Intelligente onderwijssystemen

Een verkennend onderzoek naar intelligente onderwijssystemen en implementatie daarvan in Prolog

Paul Bergervoet

RUU-CS-88-37
December 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Intelligente onderwijssystemen

Een verkennend onderzoek naar intelligente onderwijssystemen en implementatie daarvan in Prolog

Paul Bergervoet

Technical Report RUU-CS-88-37
December 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.089, 3508 TB Utrecht
The Netherlands

Inhoud

Hoofdstuk 1

Intelligente onderwijssystemen

achtergrond

Op het gebied van de toepassing van computers in het onderwijs is de laatste jaren een snelle ontwikkeling gaande: in hoog tempo worden computers geintroduceerd in scholen voor basisonderwijs en voortgezet onderwijs en allerlei instanties houden zich bezig met de ontwikkeling van educatieve software. Zo wordt er een zelfstandig vak 'informatica' ontwikkeld, dat een plaats moet vinden in het leerplan van deze scholen, maar ook wordt er materiaal ontwikkeld dat toegepast kan worden in bestaande schoolvakken. In de laatste categorie vinden we allerlei vormen, variërend van het gebruik van de computer als hulpmiddel (grafische pakketten, simulaties) tot geautomatiseerde lessen, waarbij een computer de leerling instrueert en vaak ook overhoort. Deze laatste toepassing wordt meestal aangeduid met CAI (*Computer Assisted Instruction*).

Bij de ontwikkeling van educatieve software blijken twee problemen steeds weer in meer of mindere mate op te treden:

— Het is moeilijk programma's op zinnige wijze te laten reageren op door de leerling gemaakte fouten.

— Het is moeilijk programma's zodanig te maken dat ze zich kunnen aanpassen aan door de leerling gekozen oplossingsstrategieën of handige verkortingen.

Het eerste type probleem treedt natuurlijk veelvuldig op in de informatica, maar is juist bijzonder belangrijk in de vakgebonden CAI-toepassingen. Vaak kunnen uit foute antwoorden conclusies getrokken worden: Welk deel van de stof beheerst de leerling wel en welk niet? Zijn er nauwkeurig vast te leggen lacunes in de kennis van de leerling?

Dit probleem speelt met name een rol bij het onderwijzen van algoritmen. Algoritmen komen in alle niveaus van het onderwijs voor. In het basisonderwijs vinden we ze bij het cijferen, bijvoorbeeld bij de staartdeling. In het voortgezet onderwijs vinden we ze in de wiskunde, het meest complexe algoritme is daar het functieonderzoek. Ook in andere vakken zijn er zaken die als algoritmen gezien kunnen worden, zoals het opstellen van reactievergelijkingen in de scheikunde.
Bij het onderwijzen van dergelijke algoritmen blijkt dat leerlingen vaak een groot deel van een algoritme beheersen en alleen op nauwkeurig te bepalen punten fouten maken. Wanneer deze fouten consequent gemaakt worden, worden ze wel aangeduid als *bugs*.[1]

De verbetering van deze bugs vergt meestal een specifieke aanpak, wat detectie ervan noodzakelijk maakt.

Het tweede type probleem vinden we vooral bij CAI-programma's die bedoeld zijn om stapsgewijs een algoritme te onderwijzen. De volgorde van stappen is meestal te rigide vastgelegd, waardoor het programma begint te zeuren als een leerling twee stappen in één keer doet of een andere volgorde van stappen kiest in een situatie waarin deze niet van belang is.

Het onderzoek naar zogenaamde *Intelligent Tutoring Systems*, ofwel intelligente onderwijssystemen, richt zich voor een belangrijk deel op het oplossen van de twee bovengenoemde problemen. Allereerst wordt er gewerkt aan systemen die kunnen bepalen welke fouten door leerlingen gemaakt worden. Dit systeem moet conclusies daarover kunnen aanreiken aan een "tutor". Bij het onderzoek naar deze tutors speelt het tweede probleem een belangrijke rol. Er wordt gewerkt aan tutors die de leerling het initiatief laten en de leerling niet dwingen zich aan het programma te conformeren.

doelstelling

Het onderzoek in dit rapport richt zich op het eerste type probleem, hoewel zal blijken dat het nauw verwant is met het tweede. Het onderzoek wordt in het bijzonder toegespitst op situaties waarin formulemanipulatie (in het bijzonder cijferen en algebra) en de bijbehorende algoritmes onderwezen worden.

Het herkennen van fouten in door leerlingen gehanteerde procedures kan worden opgevat als *diagnose*. De vergelijking met een expertsysteem als MYCIN [4] dringt zich daarom op: MYCIN probeert op basis van symptomen de oorzaak (ziekteverwekker) van een ziekte te vinden. Een programma voor het onderwijs dat bovengenoemde problemen aanpakt zal ook op basis van symptomen (antwoorden op opgaven) oorzaken (fouten in procedures) moeten kunnen vinden.
Daarnaast zijn er andere parallellen. Zo zal een diagnostisch systeem met onzekere informatie moeten kunnen werken; leerlingen maken immers ook inconsequente fouten, reken- en schrijffouten, die de diagnose vertroebelen. Zo dient zich onmiddellijk de vraag aan of intelligente onderwijssystemen ontworpen kunnen worden vanuit dezelfde principes als de tegenwoordige expertsystemen.

Verder is er een taal gezocht om een intelligent onderwijssysteem in te implementeren. Er is gekozen voor Prolog vanwege de veronderstelde geschiktheid van Prolog om rule based expert systems in te ontwikkelen en vooral ook vanwege het feit dat Prolog met relaties werkt en de mogelijkheid biedt om non-determinisme tot uitdrukking te brengen.

