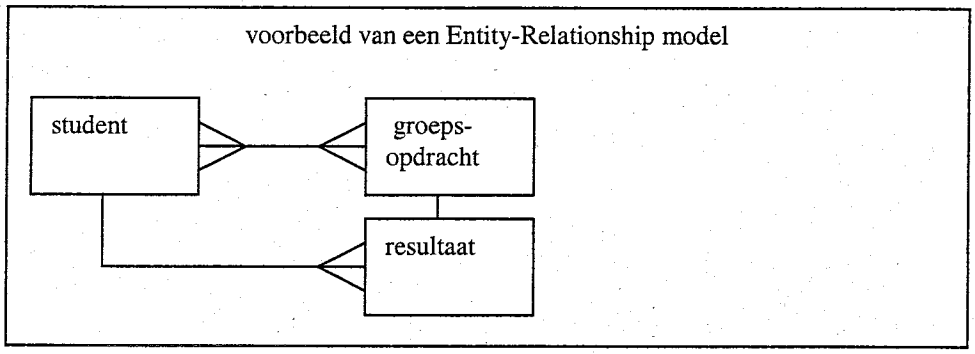
In het programma zijn in abstracto een aantal componenten te onderscheiden, die met elkaar samenwerken. Er is onmiddellijk een driedeling in een Interface-component die het *hoe* van de interactie met de gebruiker verzorgt, een didactische component die die interactie aanstuurt (dus het *wat* van die interactie verzorgt), en een logica-component die de manipulaties op de logica uitvoert. Voorts heb ik in mijn modellering gekozen voor het beschrijven van een database-component met functionaliteit voor toegang tot een database. De opbouw in deze vier componenten heeft het voordeel dat het een duidelijke band heeft met de praktijk: de beschrijving van de architectuur zal overeenkomen met de manier waarop de (eind)gebruiker over het systeem denkt. Daarnaast maakt deze opdeling de software flexibeler en gemakkelijker uitbreidbaar. Omdat de Interface-component als een client-side object zal worden geïmplementeerd, laten zich aanpassingen in het Interface (door bijvoorbeeld verandering van de hardware van de client) gemakkelijker doorvoeren.
Deze intuïtieve indeling in componenten kan met behulp van verschillende technieken worden gemodelleerd[16]. De twee bekendste zijn DFD (Data Flow Diagramming) en ERM (Entity Relationship Modelling). DFD modelleert het systeem vanuit een logisch perspectief; er is geen onderscheid tussen het conceptuele niveau en het interne niveau (zoals bij entity-relationship modellen). Een voordeel van DFD is dat het modelleren een logiceren is: uit een (intuïtief) fysisch model van het te beschrijven systeem wordt een logisch model afgeleid. Wat op het eerste gezicht verschillende processen zijn, is in een logisch model vaak beter als één proces te modelleren[17].



Een Entity-Relationship model beschrijft het systeem in termen van abstracte entiteiten: er wordt een onderscheid gemaakt tussen een conceptueel niveau en een intern niveau van beschrijven. Het is intuïtief, en heel flexibel: doordat het zich concentreert op de structuur van de data (het karakteriseert vooral de afhankelijkheden tussen de entiteiten), is de manier waarop programmatuur met de data omgaat niet afhankelijk van het modelleer-proces. Een ERM biedt een goede

[16] Zie: [Tudor1995], hoofdstuk 3 voor een beschrijving van verschillende gebruikelijke modelleer-technieken.
[17] Bijvoorbeeld: een logisch DFD-model zal een proces "handle order" bevatten, i.p.v. twee processen "validate order" en "execute order". Naar: [Tudor1995], blz. 45.

basis voor een (relationele) database; het vermijdt data redundantie door de data te modelleren bij de entiteiten, en zorgt voor een beter begrip van het probleem-domein. In een ERM kunnen de *cardinaliteiten* van relaties worden aangegeven (one-to-one, one-to-many, many-to-many), exclusiviteit (A kan een relatie hebben met òf entiteit B, òf entiteit C), en recursie (een entiteit heeft een relatie met zichzelf). Deze mogelijkheden zijn erg goed voor het beschrijven van een data-strcutuur, maar een ERM is moeilijker uit te breiden met dynamische aspecten dan een DFD.



Een modernere beschrijvingstaal is [UML] (Unified Modeling Language). Ik heb gekozen voor een beschrijving in UML. In UML worden verschillende *views* onderscheiden; ik werk alleen die *views* uit die verhelderend zijn voor de architectuur. De symbolen die in UML worden gebruikt komen veelal overeen met DFD of ERM; waar ze toch verschillen, heb ik dat aangegeven. De wijzigingen in de beschrijvingswijze die ik heb aangepast (zoals bijvoorbeeld het combineren met dynamische aspecten: naast dataflows zijn er in het model ook invocations) zal ik ook bij het model bespreken.

Deze opdeling in componenten gaat uit van een statische aanpak: de componenten zijn, hoewel in dit stadium nog abstract, verzamelingen routines die kunnen worden aangeboden aan andere routines. Ik denk dat er voor een adequate beschrijving van de architectuur van de dialoog met de gebruiker, of van het stuk software dat de dialoog verzorgt, er een dynamisch aspect aan het model moet worden toegevoegd. In dat model kunnen de paden worden aangegeven die worden gevolgd in het doorlopen van het programma, en op die manier laat zich een communicatief proces veel beter beschrijven. De top-laag van het programma is implementatie-technisch over het algemeen redelijk eenvoudig, en laat het dus gemakkelijk toe om het te beschrijven in termen van communicatieve processen tussen programma en gebruiker.
Een combinatie van een statisch model en een dynamisch model heeft dus twee functies: het is zowel een *plattegrond* van het programma, als een *stroomschema*. In de plattegrond wordt de logische samenhang van functionaliteiten beschreven, terwijl in het stroomschema de communicatieve samenhang van functionaliteiten wordt belicht. Ik heb in het algemene plaatje beide modellen gecombineerd, om een overzicht te kunnen geven van het gehele systeem.

De architectuur-schema's in dit hoofdstuk bevatten de volgende symbolen:


Component die als statische component beschreven wordt in hoofdstuk 4.2 (in DFD-termen: een process)

Component die als dynamische component (dialoog) beschreven wordt in hoofdstuk 4.3

Een data-bron (in DFD-termen: een data-store)

Een globale data-bron, beschikbaar voor alle componenten

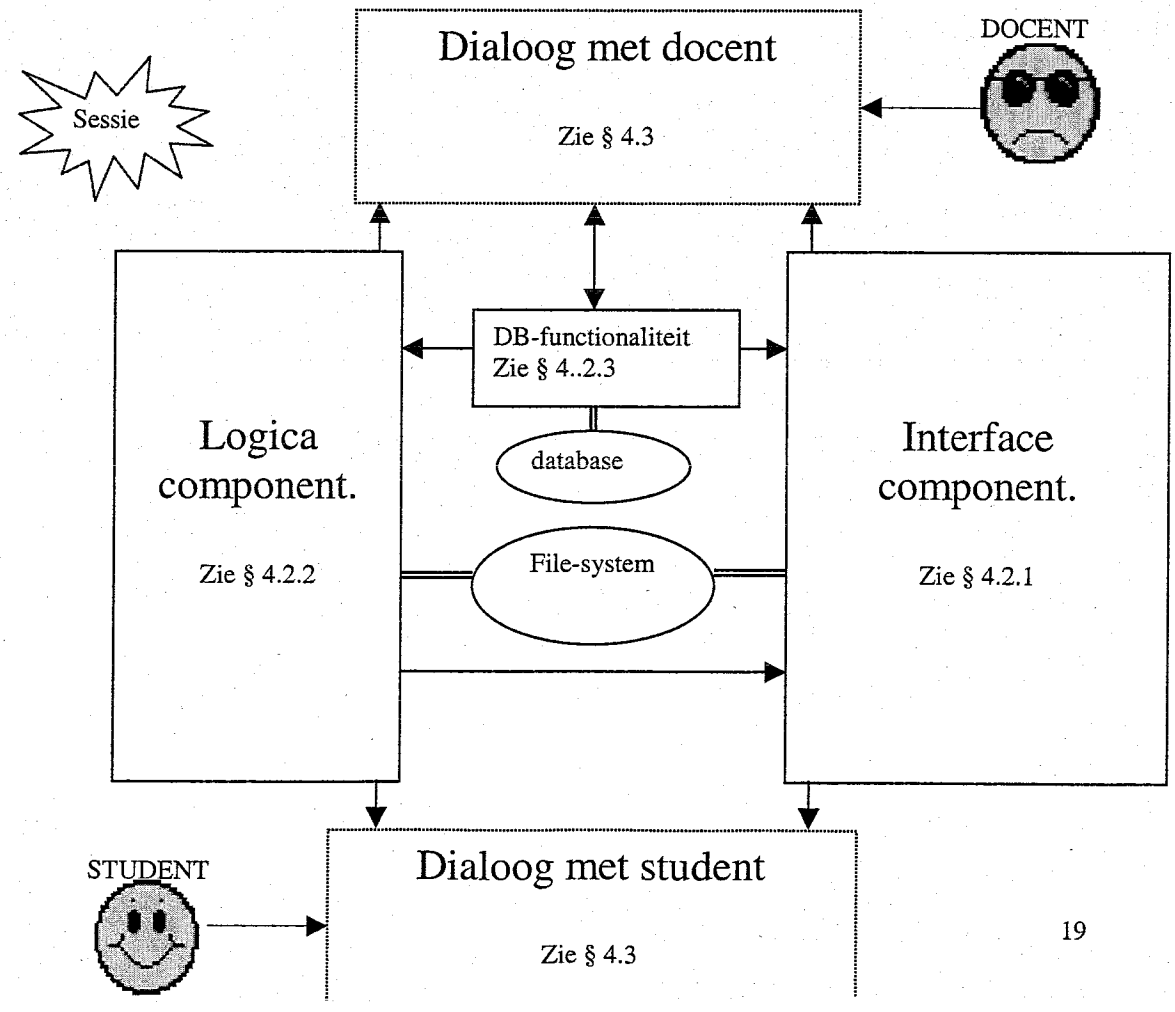
Datalink: verbinding met een data-bron (in DFD-termen: een dataflow)

Aanroep van functionaliteit uit dezelfde of een andere component

< var >Een parameter bij het aanroepen van een routine (in de gedetailleerde beschrijving)

4.1 Algemeen

Het programma kan voorgesteld worden zoals in onderstaande afbeelding, als een aantal componenten die met elkaar interageren. De componenten *logic*, *Interface*, en *database functionaliteit* hebben een statisch karakter: ze kunnen worden beschreven in termen van routines die aangeroepen kunnen worden met bepaalde parameters. De componenten *dialoog met docent* en *dialoog met student* hebben een dynamisch karakter: deze componenten moeten worden beschreven in termen van de *walkthrough* die de student respectievelijk docent erdoorheen maakt.



De *logica*-component wordt door de beide dialogen gebruikt, en door het Interface. Vooruitlopend op de implementatie van de logica-component als een *class*, kunnen we zeggen dat er twee instantiaties van zullen worden gemaakt: één voor de docent, die de correcte logica bevat, en één voor de student, die de logica bevat die de student aan het opbouwen is. Het Interface-component gebruikt de logica voor het weergeven van de logica op het scherm, waarbij ze moeten samenwerken. De logica kan omgekeerd niet direct afhankelijk zijn (zonder tussenkomst van de dialoog-componenten) van het Interface.

Het Interface-component wordt alleen door de dialogen aangeroepen, en is daarmee relatief onafhankelijk binnen het systeem. Deze component kan dan ook zelfstandig geïmplementeerd worden, waarbij er gebruik kan worden gemaakt van standaard *toolkits*.

De data-component *sessie* is globaal, dus algemeen beschikbaar voor alle componenten. Een sessie is de verzameling data die hoort bij één instantie van een rol (docent of student). Deze data representeert de positie van de gebruiker in het systeem.

4.2 Statische beschrijving

De componenten *logica, Interface,* en *database* kunnen in een statisch architectuurmodel worden beschreven. Zo'n beschrijving abstraheert van het temporele aspect, oftewel de manier waarop een component in de tijd functioneert. De dynamische beschrijving, in hoofdstuk 4.3, abstraheert juist van de statische structuur van de componenten, om het temporele aspect te benadrukken.

4.2.1 De logica-component

In dit hoofdstuk wordt de logica-component besproken. De belangrijkste ontwerp-beslissingen die hier een rol hebben gespeeld worden verantwoord. Dit hoofdstuk is de documentatie bij het interne deel van het programma. De manier waarop de logica wordt gepresenteerd aan de gebruiker wordt beschreven in hoofdstuk 4.3, waar een dialoogmodel wordt ontwikkeld, en in hoofdstuk 5, waar de grafische implementie van dit model wordt beschreven.

De logica-component is het stuk programma dat de logica verzorgt. Hij moet voldoen aan de eisen Lo-1 t/m Lo-8. Ik beschrijf de component in termen van een *class*, omdat dat een goed overzicht geeft[18].
De *class* logica is verbonden met één unieke logica, dus een unieke set formules. Op deze set formules kan een instantie van de *class* logica een aantal bewerkingen

[18] Uiteindelijk is de logica-component ook geïmplementeerd als een *class*, maar dat is niet de reden waarom hij in dit hoofdstuk als zodanig is beschreven.

uitvoeren. Ik beschrijf deze bewerkingen hier vanuit een functioneel perspectief: aan de hand van de functionele eisen die direct volgen uit de eisen Lo-1 t/m Lo-8.

Voordat ik de architectuur beschrijf, zal ik eerst de datastructuren die hier een rol spelen kort toelichten. Uit de Lo-eisen werd duidelijk dat de logica-component *formules* moet bevatten, die gemanipuleerd kunnen worden. Deze formules zijn een combinatie van symbolen. Zonder hier in te gaan op implementatie, kunnen we het datatype formule als volgt karakteriseren met een reguliere expressie:

$$for = \{\{(\}^{i}.for.oper.\{\sim\}^{0|1}.for.\{)\}^{i}\ \} \mid var$$

waar for is een formule;
var is een variabele;
oper is een logische operator, waarvoor geldt
`oper = { "&" | "|" | "=>"}`

De formules van een logica (een instantiatie van de component logica) laten zich dus beschrijven als

$$for^{*}$$

Ik beschrijf hier eerst de pre- en postcondities bij de functionaliteit waarvoor dat nuttig is, en geef daarna een DFD.

loop-detect

bepaalt of er een lus komt als <formule> aan <formules> wordt toegevoegd

pre: <formules> bevat geen lus & <formule> is een syntactisch correcte formule

post: (loop) = er ontstaat een lus als <formule> wordt toegevoegd aan <formules>

** door uit te gaan van *geen* lus voor de detectie, hoeft alleen de tak te worden bekeken waarin <formule> zou komen. Dit maakt het efficiënter (het is, technisch gesproken, een branch en bound conditie).

bepaal wortel van de boom

pre: <formules> zijn de formules van een logica

post: (wortel) is de variabele die de wortel in de boom is als die bestaat, anders *false*

** De wortel is de eindconclusie van waaruit de student moet reconstrueren.

controleer plaats in de logica van een formule

pre: <formule> is een syntactisch correcte formule

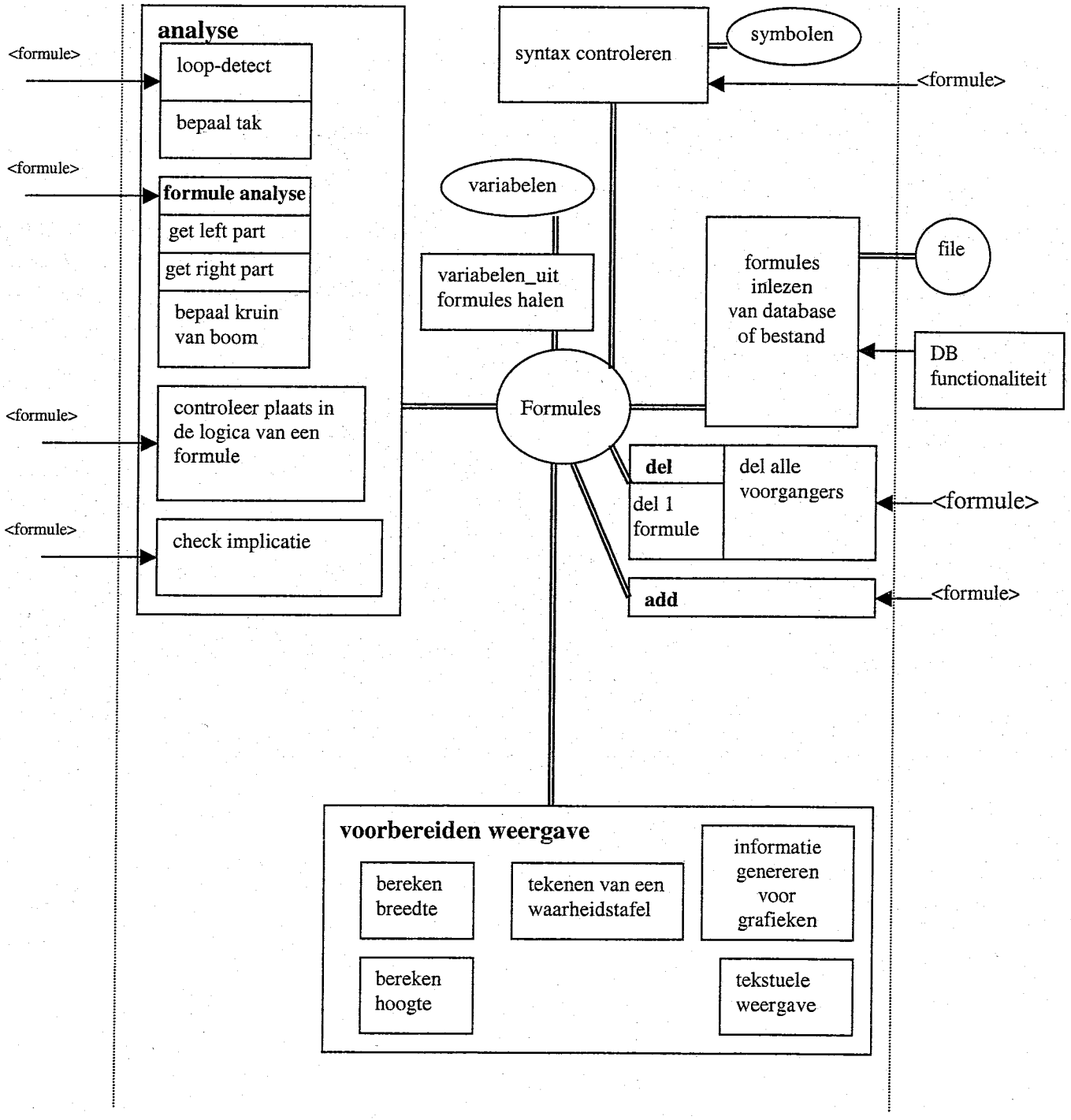
post: <comment> is een string die aangeeft wat de plek is van <formule> in <formules>. Als <formule> helemaal niet in <formules> voorkomt, geldt <comment> is *false*.

Anders bevat <comment> het commentaar dat nodig is om <formule> correct te plaatsen.

syntax controleren

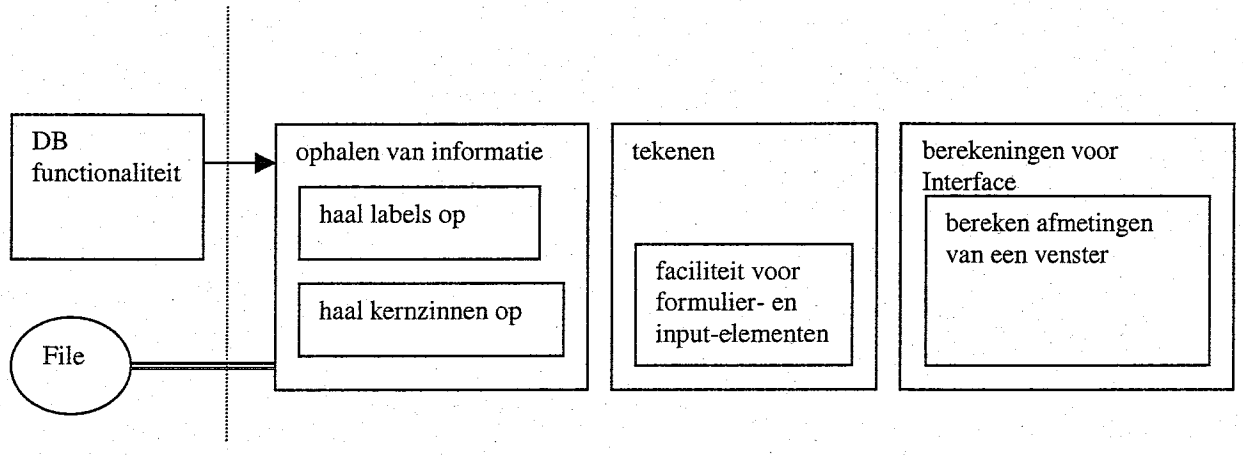
pre: <formule> is een formule

return: *true* als <formule> syntactisch correct is, dus als er alleen geldige variabele-namen en operatoren in voorkomen, en de expressie *geparsed* kan worden als een formule



4.2.2 Het Interface-component

Het Interface-component biedt routines aan ten behoeve van het Interface. In de abstracte beschrijving van het programma wordt een Interface-component voorgesteld om de statische aspecten van het User Interface te kunnen beschrijven, naast het dialoog-model.
De functionaliteit die deze component aanbiedt, is te scheiden in (1) *manipulatie van randapparaten* (op het scherm tekenen), (2) *berekeningen die specifiek voor het Interface moeten gebeuren,* en (3) *informatie ophalen die voor het uiterlijk van het Interface belangrijk is.* Het Interface-component gebruikt functionaliteit van de database-component en van de logica-component.
Op deze manier is het statische aspect toch beschreven, en kan in § 4.3 het dynamische model worden beschreven.



4.2.3 De Database

De architectuur van de database laat zich onafhankelijk van de andere componenten beschrijven. In de eerste paragraaf van hoofdstuk 5, over de keuze van de techniek, zal ik iets zeggen over de implementatie van de database in Oracle. Hier beschrijf ik het hoe en waarom van de tabellen.
De relationele database bevat vijf verschillende tabellen. De datatypes zijn "text", behalve bij ID, en bij de velden waar dat expliciet is vermeld.

secties

De tabel met de directe gegevens over de verschillende secties, dus de verschillende teksten die de studenten gepresenteerd krijgen. De velden van deze tabel zijn:

ID	titel	URL	redenering

Een sectie heeft een **titel**, en hoort bij een **redenering**. Normaal gesproken kan een redenering worden geïdentificeerd met een cursus, en aangezien het programma voor één cursus wordt gemaakt, is het veld overbodig. Om toch de uitbreidbaarheid te waarborgen, is het veld eraan toegevoegd.
Een **URL** verwijst naar de bron-tekst die de student moet bestuderen. Normaal gesproken is dat een file die op de server staat, maar door het als URL op te slaan, kunnen willekeurige teksten op het WWW met het programma aan studenten worden gepresenteerd.

kernzinnen

ID	tekst	sectie (number)	label	prioriteit

Een kernzin heeft een **tekst**, hoort bij een **sectie**. De secties kunnen dus worden geanalyseerd in termen van elkaars kernzinnen. Een kernzin kan een **label** hebben. Dit is een tekstveld dat in het Interface constant getoond kan worden. Als zodanig neemt het een plaats in tussen het **ID** van de kernzin, en zijn tekst (zie § 5.2 over details m.b.t. de implementatie van het Interface).
Een kernzin heeft een **prioriteit**. Dit is een nummer dat relatief is ten opzicht van de sectie waarbij de kernzin hoort. Er zijn dus 2 ordeningen van kernzinnen: op ID, en op prioriteit. Dit zal ook nuttig zijn voor het Interface.

formules

ID	formule	sectie (number)

Een formule, zoals die boven is beschreven in § 4.2.2 over de logica-component, heeft een **formule**-expressie, en hoort bij een **sectie**. Er is gekozen voor het letterlijk opslaan van de formule, en niet van de interne relaties waaruit die formule bestaat. Dit omdat ook invoer via een tekst-bestand mogelijk moet blijven, en omdat de formules al in detail door de logica-component geanalyseerd worden.

opties

	ID	tekst	kernzin (number)	correct (1/0)

Een optie heeft een **tekst**, hoort bij een kernzin, en kan wel of niet correct zijn. De tekst van een correcte optie is dus 2 keer opgeslagen: bij de kernzin zelf en bij de optie. Er is gekozen voor dit stukje redundantie van informatie, vanwege de flexibiliteit van het programma: er moet met opties gewerkt kunnen worden zonder dat er kernzinnen beschikbaar hoeven te zijn (door gebruikers die alleen de "eerste fase" willen gebruiken – zie verder § 4.3 voor een beschrijving van de fases in de dialoog met de gebruiker).

uitleg

	ID	tekst	optie (number)

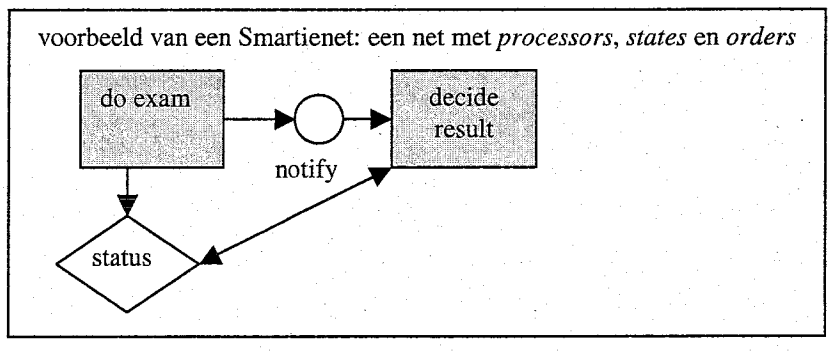
Een uitleg met een **tekst** hoort bij een **optie**, en kan worden getoond op het moment dat de student een verkeerde optie heeft aangeklikt. Voor de flexibiliteit van het programma is het ook mogelijk een uitlegtekst bij een goede optie de geven. Dit heeft dan de didactische functie van een hint achteraf, een geheugensteuntje, of een bonus omdat de student het goede antwoord heeft gekozen.

4.3 Dynamische beschrijving: de dialoog met de gebruiker

In deze paragraaf is de dialoog met de gebruiker beschreven. Ik heb dat op twee manieren gedaan. Ten eerste op een vrij abstracte manier om de dialogen abstract te karakteriseren (zie § 4.3.1); en ten tweede meer in detail door middel van petrinet-modellen (zie § 4.3.2). Eerst zal ik echter een korte verantwoording geven van het gebruik van petrinet-modellen voor een dynamische modellering van het systeem.
In [Dietz1991] wordt beargumenteerd dat een conceptueel model, zoals dat bij het statische modelleren wordt gebruikt, ook bij dynamisch modelleren moet worden toegepast. Een conceptueel proces-model heeft dan de volgende eigenschappen:

- A conceptual process model expresses the pragmatic structure of the communication between the components, independent of the way in which this communication is realized. It abstracts particularly from all information collecting, processing, storing, transporting and distributing activities.
- There is exactly one conceptual process model for every system, but there are several candidate logical process models. These are all correct (logical) realizations of the system, differing only in the logical structure of information processors, flows and stores by which the system is realized.[19]

Zo'n conceptueel proces-model kan grafisch worden voorgesteld als een soort petrinet. Dietz beschrijft communicatie in termen van *orders* en *statements*. Een order is de combinatie van een 'request' en een 'promise' op een bepaald tijdstip; een statement is de toestand van een systeem op een bepaald tijdstip. Een processor kan orders uitvoeren, en kan daarbij afhankelijk zijn van statements. Een net met processoren, banks en channels heet een Smartienet.



Deze manier van combineren van statische en dynamische aspecten van het systeem is aantrekkelijk voor een totaalbeeld van het systeem, maar alleen nodig als die aspecten werkelijk niet los van elkaar beschouwd kunnen worden. Dat is bij ons programma wel het geval; we zullen daarom uitgaan van 'normale' petrinetten om de communicatie te modelleren[20].

[19] [Dietz1991], blz. 38.
[20] De petrinet-beschrijving komt in plaats van het *state transition diagram* van UML.

4.3.1 Een globaal stroomschema

Een *dialoog* is te karakteriseren als een reeks $< x_1, y_1, x_2, y_2, \ldots >$.
Waar x_i een actie is van de gebruiker en y_i een actie is van de computer.
Er geldt dat y_i direct afhankelijk is van x_i voor alle $i>0$, en dat x_i+1 afhankelijk is van y_i èn de gebruiker. De communicatie-activiteiten kunnen zo worden gekarakteriseerd in termen van een proces-model. Waar nodig kan dit in meer detail worden uitgewerkt[21].

Er zijn een aantal subdialogen in het programma. Ik beschrijf die hier als paden. In deze beschrijving van de subdialogen komen reeds een aantal ontwerp-beslissingen naar voren. Die zal ik hier toelichten. Voor de exacte beslissingen, zie § 4.3.2.

- **de centrale dialoog**
  De acties zijn hier
  login = de student logt in.
  kies = selecteer een link in het openings-scherm [gebruiker];
  link = verbindt naar een andere dialoog [programma].
  loguit = de student logt uit.

  < login . (kies, link)* . logout >

- **het kiezen van een logica**
  De acties zijn hier
  sel = selecteer een logica [gebruiker];
  toon = toon de tekst van een logica [programma]
  kies = de student kies de geselecteerde logica [gebruiker].

  < (sel, toon)* . kies>

- **de eerste-fase**
  De acties zijn hier
  sel = selecteer een zin [gebruiker]
  kies = kies de geselecteerde zin [gebruiker]
  goed = meldt *ok* [computer]
  fout = meldt *fout* [computer]

  $< (((sel.kies)^* . goed)^* . ((sel.kies)^* . fout)^*)^* >$

  De gebruiker kan in deze dialoog net zolang doorgaan als hij wil, totdat hij alle goede kernzinnen heeft gekozen.

- **de tweede-fase**
  De acties zijn hier
  kies_add = voeg een kernzin toe [student]

[21] Te denken valt aan situaties waarbij deadlock-gevaar dreigt, of waar de interferentie van communicatie-acties niet direct duidelijk is.

add	= voeg een kernzin toe [programma]
kies_del	= delete een deel van de formules [student]
del	= delete een deel van de formules [programma]

$<(kies\_add.add)^{+}.((kies\_del.del)^{*}.(kies\_add.add)^{*})^{*}>$

De gebruiker moet eerst een zin hebben toegevoegd, waarna hij er telkens kan toevoegen of verwijderen
Er is geabstraheerd van de zijwegen in de dialoog, de mogelijkheden van de gebruiker om tussentijds op "help" te drukken, of terug te gaan naar de centrale dialoog.

- **de uitleg**
  De uitleg-dialoog is triviaal. Er wordt gebruik gemaakt van een standaard hypertekst om hulp-items te presenteren.

- **de docent-editor**
  De acties zijn hier:

kies_tabel	= kies een tabel [docent]
toon_tabel	= toon die tabel [programma]
sel_item	= kies item uit tabel [docent]
toon_item	= toon dat item [programma]
kies_add	= voeg een item toe [docent]
add_item	= voeg een item toe [programma]
kies_del	= del een item [docent]
del_item	= del een item [programma]
kies_upd	= wijzig een item [docent]
upd_item	= wijzig een item [programma]

$< (kies\_tabel.toon\_tabel).\ ((kies\_tabel.toon\_tabel)^{*}.\ ((sel\_item.toon\_item)\ .(kies\_del.del\_item \mid kies\_upd.upd\_item))^{*} \mid (kies\_add.add\_item)^{*})^{*} >$

Een docent kiest een tabel, en kan vervolgens items toevoegen, verwijderen of wijzigen. Voor het verwijderen of wijzigen moet eerst een item geselecteerd zijn.

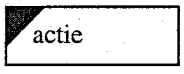
4.3.2 Dialoog-modellen

Met het logica-deel zijn de ontwerp-beslissingen die achter de schermen een rol spelen besproken[22]. We concentreren ons nu vooral op het Interface gedeelte. Hier beschrijf ik het ontwerp van de dialoog met de gebruiker. De dialoog-aspecten van het hele programma worden hier beschreven, om de nadruk op de eenheid ervan te houden. Er vallen dan de volgende delen te onderscheiden:

- een didactische dialoog. Deze dient voor het aanleren van de logica.
- een administratieve dialoog. Deze dient voor het algemene bedienen van het programma.

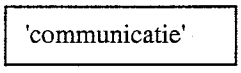
Deze twee dialogen zijn in de praktijk met elkaar verstrengeld (een student moet 'huishoudelijke' taken zoals het openen van een andere logica verrichten terwijl hij bezig is met een leerproces); toch worden ze hier onderscheiden omdat ze twee afzonderlijke perspectieven op het programma vormen die zelfstandig kunnen worden getoetst aan de eisen van de gebruiker.

Het formaat dat ik heb gekozen voor de dialoog-modellen is een afgeleide van petri-netten. Het nuance-verschil is dat er twee verschillende soorten acties zijn: communicate-acies en acties die door de computer worden uitgevoerd (data-transacties). In de regel moeten pijlen van acties naar toestanden en omgekeerd lopen. Daarop zijn uitzonderingen: als een pijl het deel-model verlaat, dan eindigt ze niet in een toestand. Dit moet worden genegeerd bij analyse over de hoeveelheid tokens die in het systeem aanwezig is.
De symbolen waarvan ik in de dialoogmodellen ben uitgegaan zijn dus:



een actie van de computer (een berekening of data-transactie uitvoeren)

een toestand



een communicatie-actie

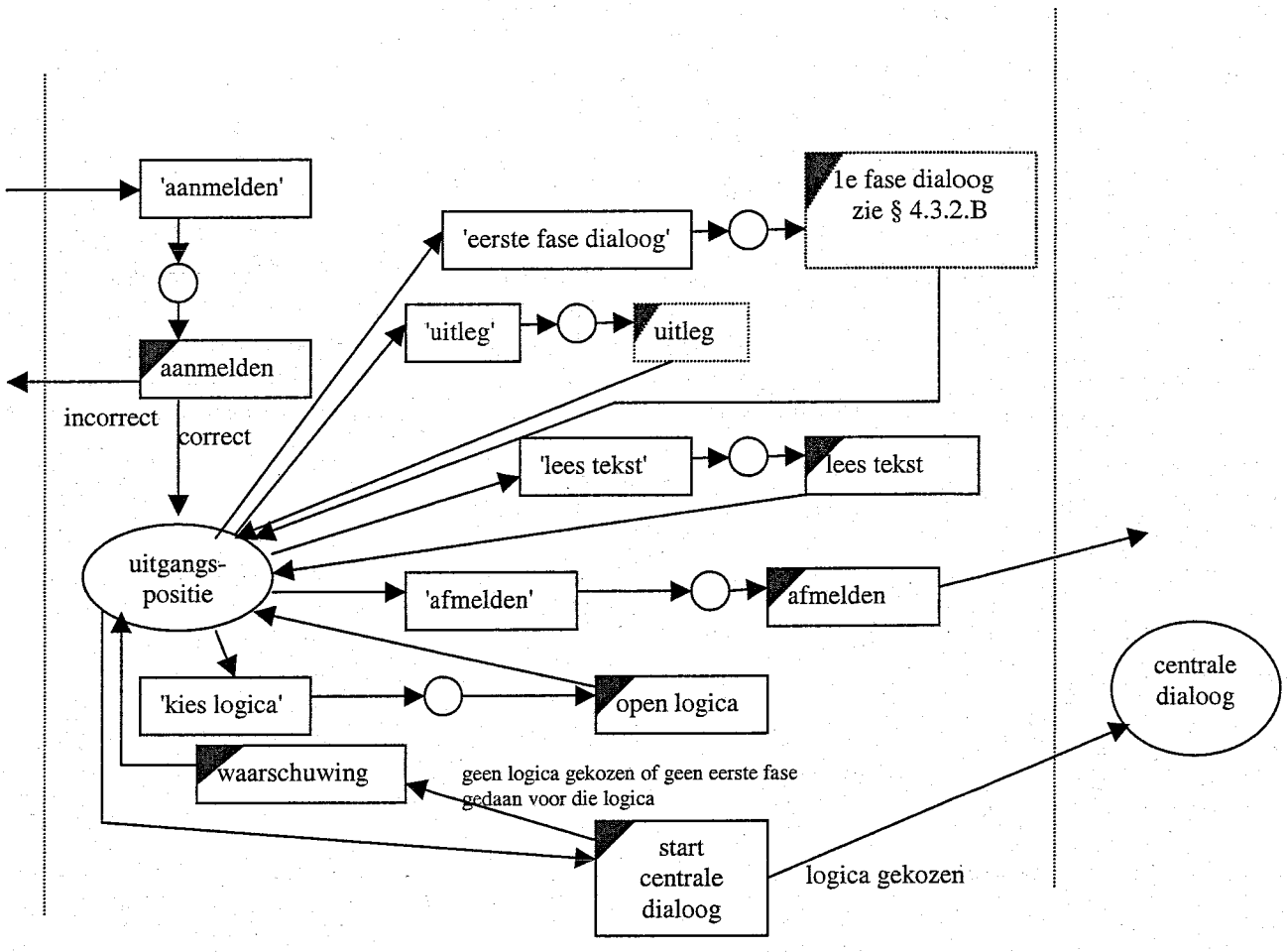
Bij de pijlen kunnen de condities worden aangegeven in het geval van een actie met meerdere uitgaande pijlen. Het dialoogmodel begint bij de situatie dat een student het programma opstart.