In het onderzoek zal het volgende aan de orde gesteld worden:

---

1. Zie paragraaf 2.3 voor een uitvoerige bespreking van bugs.

1. Welke karakteristieken zijn wenselijk voor empty shells van expertsystemen die toegesneden zijn op de problematiek van het diagnostiseren van fouten, die door leerlingen gemaakt worden in situaties waarin formules gemanipuleerd moeten worden. Daarbij dient vooral gedacht te worden aan toepassingen binnen de exacte vakken.
2. Het implementeren van een mogelijk geschikte empty shell voor de onder 1 genoemde doeleinden in Prolog.
3. Het bouwen van een onder punt 1 en 2 beschreven expertsyteem, toegepast op de diagnostisering van fouten bij algebravraagstukken.

Een onderzoek van dit soort mag ambitieus genoemd worden, omdat er al tenminste vier forse onderwerpen in te onderscheiden zijn:

1. Kenmerken van expert system shells in het algemeen, en in het bijzonder het redeneren met onvolledige en onzekere kennis.
2. Onderzoek naar specifieke eisen aan shells voor intelligente onderwijssystemen en onderzoek naar al ontwikkelde systemen.
3. De representatie van kennis uit het domeingebied, in dit geval algebraische manipulatie, en onderzoek naar door leerlingen gemaakte fouten en representatie daarvan in het expertsysteem.
4. Implementatie van dit alles in Prolog.

Daarbij komt nog dat het terrein in hoge mate onontgonnen is en de problematieken nog verre van doorgrond zijn. Het dient dan ook niet te verbazen dat de doelstellingen nog maar ten dele gerealiseerd zijn. Dit rapport dient daarom eerder gezien te worden als een verslag van een vooronderzoek, waarin het terrein verkend wordt en richtlijnen worden aangegeven, waarlangs de ontwikkeling van het genoemde expertsysteem zou kunnen verlopen.

verkenning van bestaande systemen

In de cognitieve psychologie is al vrij veel onderzoek gedaan naar intelligente onderwijssystemen. Een aantal systemen is al ontwikkeld of in ontwikkeling. Twee daarvan zijn BUGGY, een zeer ver ontwikkeld systeem voor het diagnostiseren van fouten in aftreksommen en het Leeds Modelling System (LMS), dat betrekking heeft op het oplossen van lineaire vergelijkingen. Beide systemen bleken een rijke bron aan ideeën te bevatten.

BUGGY probeert fouten te diagnostiseren in het bekende aftrekalgoritme waarbij de getallen onder elkaar geschreven worden. Hiertoe werd het algoritme genoteerd in een hiërarchie van procedures en subprocedures. Het algoritme wordt hieronder gegeven, in Prolog-clauses. Merk op dat getallen worden weergegeven als lijsten van cijfers, met het minst significante cijfer vooraan, 218 wordt zo dus [8, 1, 2].
BUGGY werd ontwikkeld om te onderzoeken of een leerling een bepaald soort fout consequent maakt. Consequente fouten wijzen er immers op dat de leerling een bepaald

```
        minus(Type, Bs, Os, Us):
De aftrekking Bs-Os resulteert in Us volgens algoritme Type.
Getallen worden gerepresenteerd door lijsten en length(Bs)<=length(Us).

        leen(Type, Bs, NBs):
Lenen van Bs volgens algoritme Type resulteert in NBs.

        naam_type(Type, Naam):
Naam is een omschrijving van het algoritme met nummer Type.

naam_type(0, 'correct algoritme.').

minus(0, [B | Bs], [O | Os], [U | Us]):-
            kolom_kan(B, O),
            tafel(B-O, U),
            minus(0, Bs, Os, Us).

minus(0, [B | Bs], [O | Os], [U | Us]):-
            not kolom_kan(B, O),
            leen(0, Bs, NBs),
            tafel(10+B-O, U),
            minus(0, NBs, Os, Us).

minus(0, Bs, [], Bs).

leen(0, [B | Bs], [NB | Bs]):-
            not is_nul(B),
            tafel(B-1, NB).

leen(0, [0 | Bs], [9 | NBs]):-
            leen(0, Bs, NBs).
```

Figuur 1.1: Het aftrek-algoritme in Prolog.

deel van het algoritme niet beheerst, en daarom wellicht op dat onderdeel hulp nodig heeft. Uit willekeurige fouten kunnen zulke conclusies niet getrokken worden.
Fouten worden gerepresenteerd door *buggy* varianten van de subprocedures, die *mal-rules* of *bugs* genoemd worden. Diagnose wordt gesteld door te zoeken naar een combinatie van mal-rule(s) en correcte regels waarmee het gedrag van de leerling gesimuleerd kan worden: de incorrecte procedure moet hetzelfde (foute) antwoord geven als de leerling. Daarbij wordt gezocht naar mal-rules die door een leerling consequent worden toegepast.
Voor het aftrek-algoritme bleek diagnose al snel zeer ingewikkeld te worden. Na onderzoek van gegevens van enkele duizenden leerlingen werden meer dan 100 mal-rules gevonden. Het aantal foute algoritmen dat door leerlingen gehanteerd werd, was nog veel groter, omdat veel leerlingen met meer dan één mal-rule werkten.

In feite wordt hier volgens een *generate-and-test* model gewerkt. Eerst worden mogelijke bugs *gegenereerd*, waaronder ook samengestelde bugs, die uit verscheidene mal-rules bestaan. Hier zijn verschillende technieken nodig om de zoekruimte te beperken; het aantal combinaties van 4 mal-rules is bij het aftrek-algoritme immers al in de orde van 100^4. Er wordt onder andere gebruik gemaakt van heuristieken, die aangeven of bepaalde combinaties van mal-rules mogelijk zijn.

Het *testen* van een bug is gecompliceerd omdat er allerlei onzekerheden in de vergaarde gegevens zitten. Als een leerling een bepaald soort fout consequent maakt, kan het toch zijn dat de mal-rule de antwoorden niet altijd correct voorspelt; leerlingen maken immers ook willekeurige fouten en reken- en schrijffouten. Ook kan het zijn dat de leerling helemaal geen mal-rule hanteert, maar gewoon slordig werkt. Dit leidt tot een gecompliceerde vorm van afwegen van verschillende hypothesen.