In dit dialoogmodel is de communicatie van de student met de Argument Editor beschreven. Het model is opgesplitst in vijf delen: de openings-dialoog, de eerste-fase-dialoog, de centrale dialoog, het kladblok en de uitleg-dialoog. De vijf dialogen worden grafisch gepresenteerd op de boven beschreven manier, en zijn van commentaar voorzien waar dat nodig is. De ontwerp-beslissingen die te maken hebben met de manier waarop een dialoog wordt uitgevoerd zijn hier verantwoord.
De grafieken spreken voor zichzelf. De enige 'afkorting' waarvan ik extra gebruik heb gemaakt zijn de stippellijnen: tussen de stippellijnen links en rechts zijn de

[22] Technische ontwerp-beslissingen over de implementatie staan in hoofdstuk 5.1.

communicatie-acties en computer-acties gestructureerd. Verder betekenen de gestippelde kaders acties die elders nog verder worden uitgewerkt; ze zijn voorzien van een paragraaf-nummer.

4.3.2.A De openings-dialoog

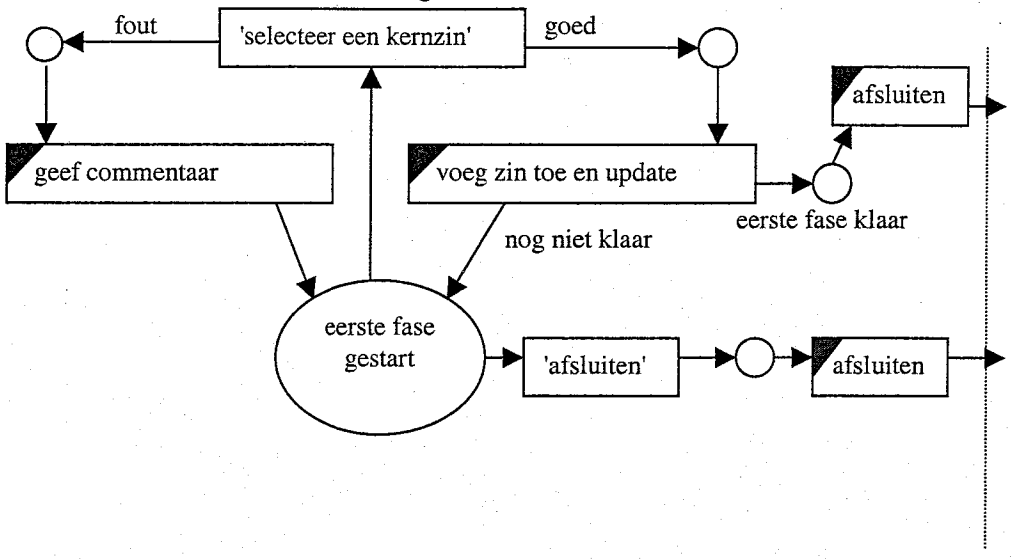


Als de student in de openings-dialoog komt, moet hij de communicatie-actie 'aanmelden' verrichten. Het programma zal dan, bij correct aanmelden, in de uitgangs-toestand van de openings-dialoog raken. Dit aanmelden is een aparte stap, omdat het programma beschermd moet worden: alleen de doelgroep moet toegang hebben. Om toch een snelle toegang te geven voor hen, die slechts geïnteresseerd zijn in een demonstratie van het programma, is er een 'shortcut'-route, die een standaard-logica opent en de eerste fase-dialoog daarmee opstart. In de praktijk zal het aanmelden gebeuren met een unieke sleutel voor een student; die bezitten studenten reeds in de vorm van een bibliotheekwachtwoord. Hierdoor hoeft er in ons programma geen communicatie plaats te vinden over de bevestiging van de identiteit van de student.
Vanaf deze uitgangspositie is het mogelijk om uitleg te vragen (zie § 4.3.2.E). Uitleg is nog niet beschikbaar voordat de student is aangemeld. De reden daarvoor is dat we

het Interface willen opbouwen als *tool*[23]: de student moet het idee hebben dat hij via het Interface bewerkingen kan uitvoeren. Daarvoor moet het Interface een vast en vertrouwd centrum hebben, te vergelijken met een werkplaats. Dit is hier de uitgangspositie van waaruit de student alle relevante handelingen kan verrichten.
Eén van de mogelijkheden vanuit de uitgangspositie is om een logica te kiezen, die vervolgens wordt geopend (zie § 4.3.2.C). Het kiezen van een logica is expliciet gemaakt. Hierdoor moet de student zelf aangeven met welke tekst hij wil werken. Dit draagt bij aan een meer actieve opstelling[24], en richt de aandacht van de student waar die moet zijn, namelijk op de tekst die hij analyseert. Nadat er een logica is gekozen, komt de student terug in de 'werkplaats'. Hij heeft nu, metaforisch gesproken, het materiaal dat in zijn bankschroef staat veranderd.
Vanuit de uitgangspositie kan de student kiezen voor 'lees tekst': de tekst zal dan worden gepresenteerd; als er al een logica is gekozen, dan wordt de tekst die daarbij hoort (eventueel in haar context) getoond.
De uitgangspositie wordt verlaten door naar de centrale dialoog of de eerste fase dialoog te gaan (zie § 4.3.2.B). We kiezen bewust voor een zo eenvoudig mogelijk gezicht van het programma door het rond een centrale dialoog op te bouwen, en de toegang tot die dialoog vanuit één uitgangspositie te structureren. Als de student de centrale dialoog wil opstarten zonder dat hij een logica heeft gekozen of als hij de eerste-fase dialoog nog niet heeft gedaan, wordt hij daarop gewezen. Hier wordt de student weer gedwongen expliciet te maken welk 'materiaal' hij wil gaan bewerken.
De aangemelde student kan zich weer afmelden, en zo de Argument Editor verlaten.

[23] Cf. [Preece1994], blz. 461.
[24] Uit de testsessies bleek dat het voor de studenten van groot belang is om zich bewust te zijn van het verband tussen de papieren tekst en wat er op het scherm verschijnt.

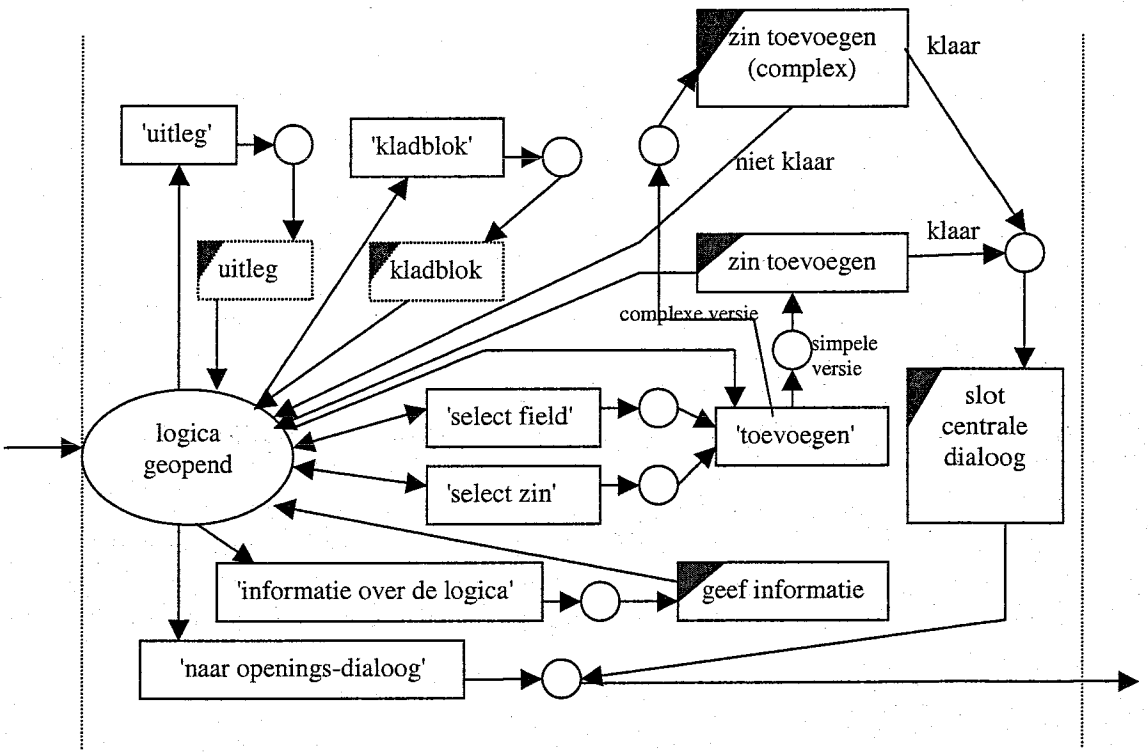
4.3.2.B De eerste-fase dialoog



In de eerste fase moet de student de zinnen uit de tekst halen. De dialoog die hierbij wordt gebruikt legt de nadruk op een rondgang: de student moet telkens een paragraaf lezen en daarbij aangeven wat de kernzin is (of wat de kernzinnen zijn). Als hij dit niet goed doet, moet hij overvloedig van commentaar worden voorzien[25]. De lus die de student zodoende doorloopt is herhaaldelijk 'het grondig lezen van een paragraaf - het uitkiezen van een kernzin - het lezen van het commentaar'. Om dit zo natuurlijk mogelijk te laten gebeuren hebben we voor één enkele invoermogelijkheid gekozen, namelijk het selecteren van zinnen uit meerdere keuzemogelijkheden[26].
Op het moment dat de student kiest om de eerste fase af te sluiten of als hij alle antwoorden correct heeft, eindigt de eerste fase. Het programma zal dus net zo lang doorgaan met vragen aan de student om de zinnen in te vullen, tot hij ze allemaal goed heeft.
De student komt dan terug in de uitgangs-positie, vanwaar hij kan kiezen om met de volgende fase (de centrale dialoog) verder te gaan.

[25] De eerste fase steunt vooral op het commentaar dat de studenten van het programma krijgen, blijkt uit de testsessies. "*Evaluatie*: de eerste fase is zonder veel problemen geaccepteerd. Het presenteren in een willekeurige volgorde van alternatieven, en het geven van veel didactisch commentaar behoort bij de fine-tuning van het programma."
[26] De studenten zijn over het algemeen gewend aan meerkeuze-vragen. (Zie de testsessies)

4.3.2.C De centrale dialoog

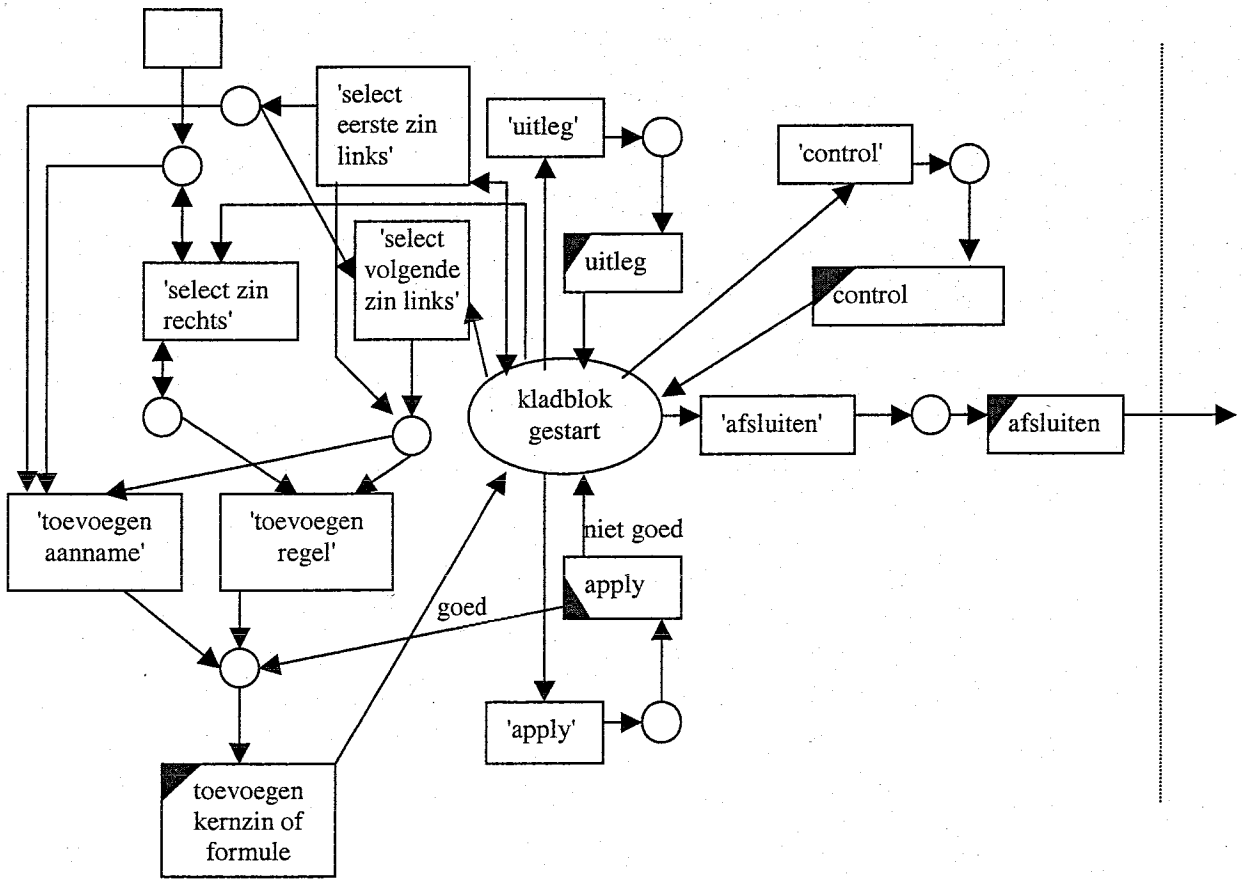


De centrale dialoog is het gezicht van de Argument Editor. De leer-elementen (zie 3.1) zijn rond deze dialoog gestructureerd. De centrale dialoog laat studenten de logica zelf reconstrueren. Ik ben daarbij uitgegaan van de gegevens uit de testsessies: de studenten gaven allen de voorkeur voor een dialoog waar gedurende de gehele sessie in één blik duidelijk is wat er moet gebeuren. Dat betekent dat de studenten de redenering op een passieve manier benaderen: ze willen liever niet de redenering re*construeren*. Daarom hebben we een onderscheid gemaakt tussen een ingewikkeldere en een simpele versie, waarbij de simpele verplicht gesteld kan worden. De dialogen die zich afspelen in deze versies komen min of meer overeen; daarom zijn ze hier in één paragraaf beschreven. De manier waarop de zinnen worden toegevoegd aan de reconstructie, is eigenlijk het enige verschil; daarom is dat ook als twee verschillende acties gemodelleerd, namelijk "zin toevoegen" en "zin toevoegen (complex)".

De student doorloopt een lus waarin hij de kernzinnen die hij in de eerste fase heeft geïdentificeerd kiest en ze op plaatsen in de structuur neerzet (de simpele versie) of de bestaande structuur wordt uitgebreid met een plaats voor die kernzin (de ingewikkeldere versie). Als hij een zin correct heeft gekozen, wordt die direct ingevuld; als de student een fout maakt, krijgt hij daarover commentaar (of hij een stap heeft overgeslagen, of een argument is vergeten). Het kiezen van een zin en het selecteren van een veld waar die zin in moet worden geplaatst gebeurt simultaat: dit is met een petrinet-constructie gemodelleerd: er moeten *tokens* in twee verschillende plaatsen liggen voordat de communicatie-actie 'toevoegen' kan vuren. Het selecteren van een kernzin of het kiezen van een veld verandert de toestand in principe niet: dat

verklaart de dubbele pijlen van de 'select'-acties naar de plaats 'logica geopend'. Het token uit 'logica geopend' moet later worden weggehaald door de actie ' toevoegen'.
Als de student alle zinnen op de juiste manier heeft toegevoegd, eindigt de centrale dialoog. Dit gebeurt expliciet, waarbij de student nog extra informatie krijgt aangeboden over de redenering en de logica die hij succesvol heeft gereconstrueerd. Daarna gaat het programma terug naar de openings-dialoog.
Vanuit het uitgangspunt van de centrale dialoog moet de student de functies die hij nodig zou kunnen hebben kunnen opvragen. Dat betekent dat hij naar de uitleg-dialoog kan gaan voor uitleg over de werking van het programma (zie 4.3.1.E), het kladblok (4.3.2.D) kan opstarten, informatie kan krijgen over de specifieke logica waarmee hij op dat moment werkt, en terug kan gaan naar de openings-dialoog.
De informatie over de specifieke logica (in de vorm van aantekeningen bij de kernzinnen en de manier waarop die met elkaar worden verbonden) moet goed worden gescheiden van de uitleg-dialoog. Ze zijn hier daarom ook apart gemodelleerd.

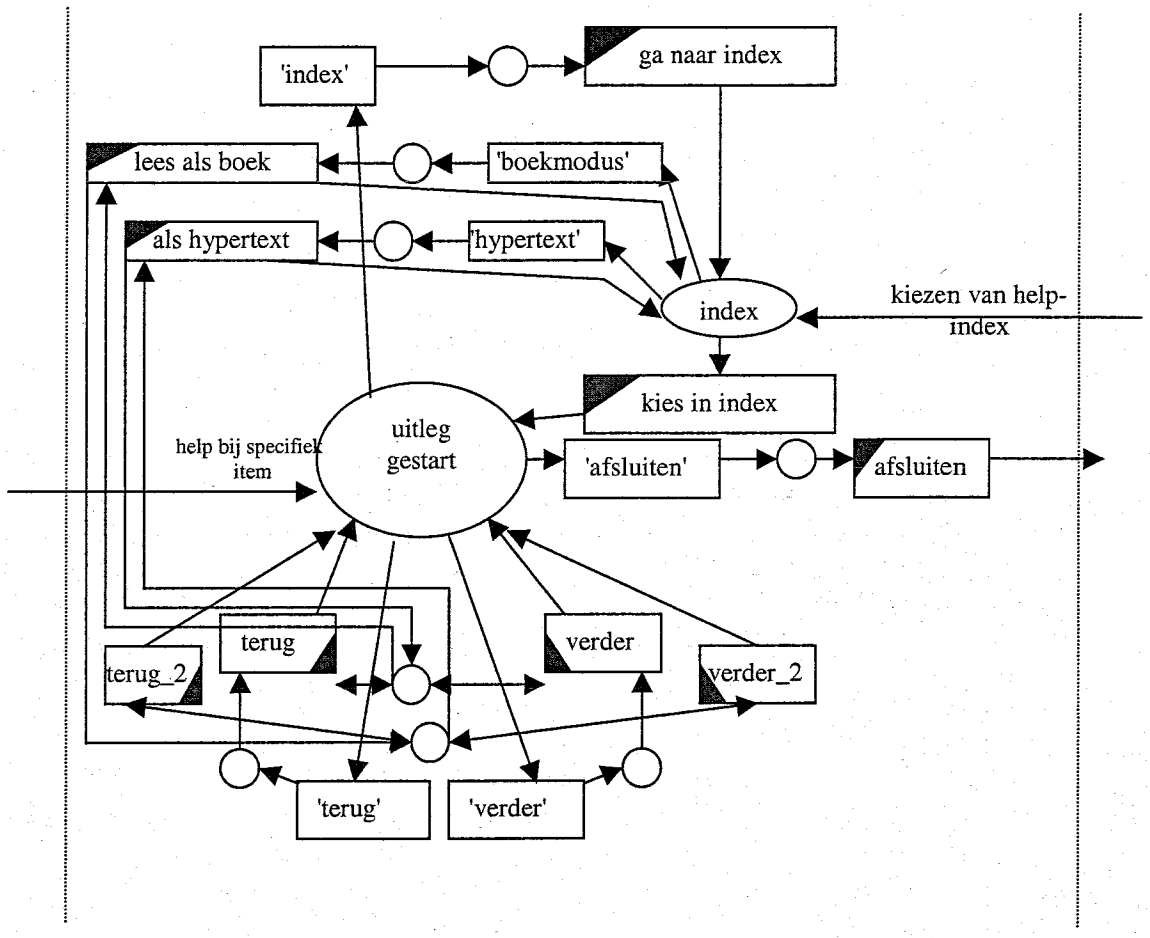
4.3.2.D De kladblok-dialoog



De kladblok-dialoog is een soort logica-rekenmachine die de student kan gebruiken als hulpmiddel bij de centrale dialoog[27]. Daarnaast biedt het wat meer mogelijkheden om vrij met de logica om te gaan, zodat er rekening kan worden gehouden met het niveauverschil tussen de studenten[28] qua omgang met (formele) logica, en er toch een duidelijke eenheid in het programma bestaat, dat gestructureerd is rond de centrale dialoog.
Vanuit de uitgangspositie, 'kladblok gestart', kan de student uitleg vragen over de werking van het kladblok, of de logica die hij tot dan toe heeft ingevoerd controleren.
Het invoeren van een logica gebeurt volgens het volgende procedé: de student selecteert zinnen 'links' en 'rechts'. De metafoor die hier gebruikt wordt om de student te laten aangeven of hij een regel of een aanname invult, is de metafoor van een invulformulier. De student moet kiezen of hij een regel of een aanname invoert. In het geval van een aanname heeft hij precies 1 kernzin 'links' uitgekozen. In het geval van een regel heeft hij tenminste 1 kernzin 'links' gekozen, en precies 1 kernzin 'rechts'. Deze condities zijn door middel van petrinet-constructies verwerkt in het dialoogmodel. Er is daarvoor een onderscheid gemaakt tussen het selecteren van de eerste zin 'links' en de latere zinnen, omdat alleen één enkele zin als aanname kan fungeren.
Het is belangrijk om het onderscheid tussen aannames en regels expliciet te laten maken door de studenten. Ze houden zich dan in principe bezig met de leermomenten, zoals die zijn geformuleerd in hoofdstuk 3.1. (In dit geval gaat het vooral om de didactische eis Di-2).

[27] De kladblok-dialoog is geïmplementeerd in een evaluatie-model. De studenten die deel hebben genomen aan de test (bijlage 1) vonden echter deze vorm niet aantrekkelijk, en kozen voor een representatie met grafische middelen.
[28] Eén van de guidelines van [Preece1994] is om rekening te houden met verschillende user level's (blz. 490). Normaal gesproken gaat het om verschillende (intellectuele of motorieke) capaciteiten om het programma te bedienen; de stelling kan ook worden toegepast op het verschil in achtergrondkennis bij de studenten. (die al dan niet een wiskundige onderlegging hebben).

4.3.2.E De uitleg-dialoog



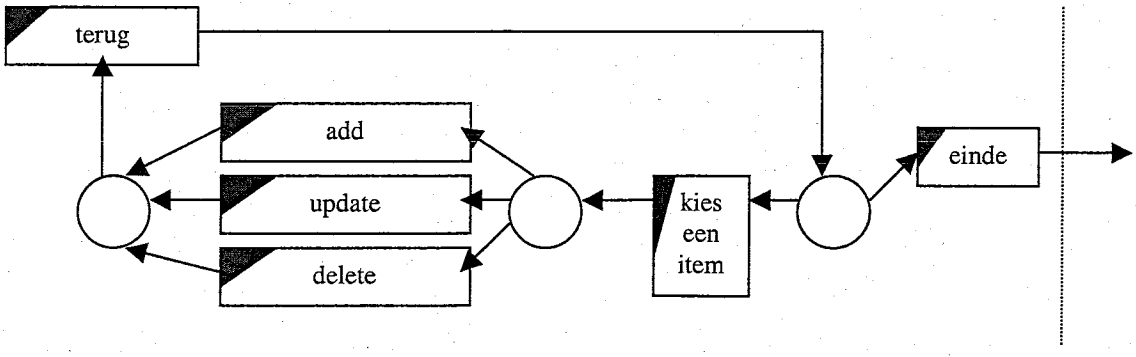
De uitleg-dialoog is gemodelleerd naar een standaard, zoals die ook bij veel web-browsers voorkomt, namelijk met een terug - index - vooruit structuur. De student heeft ook hier te maken met een vertrouwde omgeving, die bestaat uit twee fragmenten, namelijk de index en de uitleg-items zelf. De uitleg-dialoog is dus gemodelleerd naar de metafoor van een boek, waarin een index staat die voor de gebruiker informatie snel toegankelijk maakt. De samenhang tussen de index en de hoofdstukken van het boek is meteen duidelijk, aangezien het fysiek om één en hetzelfde object gaat [29]. De samenhang tussen deze fragmenten moet in de implementatie (zie 6.2) worden uitgedrukt.
Vanuit de uitgangspositie met een bepaald uitleg-item kan altijd de index worden opgeroepen om snel naar een ander item te gaan. De index is ook toegankelijk via de

[29] Het is intuïtief direct en volledig duidelijk waarnaar een index in een boek verwijst; of een index voorin het boek (zoals bij Engelse uitgaven) gesitueerd is, of achterin (zoals bij Franse boeken) maakt niet uit. De samenhang met de informatiebron is duidelijk en vertrouwd; bijvoorbeeld de dikte van het boek is een maat voor de hoeveelheid informatie die met de index bestreken wordt. Bij computer-programma's bestaat er niet zo'n maat, en dat kan de vertrouwdheid wegnemen. De oplossing die ik gekozen heb, is om de dialoog zo de modelleren dat de informatie-items allen een min of meer vaste grootte hebben (voorzover dat mogelijk is): de student heeft dan met de grootte van de index een maat voor de grootte van de uitleg-informatiebron, hetgeen moet bijdragen tot de vertrouwdheid en daarmee de drempel tot het oproepen van de uitleg-dialoog verkleint.

permanente optie 'uitleg' vanuit de andere dialogen. In de implementatie kunnen vanuit bepaalde dialogen bepaalde delen van de index worden opgeroepen.
Het 'verder - terug' mechanisme geeft de mogelijkheid om te bladeren, een belangrijk middel om de vertrouwdheid en handigheid te bewerkstelligen. Er zijn twee mogelijkheden om te bladeren: voor- en achteruit bladeren door het 'boek' van de uitleg, dus volgens een vaste structuur, en bladeren door de uitleg-items die de student al heeft opgeroepen. De eerste manier heeft het voordeel dat direct duidelijk is 'waarin' de student bladert, de tweede manier dat items die recent bekeken zijn (en dus meer kans hebben relevant te zijn) kunnen worden teruggeroepen. Beide manieren combineren in één dialoog zaait te veel verwarring: de intentie van de student moet eenduidig kunnen worden afgebeeld op de toestand van het systeem ([Preece1994]: "reduce discrepancy between user's goals and system's physical state.", blz. 263). Om daaraan te voldoen, en toch de metafoor van de vaste 'boek'-structuur vast te houden, besluiten we om de uitleg-dialoog zowel als een lineair on-line 'boek' en als een hypertext met een temporele bladerstructuur aan te bieden. Daarbij moet wel duidelijk worden gemaakt dat het om dezelfde tekst gaat. De uitleg-dialoog wordt dus gepresenteerd in een boek-modus (met de bladerstructuur 'terug' en 'verder') of een hypertext-modus (met de bladerstructuur 'terug_2' en 'verder_2'). Er kan worden overgeschakeld tussen de verschillende modi, vanuit de index. (Niet vanuit een willekeurig uitleg-scherm, omdat dat de aandacht kan afleiden.)
De intentie van de student als hij uitleg oproept bij een enkel item is hulp krijgen over dát item; die hulp moet zo concreet mogelijk worden aangeboden. De hulp kan hem soms nopen meer uitleg op te zoeken. Dit moet hij via de index doen. Deze extra stap is niet te vermijden zonder de vertrouwdheid uit het oog te verliezen: we zouden handige 'shortcuts' moeten aanbrengen in de dialoog die de student echter als verwarrend kan ervaren. De temporele bladerstructuur kan in veel gevallen hem wèl helpen: het programma wordt zo ontwikkeld dat het bijbehorende leerproces bottom-up is; als de student dus reeds uitleg heeft opgevraagd bij een vorig item, kan hij die informatie snel terugvinden.

4.3.2.F De docent-dialoog

De docent moet wijzigingen kunnen aanbrengen in de database. Daarvoor staat hem een simpele Interface ter beschikking, die bijna onmiddellijk volgt uit § 4.2.3. Voor de implementatie is uitgegaan van standaard Database Interface-technieken, die verder niet beschreven worden. De docent heeft vrije toegang tot de database door één gemeenschappelijk wachtwoord: alle docenten kunnen dus bij de gegevens van de cursussen. Die kan, omdat het aantal docenten beperkt zal blijven, en er een goede communicatie tussen hen bestaat.



5. De implementatie van het Interface

In dit hoofdstuk beschrijf ik de implementatie van het Interface van de Argument Editor. In de eerste paragraaf geef ik een overzicht en een verantwoording voor de keuze van de techniek die ik heb gebruikt. Daarna zal aan de hand van algemene vuistregels voor User Interfaces een verzameling vensters worden afgeleid uit de dialoog-modellen van hoofdstuk 4.3.

5.1 De keuze van de software-techniek

De Argument Editor is geïmplementeerd in PHP 4 [30]. De keuze voor deze programmeertaal lag voor de hand, aangezien het de taal was die beschikbaar is op de server waarvan ik gebruik maak. De keuze voor een server-sided web-applicatie überhaupt, is gemotiveerd door de aard van het programma: er hoeft geen razendsnelle real-time reactie te worden gegeven door het programma, omdat het gaat om een leer-proces, waarbij de student steeds ook de tijd moet nemen om na te denken.
Naast PHP wordt gebruik gemaakt van Oracle 7 voor de implementatie van de database[31]. Ook dit systeem was voorhanden op de server die het programma draait. Een lichtere database-applicatie was een mogelijkheid geweest, maar biedt geen voordelen ten opzichte van Oracle 7, en brengt extra overhead met zich mee.
Aan de client-side is, naast van HTML[32], gebruik gemaakt van de scripting-taal JavaScript 1.2[33]. JavaScript is een wijdverbreide taal, en wordt tegenwoordig goed door web-browsers ondersteund.
Het programma is op drie verschillende browsers getest: Microsoft Internet Explorer, Netscape Navigator 4.7, en Mozilla. Alleen Netscape Navigator leverde problemen op, zij het niet onoverkomelijk.
De specificatie van de studenten-werkplek [34] waar het programma zal worden gebruikt, voldoet, en heeft beschikking over Internet Explorer 5.

5.2 De keuze van het Interface-techniek

In deze paragraaf beschrijf ik de eigenlijke *User Interface* aspecten van de implementatie. Eerst geef ik een overzicht van de voor het programma relevante aspecten van de door mij bestudeerde literatuur. Dit zal leiden tot een aantal vuistregels, die ik in het programmeren van het Interface zoveel mogelijk heb proberen toe te passen.

---

[30] Voor referentie over PHP 4 heb ik gebruik gemaakt van [Converse2000].
[31] De gebruikte handleiding is: http://info-it.umsystem.edu/oracle/svslr/svslr.1.toc.html.
[32] De gebruikte versie van HTML wordt gekarakteriseerd in de meta-tag: <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
[33] Zie [Kassenaar1997b].
[34] De specificaties zijn te vinden op http://cwis.kub.nl/~drc/giw2000/.

Algemene regels die in acht moeten worden genomen bij het implementeren van het Interface zijn[35, 36]:

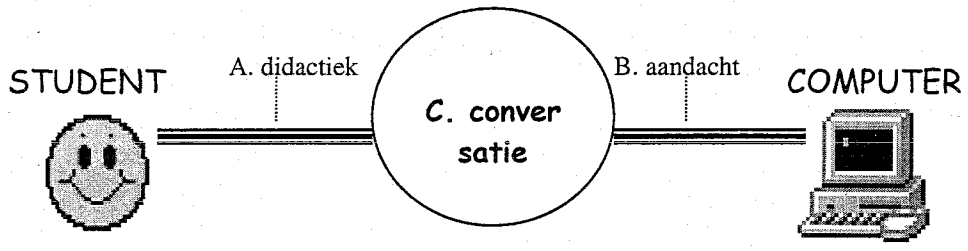
1) *allow query in depth.* Voor de gebruiker moet de informatie die hij 'in de diepte' wil vinden, gemakkelijk toegankelijk zijn. Dit is hier niet zo relevant, omdat het gaat om een *leeromgeving*, en geen zoekomgeving;
2) *design for user growth.* De gebruiker (student) moet kunnen groeien in de omgang met het programma. Dat is van belang, aangezien de vaardigheden verschillende tussen studenten, en dus ook kunnen groeien. Het Interface moet te bedienen zijn met simpele en handige sneltoetsen, maar ook uitgebreide hulp bieden bij het uitvoeren van de user goals.
3) *allow input flexibility.* De flexibiliteit van de invoer houdt verband met punt 2). Het is ook voor ons van belang, hoewel deze regel *niet* boven de kwaliteit het leerproces gestelt moet worden. Ik kom daarop nog terug;
4) *adapt to different user levels and styles.* Ook dit is in punt 2) besproken;
5) *ensure ease of understanding.* Het Interface moet eenvoudig te begrijpen zijn. Relevant, omdat studenten vaak van hun eerste indruk laten afhangen of ze een programma gebruiken. Die eerste indruk moet meteen een begrijpelijk verhaal zijn, er moet dus gelden dat het aansluit bij hun beleving, en dat direct duidelijk is hoe dit hun studeer-proces kan bevorderen;
6) *give appropriate quantity of response.* De respons die een student krijgt van een didactisch programma moet gebalanceerd zijn. Het is hier *niet* het belangrijkste dat de respons efficiënt is en de lacunes van de student volledig aanvult, maar dat de respons de student actief zelf verder laat denken.