LMS gebruikt productiesystemen om het oplossen van een lineaire vergelijking in één onbekende te beschrijven. Ook hier wordt geprobeerd een leerling die fouten maakt te simuleren door correcte regels in het productiesysteem te vervangen door mal-rules.

een model voor een diagnostisch systeem

Het principe van de mal-rule spreekt intuitief aan, juist omdat bugs als varianten van correcte procedures gerepresenteerd worden. Het principe lijkt veel breder toepasbaar in situaties waarin oorzaken worden gezocht voor slecht functioneren. Daarvoor is een representatie van correct functioneren nodig, bijvoorbeeld een gezond iemand of een correct werkend electronisch circuit, plus een representatie van de effecten van verstoringen. Diagnose vindt dan plaats door te testen of de geobserveerde verschijnselen overeenstemmen met de door het verstoorde model gegenereerde verschijnselen.

Mal-rules hebben ook nadelen. Niet alle relevante informatie kan in mal-rules gerepresenteerd worden. Dit geldt bijvoorbeeld voor *circumstancial evidence*: regels die stellen dat in bepaalde situaties de ene bug waarschijnlijker is dan de andere. Zulke regels beschrijven immers niet de effecten van verstoringen. Verder is het gebruik van mal-rules lastig in situaties van onvolledige informatie, waarin niet voldoende symptomen verzameld kunnen worden om onderscheid te kunnen maken tussen verschillende bugs.
Dit laatste is precies het probleem dat in een expertsysteem als MYCIN aangepakt wordt. Volledige informatie kan niet verkregen worden, of liever gezegd: sommige ziekteverwekkers hebben minder tijd nodig om een patiënt te doen overlijden dan het laboratorium nodig heeft om de identiteit van de betreffende ziekteverwekker vast te stellen. Daarom moet gewerkt worden met onvolledige informatie en 'circumstancial evidence', waaruit een hoge mate van onzekerheid voortvloeit. Een verzachtende omstandigheid is dat het vinden van één diagnose niet nodig is. Wanneer er een aantal onzekere, maar wel waarschijnlijke kandidaten gevonden is, kan een antibioticum toegepast worden dat *alle* kandidaten bestrijdt.

Bij intelligente onderwijssystemen hebben we te maken met een geheel andere situatie. Ten eerste is er in principe geen beperking aan het aantal symptomen, hier antwoorden op opgaven. Wanneer het systeem niet over voldoende informatie beschikt om te kunnen kiezen tussen twee hypothesen, kan men de leerling extra opgaven voorleggen. Daartegenover staat dat een eenduidige diagnose is (wanneer die bestaat) zeer gewenst is, omdat de op de diagnose volgende uitleg meestal op de bug moet zijn toegesneden. Als gevolg van deze overweging is de rol van 'circumstancial evidence' zeer beperkt: Dit wordt immers gebruikt wanneer er een gebrek aan symptomen is en geeft onzekere conclusies.

Ten tweede hebben we hier te maken met een geheel andere vorm van onzekerheid. Deze is niet aanwezig in de mal-rules, die alleen effecten van bugs beschrijven. maar in de symptomen. Foute antwoorden komen niet alleen voort uit bugs, er kan ook sprake zijn van lees-, schrijf- en rekenfouten. Met andere woorden, een systeem als BUGGY kan precies aangeven tot welke antwoorden een bepaalde bug leidt, maar deze antwoorden zullen soms afwijken van de antwoorden die een leerling die dezelfde bug hanteert geeft, omdat deze ook allerlei willekeurige fouten kan maken.
In BUGGY worden speciale technieken gebruikt om deze onzekerheden te representeren. Zo wordt onder andere gekeken of het verschil tussen het antwoord van een leerling en het door een bug voorspelde antwoord verklaard kan worden uit rekenfouten. Door deze speciale vorm van onzekerheid lijken certainty factors niet toepasbaar.
De rol van 'circumstancial evidence' is zeer beperkt. Dergelijke regels kunnen gebruikt worden in de bug-generator, zodat deze veel voorkomende bugs produceert vóór weinig voorkomende.

Samenvattend kan de volgende precisering van het generate-and-test model voor diagnostische systemen gegeven worden:

1. De *bug-tester* genereert volgens een buggy procedure antwoorden op opgaven en vergelijkt deze met de door de leerlingen gegeven antwoorden. De bug-tester beschikt over technieken om afwijkingen in het antwoord van de leerling te verklaren.

2. De *bug-generator* beschikt over correcte regels en mal-rules en kan daaruit buggy procedures construeren. Om de zoekruimte te beperken kan de bug-generator gebruik maken van heuristische regels die betrekking hebben op de mal-rules (frequentie, samen voorkomen van mal-rules in één buggy procedure) en van door de bug-tester geleverde resultaten.

overzicht van het onderzoek

In hoofdstuk 2 worden bestaande intelligente onderwijssystemen besproken. Algemene kenmerken van intelligente onderwijssystemen worden gegeven en BUGGY en LMS worden uitgebreid beschreven. Hoofdstuk 3 bevat documentatie bij een aantal werkende Prolog-programma's en is een aanzet voor de ontwikkeling van een diagnostisch systeem dat werkt op basis van mal-rules. Er wordt uitgebreid ingegaan op de representatie van rules en mal-rules in Prolog. Dit gebeurt steeds aan de hand van het aftrek-algoritme, dat in dit hoofdstuk wordt gebruikt als paradigma; het staat voor een klasse van onderwerpen in het onderwijs waarin algoritmen geleerd worden. Verder wordt een bug-tester gegeven. Aan de ontwikkeling van een bug-generator is in dit onderzoek nog weinig aandacht besteed.
In de slotparagrafen van de hoofdstukken 2 en 3 worden lijnen gegeven, waarlangs het benodigde verdere onderzoek zou kunnen lopen.

conclusie

De voorlopige conclusie uit dit vooronderzoek is dat het mogelijk lijkt om in Prolog diagnostische systemen te ontwikkelen die volgens het mal-rule principe werken. Voor

elk domein dienen dan de volgende onderdelen gegeven te worden:

1. Een Prolog-programma dat de manipulaties uit het domein uitvoert. Dit is de verzameling correcte regels.
2. Een verzameling mal-rules, die de mogelijke fouten beschrijven.
3. Een verzameling heuristische regels, die gehanteerd moeten kunnen worden door de bug-generator.