5.2.1 Vuistregels voor ons programma

De vuistregels die relevant zijn voor de Argument Editor, heb ik in verschillende groepen verdeeld.

1. Regels over *didactiek*;
2. Regels met betrekking tot *aandacht* zijn zonder meer belangrijk in didactische software;
3. regels over de *conversatie* met de gebruiker.

Deze drie groepen vuistregels zouden grafisch als volgt kunnen worden voorgesteld:



[35] Uit: [Preece1994], blz. 490.
[36] Zie ook: [Faulkner1998], blz. 56. De richtlijnen die hier worden besproken zijn flexibiliteit, relevantie, consistentie en natuurlijkheid. Deze richtlijnen zijn een subset van die van Preece (consistentie valt onder understanding en relevantie valt onder response).

A. didactiek

De vuistregels voor de didactiek, waarvan ik ben uitgegaan, kunnen als volgt worden geformuleerd[37]:

1. *Geef informatie vooraf, zodat de gebruiker weet waarop hij zich moet richten.* Als een gebruiker iets *nieuws* probeert te leren, zal hij zich op een terrein begeven, dat eerst nog onbekend voor hem is. Het programma moet een overzicht vooraf geven, zodat hij zich kan oriënteren. Het programma kan zich daarbij niet verlaten op een voorverstaan van begrippen (en uiteindelijk metaforen) zoals bij veel Interfaces wordt gedaan. Het is aan het Interface om als het ware van voren af aan de gebruiker vertrouwd te maken met iets nieuws. Deze extreme situatie komt bij kleine kinderen voor, maar voor onze toepassing is het een wat overdreven. Voor normale volwassen gebruikers is het namelijk mogelijk om een *scheiding* te maken tussen de inhoud van de informatie en de vorm waarin die gepresenteerd wordt. Het leren van de inhoud is een actief proces, waarbij het Interface weinig hulp kan bieden (het Interface kan wèl een belangrijke rol spelen in het leerproces, omdat dit proces uiteindelijk verweven is met een reeks Interface-manipulaties. Het Interface kan alleen niet de goede antwoorden aan de gebruiker voorschotelen!) Aangaande de vorm van de informatie kan een hulp-functie worden gebruikt (Zie § 5.2.2 C).
2. *Modelleer het proces.* Het leerproces moet nauwkeurig gemodelleerd worden. Waar het gaat om het aanleren van *argumenteren*, zoals in § 3.1 beschreven, in het bijzonder het met elkaar verbinden van eerder geïdentificeerde kernzinnen, is het proces erg afhankelijk van de stijl van de gebruiker. Die kan variëren van een trial-and-error methode (het Interface probeert ervoor te zorgen dat deze methode in ieder geval niet de gunstigste is – het blijft de verantwoordelijkheid van de student om na te gaan denken over wat hij invoert) tot een volledig van tevoren op papier uitwerken van de opgave. Het gaat erom dat de gebruiker een midden vindt, waarbij het Interface een positieve rol kan gaan spelen in het leerproces. Een model van het leerproces waarmee wij te maken hebben, staat in het hoofdstuk over de uitgebreide beschrijving van dialogen (§ 4.3.2).
3. *Gebruik beweging en animatie om verandering van state te representeren.* Hiervan maken we dankbaar gebruik, zij het op een iets andere manier dan Nicol misschien heeft bedoeld: we gebruiken beweging niet om het veranderen van de *interne* toestand van het programma te reflecteren, maar om informatie vooraf te geven over de toestand van het Interface. Een typisch voorbeeld is een pijltje dat 'uitklapt' in de richting van het te manipuleren Interface object als je de muis eroverheen beweegt, waardoor wordt aangegeven dat er op gedrukt kan worden. De verandering van de toestand van het programma moet zo duidelijk mogelijk zijn aan de gebruiker: die moet zich immers *in control* voelen (zie verder bij § 5.2.1 C. over conversationele richtlijnen). Maar beweging is voor deze duidelijkheid een noodoplossing, zoals in een gesprek de inhoud *ondersteund* kan worden met gebarentaal, maar deze moet niet op de voorgrond raken. Het

[37] Zie: [Nicol1990]. Dit artikel concentreert zich vooral op het leren aan *kinderen*. Het is dus slechts gedeeltelijk toepasbaar op de situatie van studenten.

programma moet dus de nadruk leggen op een duidelijke communicatie met de gebruiker, zodat beweging niet noodzakelijk wordt om toestandsveranderingen te reflecteren;

4. *Gebruik interactie zodat de intelligentie van de gebruiker centraal staat – Interfaces als coaches.* Dit is bij ons programma het geval. In het commentaar dat de gebruiker krijgt als hij een verkeerd argument aanvoert, stelt het programma zich op als een coach, die de student aanwijzingen geeft zoals een trainer een pupil aanwijzingen geeft om zich te verbeteren.

Er zijn drie soort *skill acquisitions*: cognitief, associatief en autonoom. Het is duidelijk dat bij ons sprake is van het bijbrengen van cognitieve vaardigheden. Dat betekent dat de student geen irrelevante associaties moet gaan maken – het associatieve proces moet goed gestuurd worden in het programma. Als er bijvoorbeeld een Interface-element gepresenteerd wordt moet duidelijk zo of en welke relatie er bestaat met het cognitieve element waarnaar verwezen wordt. Met andere woorden: een icoontje mag niet te snel worden *geässocieerd* met een betekenis, maar de student moet actief van het icoontje gebruik maken, wetend, dat de betekenis nooit samenvalt met de metafoor.

B. aandacht

Aan de kant van de gebruiker bevindt zich, in bovenstaande driedeling, het aspect van de aandacht. Dit moet iets breder worden opgevat als het om vestigen van een focus op bepaalhet Interface-elementen; het is de primaire gewaarwording van de gebruiker van het Interface, die voorafgaat aan zijn interactie ermee. Het aandachts-aspect is dus het onbewuste aspect van de conversatie; voor mensen met (visuele) handicaps is dit in de praktijk goed uitgewerkt[38], maar in de algemene situatie zijn hier weinig structurele richtlijnen. Toch zijn er een aantal aspecten van de grafische Interface die een behandeling verdienen. Ik behandel eerst metaforen, en vervolgens weergave (in het bijzonder kleuren).

Metaforen zijn in principe beelden, die verwijzen naar een deel van de werkelijkheid, die structureel van hen verschilt. Een metafoor voegt een nieuw beeld toe aan de werkelijkheid, waarbij wordt afgezien van sommige aspecten, of waarbij andere aspecten naar voren treden. Een hamer kan een metafoor zijn voor timmerwerkzaamheden, en een trommel voor geluid. Metaforen kunnen ook een belangrijke rol spelen in het Interface: het zijn immers geavanceerde vehicels van betekenis. Bij het maken van metaforen voor het Interface moeten voldoen aan een aantal richtlijnen:

[38] Ik ga er hier vanuit dat gebruikers met handicaps door externe faciliteiten (van het besturingssysteem) worden geholpen. In de primaire omgeving waar het programma moet functioneren, is dat inderdaad het geval. Wat het fysieke aspect van het bedienen betreft, kan worden opgemerkt dat het programma niet fysiek belastend is (het aantal muisklikken blijft bijvoorbeeld beperkt). Het voldoet daarmee aan de norm ISO 9240 (zie [Preece1994], blz. 507).

- *functional definition.* Metaforen moeten verwijzen naar een duidelijke functionaliteit. Een voorbeeld is de prullenbak in veel windows-omgevingen, waarvan de functionaliteit direct duidelijk is aan de metafoor.
- *identify user's problems with the concept.* Gebruikers kunnen echter problemen hebben met het concept in kwestie, zodat een goede functionele definitie niet mogelijk is. Er komt creativiteit bij kijken, om een metafoor te verzinnen die er juist toe kan bijdragen, dat de gebruiker meer vertrouwd raakt met het concept in kwestie!
- *metaphores must be real-world alike.* Hoe goed een gebruiker ook thuis kan zijn in een virtuele gedachtenwereld, de echte wereld blijft de belangrijkste bron voor metaforen. Dit geldt zeker als er metaforen moeten worden verzonnen met min of meer constante betekenis over een groep van gebruikers. Ons programma haalt haar metaforen-materiaal dan ook uit de echte wereld (zie § 5.2.2 voor de icoontjes die ik gebruik).

Metaforen moeten een plaats hebben in zowel het *design model*, dus het model waar de ontwerper vanuit gaat, als in het *mental model* van de gebruiker[39]. Ze moeten helder en eenvoudig zijn, en uitbreidbaar: de context waaruit Interface-metaforen worden afgeleid moet dus rijk genoeg zijn om voldoende structuur te kunnen representeren om de gebruikerstaken die met metaforen moeten worden aangeduid, van metaforen te kunnen voorzien. Voor standaard kantoor-applicaties is de metafoor van kantoor-artikelen voor de hand liggend, en rijk genoeg om alle functionaliteit te kunnen representeren. Kantoor-automatisering dient immers precies voor het uitvoeren van de taken van het bureaublad, de typemachine, het plakband, de paperclip, de prullenbak, de kaartenbak, de pen, het stempelkussen, etc.!
Bij didactische Interfaces, waar de gebruiker zich soms noodzakelijk op onbekend terrein begeeft (zie § 5.2.2 A) ligt dat anders. Metaforen moeten hier zorgvuldig gekozen worden, en er is geluk bij nodig om een context te vinden die geschikt is. Soms moet het aantal metaforen dus beperkt worden, en kunnen alleen de belangrijkste taken door icoontjes worden aangegeven.

Metaforen en andere Interface-elementen moeten natuurlijk grafisch worden gerepresenteerd. De weergave van het Interface kan de aandacht veel beïnvloeden. De indeling van het beeldscherm moet overzichtelijk zijn, en informatie moet duidelijk worden gerepresenteerd. De keuze om illustraties te gebruiken kan worden gerechtvaardigd doordat het zonder illustraties vaak moeilijk is om metaforen weer te geven. Een andere reden is dat illustraties zeer effectief kunnen zijn in een didactische Interface, mits de gebruiker gedwongen wordt met de plaatjes te interageren[40]. Ik ga nu nader in op een ander heel belangrijk aspect van de weergave: het gebruik van kleuren.
In het Interface moeten duidelijke kleuren worden gebruikt, met een contrast-werking die de aandacht ten gunste komt. Technieken om de contrast-werking te vergroten zijn

[39] Zie: [Preece1994], blz. 149.
[40] In [Teasley1997] worden experimenten beschreven, die aantonen dat plaatjes bevorderlijk zijn voor het leren, *ook voor informatie die* niet *in de illustratie staat.* "Our first study [over de integratie van tekst en illustraties, K.V.] showed that this style of presentation enhanced learning by 20%. Interestingly, we also found that learning of factual information (which was *not* illustrated by the illustration) was improved just as much as spatial information (which was illustrated by the illustration). This finding is especially encouraging, because it means that the active style increases all learning, not just that directly related to the illustration" (blz. 203-204).

screening, bleaching en *darkening*, waarbij de irrelevante elementen heel vaag orden gemaakt of juist worden geaccentueerd. Ze worden gebruikt om de aandacht op een te manipuleren Interface-element (knop) te richten. [Zhai1997] beschrijft onderzoek naar deze drie methoden. *Screening* en *darkening* zijn het meest effectief: in de experimenten waren de reactie-tijden het kleinst bij het gebruik van deze technieken. Ik gebruik zelf deze technieken, in combinatie met beweging. Een reden daarvoor is dat beweging een toestands-verandering aangeeft, en het zo de gebruiker activeert: hij moet beseffen dat hij iets aan het teweegbrengen is. Daarnaast is het in ons didactische programma zo dat de grafische elementen waarop de aandacht moet worden gericht, metaforen zijn voor abstracte cognitie-elementen (in ons geval argumenten). De beweging die het Interface-element laat knipperen, laat daarmee tevens in gedachten het argument knipperen, en knipperen is een algemene bewegings-metafoor voor ter discussie staan, of om respons vragen (die metafoor van knipperen is heel vertrouwd: elke computercursor is er één).

Naast het contrast tussen de kleuren moet de metaforische werking van de kleuren zelf worden benut. Kleuren en hun associatie (in de Westerse cultuur) zijn bijvoorbeeld:

- groen een correct antwoord / bewerking;
- rood een fout antwoord / bewerking, een waarschuwing;
- blauw een neutrale opmerking;
- oranje een onduidelijkheid;

Een goed voorbeeld over het conflicteren van eisen met betrekking tot kleuren, wordt beschreven door [Vertelney1990]: er moest een Interface worden ontwikkeld die een database met steden moest verbinden met een kaart, waarop de gebruiker door op een stad te klikken informatie over die stad kon krijgen. Het probleem is welke kleuren moeten worden gebruikt. In het voorbeeld is sprake van een monochroom display, waardoor de twee mogelijkheden zijn de steden te representeren door zwarte puntjes op een wit vlak met de zee zwart, of door witte puntjes op een zwart vlak en een witte zee.
Een representatie door zwarte puntjes op een wit vlak maakt de semantische relatie duidelijk: het gaat om de relaties tussen steden, en die moet worden benadrukt, dus op de (voor Macintosh) meest gebruikelijke manier worden weergegeven;
Een representatie door witte puntjes op een zwart vlak maakt de steden beter zichtbaar; ze lijken groter, en komt zeker de werkbaarheid van het Interface ten goede.
Dit dilemma laat duidelijk zien welke afwegingen er moeten worden gemaakt bij het implementeren van het Interface. Vertelney, Arent en Lieberman geven een belangrijk advies: "Graphic designers can and should be involved in conceptual-level Interface design decisions." ([Vertelney1990], blz. 54).
In ons geval leidt deze raad tot een bepaalde manier om de aandacht van de studenten in goede banen te leiden, door de grafische Interface. Toch moet het Interface niet alleen uitgaan van het principe om de aandacht te vangen: de aandacht is in de eerste plaats zaak van de student zelf en is voorondersteld in het programma. Wil didactische software werken, dan moet ze uitgaan van gebruikers die iets willen leren, en dus in eerste instantie met een actieve blik naar het Interface kijken. Een programma dat de aandacht van de student wil vangen, loopt het risico dat de student zich laat meeslepen en (paradoxaal genoeg) een passieve houding krijgt. Het gaat ons erom, de aandacht die de student vanuit zichzelf heeft opgebracht (heel concreet als

hij onderzoekend naar het Interface kijkt) vast te houden, zodat het gebruik van het programma boeiend blijft.

C. conversatie

De conversatie wordt hier beschouwd *voor zover* onafhankelijk van didactiek (A) en aandacht (B). Dat betekent dat we aannemen dat er een didactische inhoud moet worden gecommuniceerd naar een gebruiker die zijn aandacht correct op het Interface heeft gevestigd. Voor die conversatie willen we graag over regels beschikken.
Het artikel "conversation as direct manipulation" geeft een aantal strategieën met betrekking tot conversatie:

> The following strategies are derived from what we know about human/human conversation; 1) don't continue until an understanding that is sufficient for current purposes is reached; 2) assume that errors will happen and provide ways to negotiate them; 3) articulate the answer or response in a way that preserves the adjacency with (and apparent relevance to) the question or command; 4) represent the Interface in a way that invisibly constrains the user to act in ways the system understands and to stick to the application domain; and 5) integrate typed input with pointing and other input/output channels" ([Brennan1990], blz. 404).

Wat is het nut van deze drie strategieën in onze situatie?

1) In een leerproces waar het Interface een *stucturele rol* speelt, dus niet alleen een veraangenaming is van het proces maar niet van dit proces is te onderscheiden, zal het *begrijpen* van het Interface ook samenhangen met het beoogde leerproces. Voor dat leerproces geldt dat een actieve houding van de student vereist is; dit geldt dus in zekere zin ook voor het begrijpen van het Interface. Dat betekent dat de manier waarop het Interface werkt duidelijk moet worden als de student er actief naar vraagt, en niet eerder. De eerste strategie zal bij ons dus geen belangrijke rol spelen;
2) De student zal natuurlijk fouten maken; dit is in de beschrijving van de didactiek reeds besproken. *Errors* zijn fouten die geen enkele positieve neveneffecten hebben. De student kan bij het werken met een web-based applicatie altijd de pagina herladen. Dit zal, tenzij bij structurele problemen bij de client of de server, het probleem oplossen;
3) Dit is voor ons een belangrijke strategie: de student moet zich 'thuis' voelen in de omgeving van het programma. Het moet steeds duidelijk worden gemaakt dat hij en het programma het over hetzelfde onderwerp hebben. Bij *direct manipulation* is dit het geval: er is dan direct duidelijk dat de gebruiker en het programma dezelfhet Interface-objecten manipuleren[41]. Het probleem hiervan

[41] [Golightly1997] toont dit aan. *Direct manipulation* is inderdaad niet persé het beste in een didactische context. Het verschil tussen directe manipulatie (DM) en indirecte manipulatie (IM; een vorm van indirecte manipulatie is linguïstische manipulatie, zie [Faulkner1998], blz. 72), is dat bij de laatste extra Interface-elementen worden gebruikt (waar bijvoorbeeld een puzzle met DM moet worden gemaakt door de stukjes te slepen, gebeurt dat bij IM door bijvoorbeeld eerst op de bron, dan op het doel, en dan op het stukje te klikken en 'voeg toe' te kiezen). Een IM-gebruiker moet meer objecten internaliseren om een interne representatie van het probleem te maken. Dit is een mogelijke verklaring voor het feit dat IM-gebruikers het probleem beter begrijpen: "Having to build this internal

echter vanuit een didactisch perspectief, is dat de gebruiker het Interface-objecten kan gaan identificeren met de leer-objecten (in ons geval abstracte argumenten in een redenering). Hij verliest dan uit het oog waar het ons om gaat: het aanleren van juist de omgang met abstracte argumenten. Om dit te voorkomen moeten de objecten een min of meer arbitrair karakter hebben. Maar dit betekent ook dat het meer moeite kost om expliciet de consistentie tussen het antwoord (het commentaar) van de computer en de invoer van de gebruiker te tonen. Immers, de gebruiker mag niet worden afgeleid door de arbitraire iconen;

4) Deze strategie is voor ons niet zo relevant: voor zover de gebruiker commando's geeft die de computer niet begrijpt (vooropgesteld, dat dat mogelijk is), begeeft hij zich buiten het leerproces, en dus ook buiten onze overwegingen;
5) Ons programma moet een vaardigheid aanleren die in hoge mate tekstueel is, terwijl de studenten een grafische representatie gepresenteerd krijgen in het Interface. De terugkoppeling naar de tekst is dus belangrijk, maar wordt niet op een dwingende manier opgelegd: de student moet er zelf voor kiezen de tekst er weer bij te pakken. Hier geldt dus ook weer de vooronderstelling dat de gebruiker het programma graag *wil gebruiken*. De manier waarop we de gebruikers dit kunnen laten willen, is door *externe* motivatie: ofwel een beloning in termen van studie-punten, ofwel een aantrekkelijkheid van het Interface die het sowieso leuk maakt om het programma te gebruiken. De eerste motivatie kan door een docent gegeven worden; wij proberen op de tweede manier de gebruiker te motiveren.

Het conversatie-aspect belicht dus de kern van de communicatie met het programma. Bij het implementeren van de invoer- en uitvoer-dialogen heb ik van de bovenstaande strategieën vooral de meest relevante, namelijk 3 en 5 in acht genomen[42].
De invoer van de gebruiker moet gevraagd worden waar de gebruiker dat kan verwachten, en er moet bevestigd worden (impliciet of expliciet) dat de gebruiker iets heeft ingevoerd[43]. De invoer wordt in ons programma door directe manipulatie geleverd. De reden daarvoor is dat de enige 'handeling' die de gebruiker in het Interface kan doen is 'een uitspraak doen'. De computer zal hem dan becommentariëren. Als er maar één soort communicatie-handeling is, is het overbodig om een afstand te scheppen tussen Interface-objecten en gebruiker, omdat een dergelijke afstand er vooral toe dient om aan te geven dat er verschillende soorten handelingen kunnen worden uitgevoerd. Een ander aspect van de invoer is *semantische directheid*. Dit is een gevaarlijk aspect, omdat de student zeker niet de semantiek in het Interface moet gaan verwarren met de semantiek van datgene wat hij moet leren (dit kan niet samenvallen, omdat de student juist moet leren *actief* om te gaan met redeneringen, dus in gedachten het Interface steeds een stap voor moet blijven). Dit is de reden waarom de metafoor voor 'een argument aanvoeren' niet is geïmplementeerd als een drag-and-drop mechanisme: dan zou de student denken in termen van Interface objecten die hij met elkaar verbindt door ze te verschuiven, en

---

representation requires deeper processing of the problem domain. This would, in turn, lead to the learning benefits that have been predicted with a more difficult problem solving Interface" (blz. 161). Door de interactie minder direct te maken, zullen gebruikers zich meer concentreren in het domein waarin ze werken, en zich niet laten meeslepen door het Interface.

[42] Zie: § 5.2.2 voor opmerkingen over de concrete dialoog van de Argument Editor.

[43] Zie: [Preece1994], blz. 240.

niet meer in termen van argumentatie. De student moet gedwongen worden (door het Interface) om te beginnen bij een argument, en dan te proberen dat te onderbouwen. Dit is de kern-gedachte van onze Interface-aanpak. Er is dus geen semantische directheid bij de *manipulatie* van het Interface, maar wel bij de *evaluatie* ervan: nadat een student een argument heeft toegevoegd, kan hij rustig kijken wát het Interface-object betekent, zonder dat hij een irriterende indirecte link moet volgen. Deze directheid maakt ook de overeenstemming van de user's goals en de uitvoer van het programma duidelijk. Deze overeenstemming is belangrijk voor het vertrouwen dat de gebruiker in het systeem heeft, en de mate waarin hij zich *in control* voelt. Deze combinatie van indirecte manipulatie en directe evaluatie is naar mijn inzicht een nuttige aanpak voor didactische software.

Er zijn in het Interface nog een tweetal elementen van computer-uitvoer, die niet tot de didactische component of het aandachts-aspect herleid kunnen worden zonder de conversationele crux uit het oog te verliezen: berichten en online help.

Berichten aan de gebruiker moeten, als ze eenmaal de aandacht op zich gevestigd zien, aan de volgende richtlijnen voldoen:[44]

- *een positieve toon.* Berichten moeten de gebruiker stimuleren zelf verder te denken. Dit is zeker in ons programma van belang. Bij een bericht met een negatieve toon, zoals "Invalid password" zal de gemiddelde gebruiker minder efficiënt reageren dan bij "Please try again";
- De gebruiker moet *in control* zijn. De gebruiker moet duidelijk het idee hebben dat *hij* de computer bestuurt en niet omgekeerd. Vooral in een leerproces dat de gebruiker actiever wil maken, is dat heel belangrijk. De gebruiker moet bij een melding altijd alle relevante keuzemogelijkheden aangeboden krijgen. Meldingen die alleen weergeven wat het programma sowieso gaat doen, moeten worden vermeden;
- *een consistent formaat.* Berichten moeten helder en duidelijk worden weergegeven, en de gebruiker moet het idee hebben dat het programma één taal spreekt.

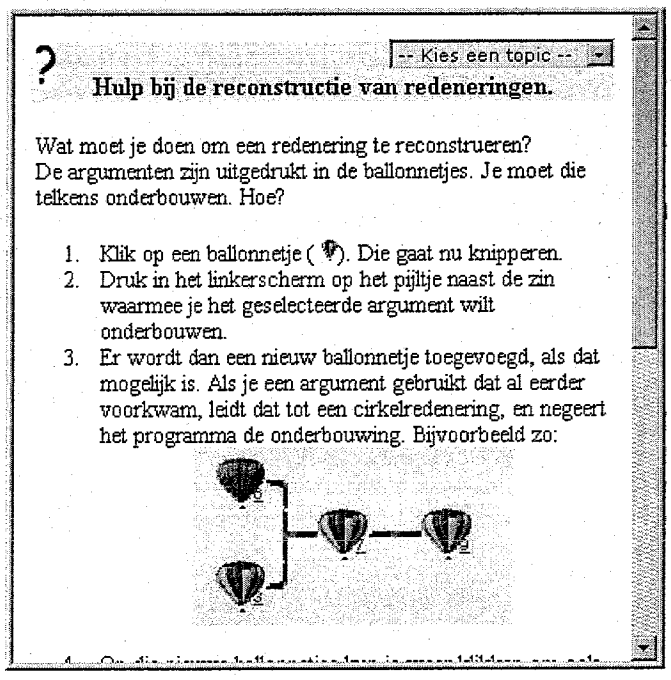
Shneiderman argumenteert tegen het presenteren van computers als mensen ([Shneiderman1998], blz. 380): het onderscheid tussen computers en mensen moet juist duidelijk worden gemaakt. Dit is in de conversatie van belang, omdat de gebruiker de computer niet op alle vlakken als een gelijkwaardige communicatie-partner moet zijn (hier gaan we dus in tegen een bekend communicatie-paradigma voor communicatie tussen mensen). De computer moet op momenten waar de controle geheel bij de gebruiker ligt, niet meer zijn dan een hulpmiddel voor het doel dat de gebruiker wil bereiken. Bovendien is het voor een didactisch proces belangrijk dat de gebruiker zich superieur aan de computer voelt: als de computer alles wat er te bereiken valt zelf zou beheersen, zou de motivatie kunnen wegvallen. Natuurlijk beheerst de computer de vaardigheden die we de gebruiker willen aanleren zelf niet, maar het gaat erom, de gebruiker het idee te geven dat hij vooruitgang boekt. Daarom is het zo belangrijk dat de berichten van de computer duidelijk, positief en eenvoudig zijn gesteld.

[44] Zie: [Shneiderman1998], blz. 378.

De online help-functionaliteit moet aantrekkelijk zijn, en goed toegankelijk. [SelenNicol1990] onderscheidt 5 soorten vragen die de gebruiker aan een help-functie kan stellen:

1) Goal oriented: Wat kan ik met dit programma doen?
2) Descriptive: Wat is dit? Wat kan ik hiermee doen?
3) Procedural: Hoe moet ik dit doen?
4) Interpretive: Waarom gebeurde dat? Wat betekent dat?
5) Navigational: Waar ben ik?

De eerste categorie help moet altijd beschikbaar zijn: als de gebruiker op een *vaste* plek in het Interface de help-functie oproept, moet hem direct duidelijk worden wat hij met dat deel van het programma kan doen. Tevens moet hem ten allen tijde een exacte beschrijving van de functionaliteit van het programma ter beschikking staan, bijvoorbeeld door op de naam van het programma te klikken, die permanent in beeld moet zijn.
De tweede categorie hoeft niet te leiden tot een expliciete help-functionaliteit. Selen en Nicol geven aan dat hulp ook impliciet kan worden gegeven. Deze optie is voor de toegankelijkheid en eenvoud van het programma te verkiezen boven een andere. Concreet betekent dit, dat de vraag "Wat betekent dit Interface-element?" beantwoord moet kunnen worden aan de hand van het element zelf. Het moeten dus metaforen zijn die voldoen aan het in B. beschreven karakter.
Procedurele hulp moet verbonden worden met doel-georiënteerde hulp. De context-gevoelige help-teksten van de argument-editor hebben een standaard opbouw (vergelijk de consistentie-eis aan berichten). Een voorbeeld van zo'n hulpscherm[45] is.



[45] In het programma heet het ook wel "uitleg". De keuze voor deze term boven "hulp" moet benadrukken dat de computer weliswaar niet passief is in het leerproces, maar toch door de gebruiker steeds in die rol gedwongen kan worden.

Algemene richtlijnen voor het implementeren van on-line help, waarop ik me gericht heb, zijn[46]:

- *On-line help should never be a substitute for good Interface design.* De belangrijkste help-taak blijft in het Interface zelf liggen; het komt erop aan zo goed mogelijk betekenisvolle elementen in het Interface te plaatsen. On-line help is altijd een noodoplossing, en het streven moet zijn om hem overbodig te maken, niet om hem een eigen leven te laten leiden;
- *Help should be context-sensitive; it should not take the user away from the task at hand.* Help moet specifiek zijn: toegespitst op waar de gebruiker mee bezig is. Dat kan door een helptekst afhankelijk te maken van de plek in het programma waar de gebruiker zich bevindt, maar ook door de hulp afhankelijk te maken van de stijl van de gebruiker (dat hoort ook bij de context!). Dat laatste kan bijvoorbeeld door een bondige formulering te geven als uitleg, met als aanvullende tekst een voorbeeld.
  Dit is in ons programma het geval. De uitleg die de student krijgt is steeds toegespitst op waar de student zich bevindt. De context-gevoeligheid kan op twee manieren gestalte krijgen in het Interface: 1) steeds andere elementen kunnen worden aangeklikt voor hulp over dat element; 2) één constant zichtbaar element geeft hulp over een steeds wisselende context. Ik heb gekozen voor de tweede mogelijkheid[47], omdat zo de metafoor van een *tutor* beter gerealiseerd wordt: de uitleg-functie kan orden gezien als één wezen (persoon), namelijk diegene die verschijnt als men op X drukt. Als de hulp-functionaliteit als een persoon wordt gezien, kunnen we ook bewerkstelligen dat die zich in principe op de achtergrond houdt, en waar nodig om hulp gevraagd kan worden. Dit bevordert de actieve instelling;
- *Help systems should assist users in framing their questions and provide different help for different questions.* De gebruiker moet bij ons echter *zelf* kunnen formuleren wat hij wil weten. Eerder wees ik op de verstrengeling van het Interface en het leerproces. Dit speelt ook hier een rol, naast het feit dat dit met een andere eis kan conflicteren: de eis namelijk dat de gebruiker zich *in control* moet voelen over hetgeen hij doet. Hoe meer de gebruiker de hulp die hij krijgt kan beheersen, hoe nuttiger hij het vindt. Hulp-onderwerpen moeten dus gemakkelijk te selecteren zijn: bij het oproepen van de hulp-functie verschijnt de uitleg over de huidige context, maar is direct terug te schakelen naar de inhoudsopgave van de hulp. Het programma moet dus niet tè actief de gebruiker assisteren in het stellen van vragen;
- *Help systems should be dynamic and responsive.* Ook de dynamiek van help-systemen is iets waar voorzichtig mee moet worden omgesprongen; en is in ons programma niet echt nodig. In grotere multi-user systemen kan het handig zijn als gebruikers zelf tekst kunnen toevoegen aan de hulp-functie, maar dit past niet echt in een *educational setting*. Als vuistregel moet gelden: "gebruik dynamische help voor dynamische functionaliteit, en statische help voor statische functionaliteit." Functionaliteit die een eeuwenoud concept

[46] Zie: [SelenNicol1990], blz. 153 en [Kearsley1988], blz. 76.

[47] De student kan op het vraagteken klikken en krijgt dan context-gevoelige uitleg in een nieuw venster. De tutor-tekst (inhoudelijke hulp) staat onder in het venster. Zie: [Kearsley1988], blz. 76: "Help messages should be displayed in windows overlapping the application or in message areas at the bottom of the screen."

(vaardigheid in leren redeneren) wil aanleren, moet eenduidig en statisch worden uitgelegd. Dat wil natuurlijk niet zeggen dat er geen cross-references en short-cuts zijn; daarvan moet juist veel gebruik worden gemaakt. Het gaat erom dat de gebruiker het idee krijgt dat hij uitleg krijgt over een stuk *vaste* functionaliteit.

- *Users shouldn't need help to get help.* (!)

5.3 De implementatie van de concrete dialoog

Na het bespreken van de algemene vuistregels op de gebieden didactiek, aandacht, en conversatie, waarvan ik ben uitgegaan bij het ontwikkelen van het User Interface, beschrijf ik hier waartoe dat geleid heeft in de concrete dialoog met de gebruiker. Ik beschrijf niet uitputtend de dialoog waar het om gaat, maar belicht Interface-technisch interessante aspecten.

Drie typen gebruikers

Het onderscheid tussen verschillende typen gebruikers is eigenlijk geen onderscheid dat thuishoort in de implementatie, maar in het systeem-model. Toch heb ik ervoor gekozen om het hier te beschrijven, want de concrete aspecten van de verschillende typen gebruikers zijn Interface-aspecten. In het modelleren van het systeem is het een onnodige complicatie om uit te gaan van meerdere gebruikers-typen; daarvan maakt een goed model abstractie. Het gaat in de modellen eigenlijk om *rollen*, en niet om concrete gebruikers. In ons programma kan een docent ook goed een student-rol vervullen. Precies bij dit *vervullen* van een rol door verschillende typen gebruikers komen implementatie-aspecten kijken die hier relevant zijn.
Christine Faulkner onderscheidt drie typen gebruikers ([Faulkner1998] blz. 88):

1. "novice users";
2. "knowledgeable intermittent users"; Deze hebben een zeer goede semantische kennis, en zijn experts op het probleemgebied waar het systeem voor is. Zij hebben goede handleidingen nodig voor de technische mogelijkheden.
3. "expert users". Deze hebben een goede kennis, zowel van de semantiek, als ook van de syntax van het systeem.

Bij het implementeren van mijn programma ben ik ook uitgegegaan van een driedeling van de typen gebruikers, maar net iets anders:

1. leergierige studenten, die weinig computer-ervaring hebben, maar er de tijd voor willen nemen om iets van het programma te leren;
2. studenten, die zeer snel de werking van het programma overzien, en zo snel mogelijk willen vernemen wat het programma hen kan bieden voor de semantiek van het probleem-domein;
3. studenten, die zo snel mogelijk van het programma af willen.

Voor deze drie typen studenten zijn er verschillende aspecten van het Interface gemaakt. Het eerste type zal veelvuldig gebruik maken van de context-gevoelige uitleg-functie van het programma, en langzaam interpreteren wat de respons betekent die hij/zij krijgt. Het tweede type zal de instellingen zo veranderen dat hij alles in één

oogopslag ziet, en gebruik maken van de steekwoorden. Hij zal snel door het programma heenlopen, en de shortcuts gebruiken (bijvoorbeeld het menu dat verschijnt door op de rechter muisknop te drukken). Het derde type gebruiker zal proberen zo snel mogelijk alles weg te klikken. Hij wordt echter gewaarschuwd door een melding, die zegt dat hij eerst de zin moet lezen. Dit is mogelijk, omdat het lezen van een zin typisch meer kost dan een seconde[48]. Als er binnen één seconde na het aanvoeren van een zin een nieuwe zin wordt aangevoerd, reageert het programma:

Didactiek van het leren argumenteren

De dialogen van het programma spitsen zich toe op het leerproces van de student. Hierbij heb ik de boven beschreven vuistregels in acht genomen. De regel *give information in advance* heb ik toegepast in het algemene idee dat ten grondslag ligt aan het programma. Het leerproces is zoals bekend opgesplitst in twee delen; in het eerste onderdeel herkent de student kernzinnen uit en tekst. In het tweede deel moet hij die kernzinnen met elkaar verbinden. Het eerste onderdeel dient er dus voor om de student vertrouwd te maken met een nieuw informatie-domein. Verder is ervoor gezorgd dat er verstandig wordt omgegaan met de informatie-vraag van de student. Als de student redelijkerwijs op een punt in het programma informatie nodig heeft, is die in sommige gevallen gepresenteerd (als de student een incorrect antwood heeft gekozen, krijgt hij dat terug met eventueel commentaar), maar in andere gevallen bewust achterwege gelaten omdat de student steeds een actieve rol moet blijven spelen[49].
Om de positieve rol van het Interface in het leerproces uit te werken, heb ik het Interface voorzien van een eenduidige identificatie van de handeling *het aanvoeren van een argument* met een Interface-handeling, namelijk het selecteren van een knoop en het klikken op een pijltje naast een mogelijk argument ervoor. De student kan het natuurlijk van tevoren op papier uitwerken, maar het werken met het Interface heeft toch een toegevoegde waarde. Daarvoor is ook de functionaliteit toegevoegd die de student de tot op een gegeven moment opgebouwde argumentatie in tekstuele vorm te

[48] Eigen onderzoek heeft dat uitgewezen. Zelfs één van de korste zinnen, "The defendants are not guilty" kost meer dan een seconde om te lezen, ook als je hem al een aantal keer hebt gelezen. Dit is een manier om studenten te dwingen de zinnen waarmee ze werken goed te bekijken en niet in het wilde weg te klikken.
[49] Er wordt bijvoorbeeld niet 'handig' weergegeven welke kernzinnen de student reeds heeft aangevoerd, omdat dit de actieve instelling van de student kan belemmeren. De student moet immers de zinnen goed lezen en blijven lezen. Als met het vorderen van het leerproces het aantal te kiezen zinnen afneemt, krijgt het steeds meer een legpuzzel-karakter. Dit willen we voorkomen: het moet juist zo zijn dat de student met het vorderen van het leren argumenteren steeds meer in de gaten krijgt wat zijn handelingen in argumentatieve zin betekenen.

tonen. De student beschikt dan over een nette, korte samenvatting van de tekst die hij moet bestuderen, en dat is zeer welkom bij het studeren[50].
Beweging is aan het Interface toegevoegd om *richting* aan te duiden. In het geval van de hulp-functie: die wordt door een vliegtuigje voorgesteld.