Er kunnen empty shells gemaakt worden die bovenstaande verzamelingen regels gebruiken en diagnose stellen door middel van een 'generate and test' procedure. Het is onduidelijk of de voornoemde vereisten ook in alle domeinen binnen het onderwijs vervuld kunnen worden.

Hoofdstuk 2

Bestaande systemen

In dit hoofdstuk wordt een overzicht gegeven van de literatuur over *Intelligent Tutoring Systems*, kortweg ITS. Het uitgangspunt hiervoor is D. Sleeman en J. S. Brown [12], waarin een groot aantal intelligent tutoring systems beschreven wordt. Zie aldaar voor details en achtergronden.

Er zal eerst een algemene inleiding geven worden, waarin gekeken wordt naar toepassingen van de computer in het onderwijs in het algemeen en meer specifiek naar kenmerken van ITS.
Vervolgens zullen twee specifieke systemen uitgebreid besproken worden. Eén daarvan heeft betrekking op het algoritme voor aftrekken, de tweede op een eenvoudig onderdeel van de algebra. Beide systemen hebben geleid tot de formulering van belangrijke modellen voor ITS en tot werkende implementaties. Dit heeft niet alleen allerlei praktische problemen aan het licht gebracht, maar ook geleid tot uitbreidingen van de systemen.
Daarna volgt een kort overzicht van een aantal andere intelligent tutoring systems.
In de slotparagraaf worden conclusies gegeven met betrekking tot de ontwikkeling van intelligente onderwijssystemen in Prolog voor formulemanipulatie. Deze conclusies zijn verwerkt in hoofdstuk 3.

2.1 De computer in het onderwijs

Het gebruik van de computer in het onderwijs kan grofweg in drie categorieën ingedeeld worden.

Ten eerste is er het leren *over* de computer. Weliswaar is er nog geen sprake van een vak 'informatica' in het onderwijs, maar cursussen 'informatieleer en computerkunde', ook wel 'computer literacy' genoemd, zijn vrij algemeen. In dergelijke cursussen wordt voornamelijk aandacht besteed aan toepassingen van de computer, zoals tekstverwerking en databanken. Ook wordt wel gewerkt met speciaal voor het onderwijs ontwikkelde programmeertalen, waaronder LOGO, ELAN en Ecol.

Ten tweede is er het leren *met behulp van* de computer. In deze toepassingen wordt de computer gebruikt als hulpmiddel bij het uitvoeren van allerlei aktiviteiten die in het klaslokaal onmogelijk zijn, of teveel tijd vergen. Voorbeelden hiervan zijn simulaties van (onuitvoerbare) experimenten en, in het wiskundeonderwijs, programma's die grafieken tekenen of matrixberekeningen uitvoeren. Dergelijke programmatuur moet gezien worden als 'gereedschap', dat wanneer nodig door een leerling gebruikt kan

worden. Het geeft geen leerweg aan en doet ook niet aan beoordeling van leerlingen.

Ten derde is er het leren *door middel van* de computer, meestal aangeduid met *Computer Assisted Instruction* (CAI). In tegenstelling tot het leren met behulp van de computer, proberen CAI-programma's expliciet kennis en vaardigheden over te brengen op een leerling. Daarbij worden door het programma ook vragen aan de leerling gesteld en de antwoorden worden beoordeeld.
Ook CAI is weer onder te verdelen in verschillende types. Een aantal belangrijke types zijn:

1. *Drill and practice.*
   In een drill and practice programma krijgt een leerling een aantal opgaven voorgeschoteld. De antwoorden worden vergeleken met de in het programma ingebakken goede antwoorden, bij een fout antwoord krijgt de leerling eventueel een herkansing. Het aantal goede antwoorden wordt geturfd. Er bestaan veel varianten van dit soort programmatuur, waarbij de drill and practice bijvoorbeeld in een spel gegoten is, waarbij je punten krijgt voor elk goed antwoord. In andere varianten kan de moeilijkheidsgraad van de opgaven aangepast worden aan de leerling.
   In het algemeen gaat het bij drill and practice om grote aantallen eenvoudige opgaven, zoals simpele sommen of invuloefeningen. Het nut van dergelijke programma's is vooral het interaktieve karakter: de leerling krijgt direct antwoord.
2. *Auteurstalen.*
   In een andere vorm van CAI vinden we de zogenaamde auteurstalen. Een auteurstaal is een programma-pakket waarmee een les kan worden samengesteld. Bekende auteurstalen zijn Plato, Pilot en Taiga. Bij CAI-pakketten die met dergelijke auteurstalen ontwikkeld zijn gaat het vooral om het gestructureerd aanbieden van informatie. De informatie wordt ingedeeld in pagina's (die op het beeldscherm passen) en er wordt een route door deze pagina's vastgelegd. In deze route kunnen toetsvragen worden opgenomen. De antwoorden op deze vragen bepalen de route door de stof. Bij foute antwoorden kan er bijvoorbeeld extra, meer gedetailleerde informatie worden aangeboden, of overgegaan worden op herhaling.
3. *Intelligente CAI.*
   Intelligente CAI baseert zich op 'learning by doing'. Hierbij werkt de leerling aan een probleem, waarbij de computer optreedt als *coach* of *tutor*. Bij de ontwikkeling van dergelijke CAI-pakketten komen twee punten als duidelijke probleemgebieden naar voren:
   a. Aanpassen van de tutoring aan verschillende oplossingsstrategieën voor een probleem, die door leerlingen gehanteerd kunnen worden.
   b. Aanpassen van de tutoring aan kennis en vaardigheden – of het gebrek aan kennis en vaardigheden – van de leerling.