Dit is het enige permanent bewegende object in het Interface, en vraagt zo om speciale aandacht. Bij het implementeren van de metafoor voor het aanvoeren van argumenten,
is gebruik gemaakt van een bewegend pijltje (als de muis eroverheen wordt bewogen, klapt het pijltje uit in de richting van de reconstructie). Zo wordt duidelijk welke objecten in welke richting 'verplaatst' moeten worden.
Het Interface als *coach*: door het benadrukken van de actieve instelling van studenten functioneert het programma vanzelf als coach. Dat wil zeggen dat de student het beeld heeft van een op de achtergrond werkzame controlefunctie.

Aandacht voor het Interface

De 'aandacht' beschreven we als de primaire gewaarwording van de gebruiker met het Interface. Belangrijke aspecten waren metaforen en kleuren.
Het programma gebruikt een aantal metaforen:
bij het lezen van een alinea wordt de tekst geel gearceerd. De metafoor verwijst naar het arceren dat de studenten gewoon zijn bij het lezen van papieren teksten. De aandacht wordt zo goed gericht op de alinea;

Een schaar dient voor het verwijderen van een ballonnetje. Een duidelijke en onbetwistbare metafoor;

Een schoolbord dient voor het verwijzen naar de tekstuele weergave van de reconstructie. Het programma is in opzet een innovatie ten opzichte van het schoolbord-principe van lineaire presentatie van leerstof. Het schoolbord keert hier even terug om die informatie te presenteren, die de student *zelf* in het interactieve proces heeft verzameld: daarmee is het schoolbord weer waar het thuishoort, gedegradeerd van stenen tafel tot hulpmiddel;

Een boekrol dient om terug te verwijzen naar de oorspronkelijke tekst. Een duidelijke metafoor waarover weinig verwarring mogelijk is. Hij onderscheidt zich van het schoolbord door zijn archaïsch karakter, en daarvoor is gekozen omdat hij terugverwijst naar wat vooraf ging aan de analyse: de oorspronkelijke tekst;

[50] De studenten die deelgenomen hebben aan de testsessies, gaven er blijk van graag te studeren aan de hand van zulke handzame fragmenten. In het programma is op die manier een drijfveer ingebouwd om de student zijn reconstructie-pogingen te laten voortzetten.

Vier kleine icoontjes zijn gebruikt voor de weergave van de tutor of coach-functie van het programma. Hierin worden de in §5.2.1 B genoemde kleuren gebruikt. Het vinkje is een conventioneel teken voor een correct antwoord; de student hoeft niets extra te doen. Het uitroepteken waarschuwt de student van iets waarop het moet letten, maar wat niet kritiek is (blauw staat voor neutraal). Het kruisje staat voor een pertinente fout, waar de student actie moet ondernemen. Het vraagteken duidt een onvolledigheid aan, en vraagt de student dus om een nuancering.

Wat *kleuren* in het algemeen betreft: het programma maakt gebruik van lichte kleuren en duidelijke contrasten. Belangrijke en onbelangrijke gegevens in het Interface kunnen op die manier worden gescheiden.

Conversatie: de dialoog tussen het programma en de gebruiker

Conversatie is besproken in § 5.2.1 C. Specifiek voor ons programma geldt nog een belangrijke vuistregel: de icoontjes waarmee de reconstructie wordt uitgebeeld moeten *willekeurig* zijn. Een student moet kunnen kiezen uit willekeurige icoontjes omdat daarmee duidelijk wordt dat de icoontjes geen invloed hebben op de redenering. Dit lijkt een heel voor de hand liggend inzicht, maar in de praktijk gaan mensen al snel over tot Interface-manipulatie waarbij ze hun eigen handelingen beschrijven in termen van het Interface-objecten. Dus "je moet ballon 1 daar en daar naartoe slepen". Dit is niet wat we willen bevorderen. Er moet over het Interface gesproken worden in termen van de *abstracte* elementen die erdoor worden gerepresenteerd, omdat dat het uiteindelijke leerdomein is. Om de associatie van die elementen met icoontjes (metaforen) te ontkoppelen maken we de icoontjes willekeurig naar vrije keuze van de gebruiker. Daarmee wint het programma aan aantrekkelijkheid (tijdens een voetbal-toernooi kunnen voetballen worden gebruikt om de knopen in een boom aan te duiden etc.). Maar waarom is er eigenlijk een onderscheid gemaakt tussen het Interface-objecten en de leer-elementen? Omdat de leer-elementen (de kernzinnen en hun verbanden) niet steeds gelezen worden. Bij het verbinden van kernzinnen is het ondenkbaar dat ze steeds helemaal opnieuw worden herlezen. De student zal ze onthouden, en dus ook met een associatie gaan verbinden. Het gaat er ons om, dit associatieproces zo vruchtbaar mogelijk te doen verlopen. Associatie met een plaatje op het scherm heeft geen didactische kwaliteit; de associatie moet zodanig gebeuren dat de student zich ervan bewust is dat het om een *willekeurige* relatie gaat. Daarbij kunnen twee methodes worden onderscheiden:

- de kernzin wordt geässocieerd met een een kernachtige afkorting. Een typisch voorbeeld is het representeren van de zin "The defendants are not guilty" met de afkorting "Not Guilty". Sommige studenten werken graag met deze methode, waarbij ze direct kunnen zien om welke zin het gaat;
- de kernzin wordt geässocieerd met een getal. De associatie is dan volstrekt willekeurig. Sommige studenten geven hier de voorkeur aan, omdat ze verward worden van weer nieuwe afkortingen en gevleugelde woorden.

Deze associatie is het uitgangspunt van de student: van daaruit zal hij argumenten gaan aanvoeren. Het programma zal daarop reageren, door als een *coach* of *tutor*

opmerkingen te maken over wat de student heeft gedaan. Deze opmerkingen moeten duidelijk zijn, en vooral constructief. De Argument Editor geeft dit commentaar:

- "Je moet zin X met nog meer argumenten onderbouwen".
  De student wordt aangespoord om verder te gaan op de weg die hij is ingeslagen;
- "Je hebt een (veel) te grote stap gemaakt in de onderbouwing van zin X. Voer eerst een ander argument aan".
  De student wordt aangespoord de tekst nog eens goed te lezen, en eerst een ander argument aan te voeren. Dit moet handmatig gebeuren, om de actieve instelling van de student te bevorderen (zie § 5.2.1 A voor meer over die beslissing).
- "Je hebt zin X met teveel argumenten onderbouwd. Haal er enkele weg."
- "Er zit een goed argument in de onderbouwing van Zin X maar ook foute.";
- "De onderbouwing van zin X leidt tot een cirkelredenering. Probeer opnieuw".
  Argumenten die leiden tot een cirkelredenering worden niet toegevoegd; de student moet het met deze waarschuwing doen. Hiermee wordt de associatie bewerkstelligt van een cirkelredenering met iets ongeoorloofds.
- "Je kunt geen onderbouwing toevoegen aan de aanname X";
  Een student die probeert om een aanname nog verder te onderbouwen, krijgt deze melding. De term 'aanname' werkt beter dan 'vooronderstelling', bleek uit de tests met studenten;
- "Je hebt zin X al met zin Y onderbouwd!";
- "Zin X is correct onderbouwd. Ga zo door!";
  De student moet doorgaan met zinnen onderbouwen, tot de reconstructie voltoid is. Hij zal dan door een animatie worden onderbroken, en, na gewezen te zijn op zijn 'eindresultaat' in tekstuele vorm, terug worden geloodst naar het hoofdmenu;
- "Zin X is niet correct onderbouwd.".

Dit is het belangrijkste wat de gebruiker ziet van de dialoog met het programma. Deze zinnen worden op een vertrouwde standaardplek onder in het scherm weergegeven.

6. Conclusie

De Argument Editor werd ontwikkeld om studenten te leren argumenteren. Het ging om het aanleren van een natuurlijke omgang met propositionele logica, maar zonder de studenten met een formeel model op te zadelen. Dat betekent dat we niet uitgaan van een argumentatie-model zoals dat van Stephen Toulmin, en evenmin van een specifiek juridisch argumentatie-model. De vaardigheid die we de studenten willen aanleren, is een vaardigheid die in de omgang met geavanceerde modellen eigenlijk al is voorondersteld.
Het gevolg daarvan is dat er een zware taak op het User Interface komt te liggen, namelijk die van het geloofwaardig en aantrekkelijk representeren van de redeneer-situatie. Uit test-sessies bleek dat de meeste studenten het liefst een grafische Interface hadden, waarbij het argumentatie-proces dat wordt gesimuleerd of gereconstrueerd overeenkomt met handelingen in het Interface. Concreet betekent dit dat het User Interface muis-handelingen het aanzien van argumenteer-stappen moet geven. Er werd beargumenteerd dat er daarvoor gebruik moet worden gemaakt van *indirecte manipulatie* i.p.v. de meer gebruikelijke vorm van *directe manipulatie*. Het Interface moet grafisch aantrekkelijk zijn, om de aandacht van de studenten te blijven *vasthouden*. Een actieve, leergierige instelling wordt dus verondersteld als het programma wordt gestart.
De Argument Editor is op basis van deze ideeën over didactiek en Interface ontwikkeld, en aangeboden als hulpmiddel voor studenten die Rechtsfilosofie A volgen. De resultaten in termen van verbetering van logische argumentatie-vaardigheden kunnen pas worden geëvalueerd in een later stadium, en kunnen hier dus nog niet worden gepresenteerd.

Literatuur

[Baeten1990] J.C.M. Baeten & W.P. Weijland, Process algebra, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press 1990.

[Barwise1998] Jon Barwise and John Etchemendy, *Computers, visualization, and the nature of reasoning.* In: The Digital Phoenix: How Computers are Changing Philosophy. T. W. Bynum and James H. Moor, eds. London: Blackwell, 1998, blz. 93-116.

[Bouwer1999] Anders Bouwer, *ArgueTrack: Computer Support for Educational Argumentation*. In: Proceedings of the AI-ED '99 conference, 19-23 July 1999, Le Mans, France.

[Brennan1990] Susan Brennan, *Conversation as Direct Manipulation: An Iconoclastic View*. In: The art of human-computer Interface design. Edited by Brenda Laurel. Addison-Wesley, 1990. Blz. 383-392.

[Converse2000] Tim Converse and Joyce Park, *PHP4 Bible*. Troutworks, 2000.

[Cooper1995] Alan Cooper, *About face*. IDG Books World Wide, 1995.

[Cox1993] Kevin Cox and David Walker, *User Interface Design*. Simon & Schuster (Asia) Pte. Ltd., 1993.

[Dietz1991] Jan L. G. Dietz, *A communication oriented approach to conceptual systems modelling.* In: Dynamic Modelling of Information Systems. Edited by H. G. Sol en K. M. van Hee, North-Holland, 1991, blz. 37-60.

[Dix1998] Alan J. Dix, Janet E. Finlay, Gregory D. Abowd, Russell Beal, *Human-Computer Interaction*. 2nd Edition, Prentice Hall, 1998.

[Dur1991] Remko C. J. Dur, *Dynamic Modelling for analysis and design of office systems.* . In: Dynamic Modelling of Information Systems. Edited by H. G. Sol en K. M. van Hee, North-Holland, 1991, blz. 303-321.

[vanEemeren1981] F.H. van Eemeren, R. Grootendorst en T. Kruiger, *Argumentatietheorie*. Het Spectrum, 1981.

[vanEemeren1984] F.H. van Eemeren, R. Grootendorst en T. Kruiger, *Argumenteren*. Het Spectrum, 1984.

[Erickson1990] Thomas D. Erickson, *Working with Interface Metaphors*. In: The art of human-computer Interface design. Edited by Brenda Laurel, Addison-Wesley, 1990, blz. 11-16.

[Faulkner1998] Christine Faulkner, *The essence of Human-Computer Interaction*. Prentice Hall, 1998.

[Fuller1949] Lon Fuller, *The case of the speluncean explorers*. In: *Harvard law review*, Vol. 62, No. 4, februari 1949.

[Golightly1997] D. Golightly en D. Gilmore, *Breaking the rules of direct manipulation*. In: Human-Computer Interaction INTERACT '97. Edited by Steve Howard, Judy Hammond and Gitte Lindgaard. Chapman & Hall, 1997, blz. 156-165.

[Hall1998] Marty Hall, *Core Web Programming*. Prentice Hall, 1998.

[vanderHarst1997] Guido van der Harst en Rob Maijers, *Effectief GUI-ontwerp*. Academic Service, 1997.

[Kassenaar1997] Peter Kassenaar, *HTML voor gevorderden*. Academic Service, 1997.

[Kassenaar1997b] Peter Kassenaar, *Javascript 1.2*. Academic Service, 1997.

[Kearsley1988] Greg Kearsley, *Online Help Systems*. Ablex Publishing Corporation, 1988.

[Lodder1999] Arno R. Lodder and Bart Verheij, *Computer-Mediated Legal Argument: Towards new Opportunities in Education*. The journal of information, law and technology, 1999.

[Nicol1990] Anne Nicol, *Interface for Learning: What Do Good Teachers Know That We Don't?* . In: The art of human-computer Interface design. Edited by Brenda Laurel. Addison-Wesley, 1990, blz. 113-122.

[ONeil1994] Patrick O'Neil, *Database*. Morgan-Kaufman, 1994.

[Preece1994] Jenny Preece e.a, *Human-computer interaction*. Addison-Wesley, 1994.

[Salomon1990] Gitta Salomon, *New Uses for Color*. In: The art of human-computer Interface design. Edited by Brenda Laurel. Addison-Wesley, 1990, blz. 269-278.

[SelenNicol1990] Abigail Sellen and Anne Nicol, *Building User-centered On-line help*. In: The art of human-computer Interface design. Edited by Brenda Laurel. Addison-Wesley, 1990, blz.143-152.

[Shneiderman1998] Ben Shneiderman, *Designing the User Interface*. 3rd. Edition. Addison-Wesley, 1998.

[Teasley1997] B. Teasley, K. Instone, L.M. Leventhal en E. Brown, *Effective illustrations in interactive media: what works?* In: Human-Computer Interaction INTERACT '97. Edited by Steve Howard, Judy Hammond and Gitte Lindgaard. Chapman & Hall, 1997, blz. 197-204.

[Tudor1995] D. J. Tudor en I. J. Tudor, *Systems analysis and design. A Comparison of Structured Methods.* NCC Blackwell, 1995.

[UML] http://www.rational.com/uml/resources/quick/uml_poster.jsp

[Vertelney1990] Laurie Vertelney, Michael Arent, and Henry Lieberman, *Two Disciplines in Search of an Interface: Reflections on a Design Problem.* In: The art of human-computer Interface design. Edited by Brenda Laurel. Addison-Wesley, 1990, blz. 45-56.

[Weinschenk1997] Susan Weinschenk, Pamela Jamar, Sarah C. Yeo, *GUI-ontwerp*. Academic Service, 1997.

[Zhai1997] S. Zhai, J. Wright, T. Selker and S-A. Kelin, *Graphical means of directing users' attention in the visual Interface*. In: Human-Computer Interaction INTERACT '97. Edited by Steve Howard, Judy Hammond and Gitte Lindgaard. Chapman & Hall, 1997, blz. 59-66.

Bijlage 1: evaluatie van het prototype voor het Interface

Het spreekt voor zich dat het programma voordat het wordt opgeleverd, aan de toekomstige gebruikers moet worden voorgelegd. Daarvoor is een prototype gemaakt, en hebben studenten dat in verschillende evaluatie-sessies beoordeeld. Deze bijlage beschrijft die evaluatie. De keuze voor het *moment* van evaluatie[51], namelijk in de ontwerp-fase lag voor de hand. De gebruikers tonen weinig begrip voor een systeem-analyse die nog moet beginnen, en aan de andere kant, ze willen ook geen programma testen waarop ze nauwelijks meer invloed kunnen uitoefenen.

Concrete gegevens van de evaluatie:

Totaal aantal test-personen	7
Data van de tests	31-1-2002 18-2-2002 27-2-2002 06-3-2002
Duur van de tests	iets minder dan een uur per keer
Soort test	demonstratie van een prototype, met de bedoeling om zoveel mogelijk opmerkingen de horen.

Tijdens een drietal testsessies hebben representanten van de doelgroep [52] een prototype[53] geëvalueerd voor het Interface van ArgEdit. Zij kregen de opdracht om zoveel mogelijk opmerkingen te maken over het Interface. De evaluatie heeft dus gegevens opgeleverd die variëren van esthetische voorkeuren tot inzichten in het leerproces. De opmerkingen van de studenten zijn hier gerangschikt en van commentaar voorzien.
De opmerkingen zijn onderverdeeld in drie groepen. De opmerkingen die direct betrekking hebben op het leerproces, de opmerkingen die met de visualisatie van het

[51] Zie [Faulkner1998], blz. 112. Zij onderscheidt de fasen van systeem-analyse, ontwerp en pre-productie als mogelijke fasen om evaluaties af te nemen.
[52] De primaire doelgroep is jongerejaars rechten-studenten, die in het kader van een cursus rechtsfilosofie teksten moeten analyseren en zich daarbij moeten concentreren op de logische structuren. De ervaring met computers is over het algemeen zeer wisselend bij rechten-studenten. Bij het groepje test personen waren zowel 'digibeten' als mensen die hun eigen websites onderhouden. Een interessant resultaat van de test is dat alle testpersonen de voorkeur gaven voor een direct, liefst grafisch, overzicht over het gehele gebruikers-proces (zie hoofdstuk 5.2)
[53] Het prototype dat de test-personen kregen te zien is in dezelfde taal geschreven als het uiteindelijke programma, in PHP dus. Het prototype gaf een idee van de scherm-indeling. De test-personen hadden geen problemen om het prototype als een echt programma te beschouwen. De keuze voor het evalueren met de computer in plaats van op papier laat zich onderbouwen door de observatie dat veel studenten een drempel ervaren bij het gebruiken van een computer, die op papier niet te achterhalen is. Daarenboven is bij het soort programma dat we ontwikkelen de computer vooral een *extra* hulpmiddel, en dus moet bij een evaluatie al duidelijk zijn dat de student het *naast* zijn papieren kan gebruiken. Dat zouden we uit het oog verliezen als we het Interface-ontwerp op papier zouden evalueren (overigens zijn er na de evaluatie wel schetsen op papier gemaakt van het uiterlijk van de uiteindelijke Interface).

leerproces hebben te maken, en de opmerkingen van meer algemene aard over het uiterlijk van het programma. De inhoud van de tweede tabel zijn dus de opmerkingen over het Interface die *alleen* op ArgEdit van toepassing zijn, en in de derde tabel staan die opmerkingen die ook op andere applicaties van toepassing *kunnen* zijn.
De opmerkingen zijn waar mogelijk per tabel gerangschikt naar onderdeel[54], en per groep opmerkingen is commentaar toegevoegd voor het werkplan van een mogelijke oplossing of verbetering van dit aspect.
Bij de opmerkingen waarvoor dat relevant is, is ook aangegeven wat haar prioriteit voor de gebruiker is in kolom (P) en wat haar impact voor het programma is in kolom (I). Voor beide wordt een schaal van 1 tot en met 4 gebruikt (1 het hoogste). Zo wordt de relevantie van de opmerking voor zowel de gebruiker als het programma gemeten.

Tabel 1: Opmerkingen over het leerproces en zijn structuren:

Opmerking	P	I
De eerste fase: het identificeren van de kernzinnen		
* In de eerste fase willekeurige volgorde gebruiken van alternatieven	4	4
* De tekst vooraf en tijdens de eerste fase goed laten analyseren	2	4
* De alternatieven die fout zijn goed becommentariëren	2	4
Evaluatie: de eerste fase is zonder veel problemen geaccepteerd. Het presenteren in een willekeurige volgorde van alternatieven, en het geven van veel didactisch commentaar behoort bij de fine-tuning van het programma.		
Het redeneren		
* Goed uitleggen wat er moet gebeuren	1	2
* Dwing de student de tekst te lezen	3	3
* De aandacht op het zinsmatige redeneren houden	1	4
Evaluatie: de student moet niet worden afgeleid van zijn kerntaak. De belangrijkste maatregel daarvoor is het geven van een overzicht in één keer. De student kan zich op zinsmatig redeneren controleren met behulp van grafische weergave, zolang er maar een duidelijk overzicht is (zie verder).		
Het controleren		
* Bij een formule van de vorm A->C terwijl het moet zijn A&B->C ook aangeven	2	4
* Als formules feitelijk equivalent zijn, dat dan ook aangeven	3	4
* Criterium voor controle: zo min mogelijk zinnen ingevoerd	4	3
* Criterium voor controle: zo min mogelijk keer gecontroleerd	4	3
* Vereiste voor controle: maximaal 10 keer op control gedrukt	4	3
* Als er wordt gecontroleerd, niet aangeven wat er goed is	4	3
* Als er wordt gecontroleerd, laten zien hoeveel procent goed is	4	3

[54] Sommige onderdelen kunnen nog niet worden onderscheiden in een prototype omdat hun relevantie pas bij de evaluatie blijkt.

Evaluatie: controleren is een belangrijk hulpmiddel om de student mee te laten leren. Het kan ook worden gebruikt voor een spel-element (de opdracht zo min mogelijk het programma te laten controleren). Echter: de computer moet vooral gezien worden als een ondersteuning en de student moet zich dus vrij voelen om te kunnen controleren. Ook kunnen de eisen van het zwaartepunt bij de controle leggen en van een totaal-overzicht niet helemaal met elkaar in verband worden gebracht: controle hoort meer thuis bij een opbouwend werkende reconstructie. Toch kan het gebruikt worden, maar dan als extra hulpmiddel. Een geld metafoor kan gebruikt worden bij de controle: de gebruiker moet 'betalen' voor een controle, waarmee wordt aangegeven dat controle handige informatie kan opleveren. De controle van de vorm van de eerste opmerking leert dat er een variabele is vergeten en brengt de aandacht van de student daarop.

Tabel 2: Opmerkingen over de visualisatie van die structuren:

Opmerking	**P**	**I**
Overzichtelijkheid, metaforen		
* Geen legenda gebruiken, maar hele zinnen kiezen en invoeren	2	2
* de zinnen zelf aankruisen ipv met codes werken;	2	2
* Duidelijkere metafoor voor de legenda (als extra)	4	4
* Duidelijk onderscheid tussen knoppen 'implies' en 'GIVEN'	3	4
* Een expliciete RULE knop	3	4
* Evt. metafoor met hamer voor APPLY	4	4
* de APPLY functie bij het klikken op de regels zelf	3	4
* Het woordje 'implies' gebruiken in plaats van een pijl	3	4
* Het IF...THEN scherm gebruik in de 'logische rekenmachine' / kladblok	2	2
* '(' en ')' weg	4	4
* Een duidelijker onderscheid tussen regels en aannames	3	4
* Bij een ongeldige aanname moet het ballonnetje verdwijnen	3	4
* Een 'restart' ipv 'reset'	3	4
* Kladblok automatisch opschuiven naar huidige positie;	3	4
* Kladblok moet op kladblok lijken	3	4
* De kladblokmetafoor moet als hulpje blijven	2	1

Evaluatie: de metaforen moeten duidelijk zijn. Vooral het werken met een popup-legenda is verwarrend. Bij het gebruik van codes mogen die codes geen 'logica' bevatten die vaag is voor de student, zoals afkortingen, maar moeten het gewoon codes zijn van de vorm A1...Ax (zie ook opmerkingen over algemene Interfaces). Het aantal te introduceren termen moet beperkt blijven (zo moeten de haakjes die eigenlijk bij een logische rekenmachine horen er niet bij omdat ze alleen verwarring zaaien). En waar termen worden geïntroduceerd, moet het onderscheid heel duidelijk zijn, zoalshet onderscheid tussen GIVEN en implies en het onderscheid tussen regels en aannames.
De metaforen mogen gekozen worden uit het werkdomein van de gebruiker, zoals kladblok en (rechters-) hamer. Er moet echter wel duidelijk blijken dat ze hier alleen een bepaald aspect moeten symboliseren. Met andere woorden: door het gebruik van metaforen zoals de kladblokmetafoor mogen geen verwachtingen worden geschept die het programma niet waarmaakt ("zo gebruik ik een kladblok toch niet.")

De boomstructuur

* Het idee van een boom die je in zijn geheel ziet en moet invullen is prima	1	1
* Een lege boom tekenen die de student zelf moet invullen	1	1
* De student een boom zelf laten opbouwen; slechts één knoop geven. De gemakkelijke versie moet niet worden vrijgegeven: het moet niet te makkelijk worden; de moeilijke versie moet centraal staan, maar goed uitgelegd worden.	1	1
* In de op te bouwen boomstructuur zijn twee manieren om zinnen toe te voegen; die komen overeen met wat in de argumentatieleer 'nevenschikking' en 'onderschikking'. Zorg dat deze twee manieren intuïtief duidelijk zijn.	3	3
* Boom tekenen: onder in het scherm links	4	4
* Pijlen in de boomstructuur zijn niet nodig	4	4
* Als het 'spel' afgelopen is, presenteer nog eens de gemaakte correcte boom, en de originele tekst ernaast, als 'beloning'	3	4
* De boom kan worden aangepast door op een knoop te klikken, waarbij je direct je mogelijkheden in beeld moet krijgen.	2	3
* Er kan worden aangegeven, met een ballonnetje bijvoorbeeld, welke zinnen er al zijn gebruikt. Als een zin vaker moet worden gebruikt, plaats er dan meerdere ballonnetjes voor	4	4
* Voor een moeilijkere versie zou de structuur van de boom nog onvolledig kunnen zijn.	3	3

Evaluatie: de keuze is gemaakt voor de boomstructuur. Deze geeft de student een direct overzicht en maakt meteen duidelijk wat er van hem verwacht wordt. Hij moet de boom opbouwen met de zinnen uit de eerste fase. Daarbij klikt hij op de knopen en hangt er nieuwe knopen aan. De boom kan grafisch aantrekkelijk worden weergegeven, eventueel met bewegende beelden. Door hem de boom te laten opbouwen, leert de student dus (in de metafoor die de doelgroep uitstekend blijkt te begrijpen) hoeveel variabelen in het linkerlid moeten staan van de formule. Wanneer het te moeilijk blijkt, kan er een toegang zijn tot een gemakkelijkere versie, waarbij de gehele boom al is gegeven. De prioriteit heeft echter de moeilijkere versie.
De boomstructuur is dus het centrale scherm in het programma, waarmee de student zich goed moet kunnen identificeren (hij moet het idee hebben dat hij daar "is"). De kladblokmetafoor is een hulpmiddel dat kan worden opgeroepen. Het moet heel duidelijk zijn hoe de student weer terug kan komen bij de boom.

Tabel 3: Algemene opmerkingen over het uiterlijk:

Opmerking	P	I
Navigatie		
* Een standaard-balkje om te surfen	4	4
* Een 'back' knop bij een invoerveld als het idee wordt gewekt dat er iets ongedaan gemaakt kan worden.	4	4
* Gebruik grotere nummers, die voor iedereen gemakkelijk leesbaar zijn	4	4
* 'choose' boven de selectielijst	4	4
* De help-functie moet een vaste plek op het scherm zijn; er moet een vriendelijke, uitgebreide verhelderende tekst over de reconstructie worden getoond.	1	3
* Directe navigatie boven in beeld	3	3
Evaluatie: De navigatie moet zoveel mogelijk aansluiten bij de navigatie die de student gewend is. Schuifbalken, input-velden en selectielijsten moeten niet anders zijn. De student moet ze snel op het scherm weten te lokaliseren.		
Venster-opbouw en esthetische aspecten		
* Het bovenste frame met de titel en snelle navigatie kleiner	4	4
* Het gebruik van popup vensters is niet intuïtief	4	3
* Commentaar op een vaste plaats in beeld	4	3
* Betere, vertrouwdere achtergrond gebruiken	3	4
* Nieuwe logo gebruiken; opzet naar andere KUB websites en courseinfo; blauw.	4	4
* Strakker en formeler uiterlijk	4	4
* Eén scherm waarop alles zichtbaar is; dat geldt ook meer algemeen.	2	1
* Er is geen gevaar voor het afgeleid worden van de tekst bij het gebruik van een grafiek	4	4
Evaluatie: de opbouw van de vensters van ArgEdit moeten natuurlijk zijn, en herkenning geven bij de student. De student werkt vooral met programma's als Course Info; de opbouw kan zich daarop baseren. Het programma moet professioneel en actueel ogen. Het moet direct een overzicht geven van de mogelijkheden en doelstellingen.		
Taal		
* Liever geen twee talen door elkaar gebruiken. De hele Interface in het Engels.	4	3
* Bij twee talen in ieder geval woorden die problemen geven expliciet in een hint vermelden.	4	4
* Codes voor de zinnen zijn oke, maar er moet geen andere logica in zitten dan een gewone, direct overzichtelijke nummering, als codes gebruikt worden, dan A,B,C,D, of Zin x (als analogie met de codering van wetsteksten, Art. x)	3	3

Evaluatie: de taal van het Interface moet voor vertrouwdheid zorgen. De tekst waarvan in de cursus rechtsfilosofie gebruik wordt gemaakt, is in het Engels, dus het Interface moet ook in het Engels (twee verschillende talen tegelijk kan verwarrend zijn). Toch is een Interface in de moedertaal van de student (en dat is bij het merendeel van de doelgroep Nederlands) een voordeel. Deze keuze heeft alleen betrekking op een hogere laag van het systeem, en kan dus voorlopig voor ons uit worden geschoven. Er bestaan mogelijkheden om automatisch een stel web-pagina's te vertalen; voor het vertalen van de gehele Interface is zeker niet meer dan een week nodig.
Het gebruik van codes is handig (de doelgroep van rechten-studenten is gewend aan wetteksten waar in hoge mate ook codes worden gebruikt). Maar die codes mogen geen aanwijzingen geven over de inhoud van de zin: dat is erg verwarrend. Deze uitkomst, die bijna alle test-personen beaamden, is een beetje tegen-intuïtief, en dus erg leerzaam: hoe meer informatie er gegeven wordt (impliciet in de naam van de code) hoe meer het programma de alertheid van de student stimuleert.

Overzicht

* Voorkomen dat de student in losse clusters gaat denken door een totaaloverzicht te geven	2	2
* Zo simpel mogelijk en zo min mogelijk opties	2	2
* Een mogelijkheid bieden om de tekst van de kernzinnen uit te printen op een nette manier	3	3
* Het doel: redeneren tot de uitspraak van de rechter. Het moet duidelijker zijn wat je moet doen	2	3

Evaluatie: het is heel belangrijk dat de student een goed overzicht houdt. Het aantal opties moet beperkt worden, dat is: het aantal keuze-groepen waaruit op enig moment in het programma kan worden gekozen moet minimaal blijven. Het moet duidelijk zijn wat je moet doen. Een belangrijke stelregel is "reduce discrepancy between users' goals and system's physical state ([Preece1994], blz. 267). Ook moet de normale werkwijze van de student benaderd worden, door hem de mogelijkheid te geven met papieren te werken.

Bijlage 2: Lijst van begrippen

Argument Editor het op te leveren programma

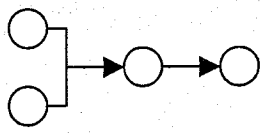
Argumentatie zie *redenering*

Boom een redenering die voor weergave is gestructureerd in een boomvorm. Het programma kent twee verschillende soorten bomen, een simpele versie (die voldoende is voor de meeste te analyseren teksten) en een versie met extra semantiek, die voor studenten met meer achtergrondkennis op het gebied van de logica nuttig kan zijn. De eerste soort boom kan logica's representeren die bestaan uit twee soorten proposities:

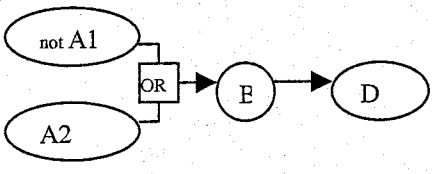
aannames (van de vorm A);

regels (van de vorm a_1 & ... & $a_n \Rightarrow$ B met $n \geq 1$).

De aannames zijn gerepresenteerd in de afzonderlijke knopen, en de regels in de groeperingen van knopen.
Een voorbeeld van de eerste soort boom is:



De tweede soort boom heeft extra semantiek bij de knopen en de pijlen. Een weergave van een redenering ziet er met deze boom-structuur bijvoorbeeld zo uit:



DFD Data Flow Modelling. Het statische model van de Argument Editor is een DFD.

ERM Entity-Relationship modelling.

Formule een expressie die uit *operatoren* en *variabelen* bestaat.

Kernzin een tekst die als bouwsteen gebruikt wordt bij de reconstructie. Wordt geïdentificeerd met een *variabele*.

Knoop zie *boom*

Onderbouwing een *knoop* die een andere onderbouwt. De onderbouwing wordt hier geïnterpreteerd als een logische implicatie. De knoop K met haar onderbouwing O vormt zo een *regel*

(K ⇒O) in een *boom*.

Operator één van te tekens '&', '|', '~'

Optie een keuze-mogelijkheid die de student heeft in de eerste fase, waar hij de kernzinnen moet onderscheiden van onnauwkeurige weergaves van de te analyseren tekst

Redenering een verzameling *secties* die bijvoorbeeld in een cursus gebruikt kunnen worden om een samenhangende groep argumentaties te presenteren.

Sectie een hiërarchische aaneenschakeling van argumenten, die door de student moet worden gereconstrueerd.

Regel zie *boom*

Structuur zie *boom*

Student een student die de Argument Editor gebruikt. In de regel is dat een rechtenstudent die aan de Universiteit van Tilburg het vak Rechtsfilosofie A volgt

Uitleg de tekst die een student krijgt getoond als hij een verkeerde *optie* kiest.

Variabele een niet verder te analyseren bouwsteen voor formules. Wordt gerepresenteerd door een string