   Het zijn juist deze punten waarop het onderzoek naar intelligent tutoring systems zich concentreert. Beide punten stellen hoge eisen aan een ITS. Het eerste punt eist dat het systeem 'kennis' heeft van het domein waarop de tutoring plaatsvindt, het tweede punt eist dat het systeem kennis heeft over mogelijke lacunes en

fouten in de vaardigheden van een leerling.
Deze soorten kennis zijn nauw verweven. Kennis over de vaardigheden van een individuele leerling wordt vaak gerepresenteerd als variant of deel van de complete vaardigheden op het domein.

Uit het voorgaande blijkt ook dat de ontwikkeling van intelligent tutoring systems nauw verbonden is met de theorie over het leren. Daarom vindt het grootste deel van het onderzoek naar deze systemen plaats in de cognitieve psychologie en dient ook een tweeledig doel: enerzijds probeert men te komen tot een theoretisch model dat het leergedrag van leerlingen beschrijft, anderszijds tot een werkend ICAI-systeem. Veel systemen zijn ook gebaseerd op zo'n theoretisch model. Deze tweeledigheid leidt echter ook tot onduidelijkheid: Vaak ontbreekt informatie over de bij de implementatie gebruikte technieken, waardoor het soms zelfs niet duidelijk is of uit de beschreven modellen ook werkende systemen zijn voortgekomen.

De representatie van de voor intelligent tutoring systems benodigde kennis staat in de volgende paragraaf centraal. Theoretische modellen over het leren uit de cognitieve psychologie zijn niet het onderwerp van deze studie, maar wel van belang bij het ontwerpen van intelligente onderwijssystemen. We beperken ons tot eenvoudige beschrijvingen van modellen, zonder wetenschappelijke verantwoording.

2.2 Kenmerken van intelligent tutoring systems

Een intelligent tutoring system bestaat uit vier onderdelen (Dede [11], Sleeman [18]):

1. Een *knowledge base* van kennis over het te onderwijzen domein, inclusief een mechanisme om deze kennis te manipuleren.
2. Een mechanisme voor het genereren van een *student model*, dat de verschillen tussen de kennis en vaardigheden van de leerling en de 'correcte' kennis en vaardigheden weergeeft.
3. Een *pedagogical module*, dat de tutoring uitvoert op grond van informatie in de eerste twee onderdelen. Dit onderdeel bevat kennis over didaktiek.
4. Een *user interface* naar de leerling toe.

In dit onderzoek gaat het om het stellen van diagnoses, en niet om het toepassen van therapieën. Daarom wordt alleen op de eerste twee delen dieper ingegaan. Dede [11] geeft een uitvoerige beschrijving van de eisen die aan de verschillende onderdelen gesteld kunnen worden. Deze eisen komen veelal voort uit uit onderzoek in de cognitieve psychologie, waar de nadruk niet ligt op het ontwikkelen van geautomatiseerde systemen. Waar mogelijk noemt Dede echter wel systemen, waarin geprobeerd is aan een of meer van deze eisen tegemoet te komen.

2.2.1 De knowledge base

Wanneer *learning by doing* het uitgangspunt is bij de ontwikkeling van een intelligent tutoring system voor een bepaald domein, zal dat systeem tenminste die kennis moeten bevatten die ook in een expertsysteem voor dat domein zit. In learning by doing werkt

de leerling immers aan (gesimuleerde) problemen uit het domein. De tutor bewaakt dit proces en moet dus beschikken over correcte methoden om die problemen op te lossen. Dit impliceert dat de ontwerper van een ITS met dezelfde zaken geconfronteerd zal worden als de 'knowledge engineer' van een expertsysteem. Daarnaast moeten aan de kennis in een ITS extra eisen gesteld worden: de kennis zal dusdanig gestructureerd en geformuleerd moeten worden, dat deze zich leent voor overdracht op leerlingen. Dede gebruikt daarbij het beeld dat de kennis in een 'gewoon' expertsysteem voor de gebruiker veelal functioneert als een *black box*, kennis in een ITS moet voor de gebruiker te doorzien zijn, ze moet functioneren als een *glass box*.
Dit wordt met name duidelijk bij de pogingen om bovenop MYCIN een intelligente tutor voor medicijnenstudenten te bouwen. Dit experiment, GUIDON geheten, wordt door Clancey beschreven (Clancey [20], ook in Buchanan [4]). Clancey constateerde dat de kennis in MYCIN niet zonder meer geschikt was voor gebruik in een ITS. Er waren zowel problemen met individuele regels, als met het gehele systeem.
In rule-based expert system wordt gewerkt met een rule-base die bestaat uit een groot aantal losstaande regels. De kennis wordt daardoor fragmentarisch: in de regels ontbreekt bijvoorbeeld informatie over de causale verbanden die achter die regels schuilgaan en het grotere kader waarin deze regels passen. Ook op globaal niveau ontbreken zaken die voor studenten van belang zijn: tijdens een consultatie stelt MYCIN vragen aan de gebruiker. Weliswaar kan MYCIN aangeven *waarom* een vraag gesteld wordt, maar dit gebeurt 'lokaal': MYCIN geeft aan in welke regel het gevraagde relevant is. Het is echter niet a priori duidelijk wat voor een *strategie* voor het stellen van de diagnose door MYCIN gehanteerd wordt.
In GUIDON leidden deze problemen tot de invoering van *meta-regels* voor de representatie van strategieën en meer algemene aanwijzingen, en tot de invoering van *causale ketens* voor de representatie van causale verbanden.

Dede geeft het volgende overzicht van verschillende vormen van kennis en kennisrepresentatie waar recentelijk onderzoek naar gedaan is. Van alle genoemde zaken kan gezegd worden, dat het van belang is dat dergelijke kennis in de knowledge base van een ITS wordt opgenomen. De mate van belang hangt echter sterk af van het domein.

Soorten kennis
De knowledge base van een ITS zal drie soorten kennis moeten bevatten:

1. *Declaratieve kennis.*
   Kennis van dit soort is beschrijvend en beantwoordt vooral de vragen die beginnen met 'wat'.
2. *Procedurele kennis.*
   Dit soort kennis geeft antwoord op vragen die beginnen met 'hoe'. Hierin vallen bijvoorbeeld algoritmen.
3. *Metacognitieve kennis.*
   Dit soort kennis wordt door Dede omschreven met 'thinking about what en how'. Hier vinden we de *problem solving strategieën.*

Verschillende invalshoeken
In een onderwijssituatie kan het van groot belang zijn om kennis op verschillende

manieren te benaderen. Het leren van (complexe) structuren en procedures wordt bevorderd door deze verschillende benaderingen, of (Dede) niveaus van betekenis. Dede noemt vier van deze niveaus.

1. *Beeldvorming.*
   In het leren speelt beeldvorming een belangrijke rol. Vaak is kennis over hoe iets precies werkt niet nodig, of te gedetailleerd om in één keer geleerd te worden. Globale beschrijvingen, metaforen en kwalitatieve redeneringen geven een leerling vaak voldoende houvast om een probleem aan te pakken. Een (dubieus) voorbeeld is de zin "*Je kunt toch geen appels en bananen optellen!*" die gebruikt wordt om te verklaren dat een formule als *3a+4b* niet vereenvoudigd kan worden.

2. *Modulariteit.*
   Kennis, en met name procedurele kennis, dient gestructureerd te zijn om goed overgedragen te kunnen worden. Zo kan een algoritme, op een hoger niveau dan de kale handelingen, omschreven worden in termen van doelen en subdoelen.

3. *Beperkingen en sterke punten van gebruikers.*
   Er bestaan aanzienlijke verschillen tussen procedures die voor mensen makkelijk te leren zijn en procedures die zich goed laten automatiseren. Dit heeft tot gevolg dat de kennis in een expertsysteem niet automatisch geschikt is voor tutoring. Een voorbeeld vinden we in verschillende algoritmes voor het rekenen. Sommige zijn niet geschikt voor het onderwijs, omdat er teveel 'even onthouden' moet worden. In expertsystemen zijn soms noodgrepen nodig om het systeem goed te laten werken Het is niet wenselijk dergelijke noodgrepen te onderwijzen.

4. *Context.*
   Nieuwe kennis staat zelden op zichzelf. Bij het leren van nieuwe procedures kunnen analogieën verhelderend werken. Ook kan een kader van algemeen bruikbare procedures van groot belang zijn. Een voorbeeld hiervan is het controleren-door-invullen bij het oplossen van vergelijkingen.

Volgorde in kennis
Kennis en vaardigheden vormen geen grijze massa waaruit een willekeurig brokje gekozen kan worden, dat onderwezen gaat worden. Het leren gebeurt via een bepaalde volgorde. Een manier om de volgorde in het leren vast te leggen is de *genetische graaf*. Hierin wordt weergegeven welke kennis noodzakelijk is vóór de verwerving van een bepaald stuk nieuwe kennis.

2.2.2 Student modelling

Een ITS moet zich kunnen aanpassen aan een individuele leerling. Ten behoeve van de tutoring dient het systeem zich een beeld te vormen van de kennis en vaardigheden van de op dat moment met het systeem werkende leerling. Dit proces heet *student modelling*. Een ideaal systeem moet kunnen vaststellen welke kennis een leerling bezit en welke ontbreekt, welke fouten door een leerling gemaakt worden en wat de oorzaak van die fouten is. Daarnaast kunnen met het leren verwante zaken voor een

leerlingmodel van belang zijn, zoals welke meta-vaardigheden een leerling heeft – bijvoorbeeld kunnen leren van gemaakte fouten – en wat de leerstijl en interesse van een leerling zijn.

Informatie kan worden afgeleid op vier verschillende manieren:

1. *Impliciet*, op basis van het 'gedrag' van de leerling en met name zijn antwoorden op vragen.
2. *Expliciet*, op grond van de dialoog tussen leerling en systeem.
3. *Structureel*, op basis van bekende feiten over de moeilijkheidsgraad van de stof.
4. *Achtergrond*, bijvoorbeeld op grond van informatie over de frequentie van een bepaald type fouten.

Op grond van deze observaties beweert Dede dat student modelling nog ingewikkelder is dan de representatie van expertise, die in de vorige paragraaf beschreven is. In het volgende zal ik me beperken tot modellen die kennis en fouten van leerlingen beschrijven.

In situaties waarin het om het leren van procedures gaat, maakt men leerlingmodellen die het 'gedrag' van een leerling simuleren. Dit wil zeggen dat het leerlingmodel in staat moet zijn te voorspellen wat voor antwoord een leerling op een bepaalde vraag zal gaan geven. Uit verschillende onderzoeken zijn een aantal benaderingen voor het opstellen van een leerlingmodel naar voren gekomen. In al deze benaderingen komt men tot leerlingmodellen door verschillen met de in het systeem aanwezige kennis van het domein te bekijken.

1. *Overlay models.*
   In de benadering van de overlay-models wordt de kennis van de leerling gezien als een deel van de 'volledige' kennis, die in het systeem gerepresenteerd is. In dit kader worden de eerdergenoemde genetische grafen gehanteerd.

2. *Differential models.*
   In de benadering van de differential models wordt het gedrag van de gemechaniseerde expert bij het oplossen van een probleem vergeleken met dat van de leerlingen. Op grond van de verschillen worden conclusies getrokken over de kennis en vaardigheden van de leerling.

3. *Perturbation models.*
   Het perturbation model gaat er van uit dat het gedrag van de leerling gesimuleerd kan worden door een 'verstoorde' versie van de kennis van de expert. Zo kan een leerlingmodel gegenereerd worden door uit de kennis van de expert bepaalde subprocedures weg te laten of te vervangen door 'buggy' subprocedures. Als de domein-kennis gerepresenteerd wordt door middel van (productie)regels, kan de leerling gesimuleerd worden door middel van zogenaamde *mal-rules*. Deze benadering wordt gehanteerd in de systemen, die in de volgende paragraaf aan de orde komen en zal daar uitvoerig besproken worden.

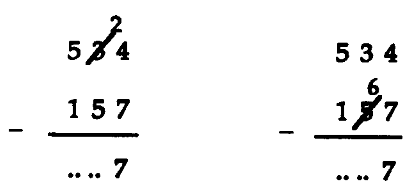
In zijn artikel bespreekt Dede verder nog de pedagogical module en de user interface, die hier buiten beschouwing gelaten worden.

Dede's voor de hand liggende conclusie is dat het ontwikkelen van intelligente tutors nog zeer veel onderzoek zal vergen dat nauw verwant is met 'some of the most complex research issues in AI'. Het onderzoek wordt verder bemoeilijkt door het feit dat het bijna onmogelijk is halfproducten in de praktijk toe te passen; alle genoemde onderdelen zijn dusdanig belangrijk, dat ze niet kunnen worden weggelaten uit een systeem dat door leerlingen gebruikt wordt. Verder plaatst hij vraagtekens bij het ontwikkelen van empty shells voor tutoring systemen. Het onderwijs in verschillende onderwerpen stelt namelijk zeer verschillende eisen op alle genoemde onderdelen.

2.3 Een case study: het aftrek-algoritme

De eerste case study heeft betrekking op het algoritme voor het maken van aftrekkingen. Er is betrekkelijk veel literatuur op het gebied van de intelligent tutoring systems die gaat over algoritmen voor aftrekken. Hiervoor zijn verschillende redenen aan te voeren. Ten eerste is het een relatief eenvoudige vaardigheid, waardoor het automatiseren van het algoritme en de bijbehorende leerlingmodellen een haalbare operatie lijkt. Ten tweede is het complex genoeg, wat het interessant maakt voor een case study over de moeilijkheden bij het stellen van diagnose.

Het uitgangspunt is het bekende algoritme waarbij de beide getallen onder elkaar worden opgeschreven en de aftrekking kolomsgewijs plaatsvindt. In dit algoritme is soms *inwisselen (lenen)* noodzakelijk. Hiervoor bestaan verschillende methoden. In de eerste methode wordt er ingewisseld in de *minuend* (het getal boven) door het getal links van de kolom waaraan gewerkt wordt met één te verminderen. In de tweede methode wordt er één opgeteld bij het getal in de *substrahend* (het getal onder), zie figuur 3.1. De eerste methode wordt het meest gebruikt, maar er zijn ook leerboeken die van de tweede, of weer andere methodes uitgaan.



figuur 3.1: verschillende methodes van inwisselen.

De fouten bij leerlingen lopen uiteen van zeer elementaire fouten, die bij bijna elke opgave optreden, tot fouten die alleen in bijzonder lastige gevallen optreden. Zo zijn er leerlingen die de getallen links uitgelijnd opschrijven en van links naar rechts te werk gaan. Bij de lastige gevallen gaat het vooral om fouten bij het inwisselen, zeker als er nullen in de minuend staan. In figuur 3.2 wordt een aantal voorbeelden van regelmatig voorkomende fouten gegeven.

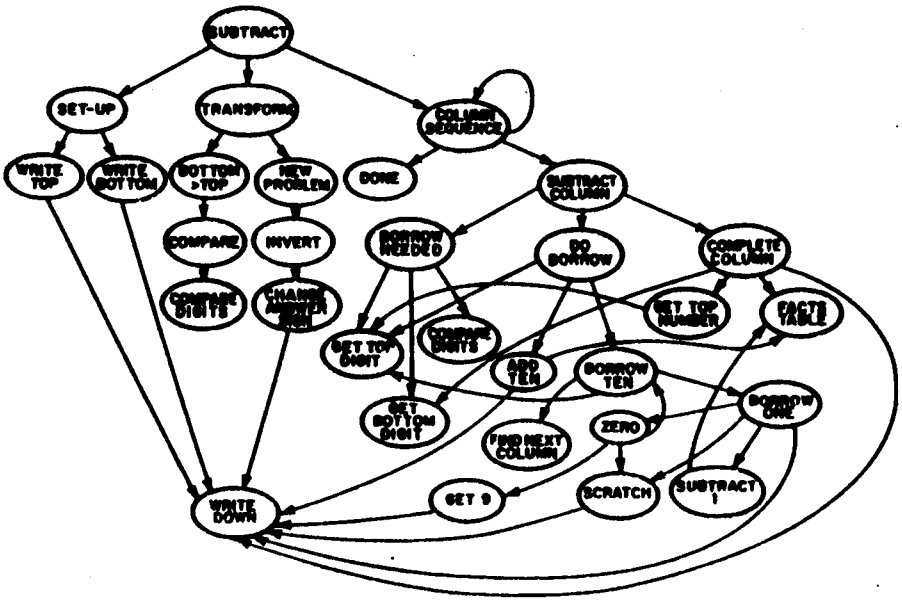
143 −28 125	The student subtracts the smaller digit in each column from the larger digit regardless of which is on top.
143 −28 125	When the student needs to borrow, he adds 10 to the top digit of the current column without subtracting 1 from the next column to the left.
1300 −522 878	When borrowing from a column whose top digit is 0, the student writes 9 but does not continue borrowing from the column to the left of the 0.
140 −21 121	Whenever the top digit in a column is 0, the student writes the bottom digit in the answer; i.e., $0-N=N$.
140 −21 120	Whenever the top digit in a column is 0, the student writes 0 in the answer; i.e. $0-N=0$.
1300 −522 788	When borrowing from a column where the top digit is 0, the student borrows from the next column to the left correctly but writes 10 instead of 9 in this column.
321 −89 231	When borrowing into a column whose top digit is 1, the student gets 10 instead of 11.
662 −357 205	Once the student needs to borrow from a column, s/he continues to borrow from every column whether s/he needs to or not.
662 −357 115	The student always subtracts all borrows from the leftmost digit in the top number.

figuur 3.2: Een aantal fouten in het aftrek-algoritme. Uit: [15]

2.3.1 Procedurele netwerken

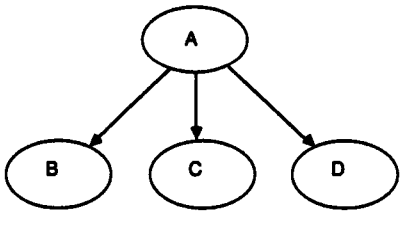
Brown e.a [13, 15, 17] hebben gedurende een aantal jaren uitgebreid onderzoek gedaan naar fouten bij het aftrek-algoritme. Dit heeft geleid tot de ontwikkeling van een diagnostisch systeem, BUGGY, en later tot diverse varianten van dit systeem, DEBUGGY en IDEBUGGY.
Het uitgangspunt van BUGGY is het concept van *procedurele netwerken.* Een procedureel netwerk omvat informatie over de manier waarop een complexe procedure opgedeeld kan worden in allerlei subprocedures.[1] Deze opdeling is bijzonder relevant bij het maken van leerlingmodellen, omdat leerlingen veelal een groot deel van de procedure beheersen en alleen consequent fouten maken in bepaalde subprocedures. Figuur 3.3 bevat het door Brown en Burton [15] gegeven netwerk voor het aftrek-algoritme.

1. In Prolog-terminologie zou men ook kunnen spreken van doelen en subdoelen.

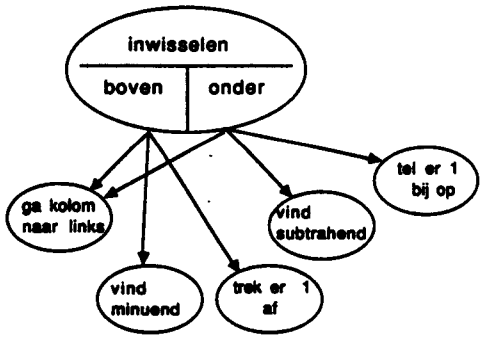


figuur 3.3: procedureel netwerk voor het aftrekken. Uit: [15]

Het procedurele netwerk bevat informatie over de samenhang van de subprocedures in het algoritme. Een verbinding van knoop A naar knoop B betekent dat een aanroep van subprocedure B (mogelijk) nodig is om procedure A uit te voeren. De volgorde van aanroep van subprocedures wordt gegeven door de volgorde van de verbindingen; subprocedures moeten van links naar rechts uitgevoerd worden. In figuur 3.4 moeten de subprocedures B, C en D in de genoemde volgorde worden uitgevoerd. Brown en Burton geven ook een manier om alternatieven weer te geven, zie figuur 3.5, waarin de keuze van de methode voor het inwisselen is weergegeven. Hoewel de notatie afwijkt van de gangbare, kunnen we het netwerk lezen als een AND/OR-tree.



figuur 3.4: volgorde van aanroep.



figuur 3.5: 'OR-node' met gedeelde subprocedures.

Wanneer we het netwerk vanuit algoritmisch perspectief beschouwen, is het ietwat merkwaardig. Een deel van van de control-informatie (procedure-calls en volgorde van aanroep) is aanwezig, maar een belangrijk deel (conditionele aanroep, iteratie) ontbreekt. Dit is duidelijk te zien bij de knopen voor *column sequence* en *subtract column*.

Bij de eerste knoop ontbreekt informatie over *wanneer* de procedure herhaald moet worden. Bij de tweede ontbreekt het verband tussen de uitslag van *borrow needed* en de aanroep van *do-borrow*. Daarom moet een procedureel netwerk niet opgevat worden als een algoritme, maar als een beschrijving van welke subprocedures vereist zijn om een procedure uit te kunnen voeren. Een compleet algoritme in Prolog, dat volgens dezelfde opdeling werkt, zal gegeven worden in hoofdstuk 4.

Verder is de mate van detail van het netwerk van groot belang. In principe kan men natuurlijk zeer ver gaan in het opdelen van procedures in subprocedures. Uiteindelijk moet men echter uitkomen op een aantal ondeelbare subprocedures. In dit geval zijn dat bijvoorbeeld operaties als het herkennen, opschrijven en vergelijken van cijfers en de *facts table*, de tafels, voor eenvoudige aftrekkingen en optellingen.

Het procedurele netwerk vormt de basis voor het genereren van leerlingmodellen. Dit verloopt volgens het *perturbatie-model*: het idee is dat de verrichtingen van leerlingen gesimuleerd kunnen worden door bepaalde subprocedures, of een groep subprocedures, te vervangen door foutieve varianten, de zogenaamde *buggy procedures*. Aan deze buggy procedures werden twee restricties opgelegd.

1. Het komt veel voor dat een leerling een deel van het aftrek-algoritme correct uitvoert. In dat geval moet dat deel van het leerlingmodel gelijk zijn aan het correcte procedurele netwerk. Met andere woorden: een bug is de kleinst mogelijke variatie op de correcte vaardigheid die de verrichtingen van een leerling nabootst.

2. Er werd onderscheid gemaakt tussen *primitieve* (enkelvoudige) en *samengestelde* bugs. Als de fouten van één leerling gezien kunnen worden als een combinatie van twee of meer andere fouten, dan moet het bijbehorende leerlingmodel deze combinatie weerspiegelen, doordat het de samenstelling is van de 'perturbaties' die nodig zijn om de deelfouten te modelleren.

Beide restricties leidden in de loop van het onderzoek tot herformuleringen van de opsplitsing van het aftrekalgoritme in deelprocedures.

Het is interessant om te onderzoeken hoe netwerken met een buggy procedure zich uiten in de antwoorden op opgaven. Het blijkt dat hier allerlei zaken op kunnen treden. Sommige bugs zullen in een groot deel van de mogelijke aftreksommen het correcte antwoord geven, bugs kunnen overlappen, in de zin dat ze in veel gevallen hetzelfde foute – of goede! – antwoord geven. Op zichzelf kleine bugs kunnen veel foute antwoorden geven, omdat ze door verschillende procedures in het netwerk worden aangeroepen. Deze dingen bemoeilijken het uiteindelijke doel: het stellen van een diagnose op grond van een aantal door een leerling gemaakte opgaven. Diagnose op grond van één opgave is onmogelijk. Bij de samenstelling van opgavenreeksen moet rekening worden gehouden met het 'onderscheidend vermogen' van de opgaven. Dit wil zeggen dat er voor elk paar buggy procedures er een opgave moet zijn die door deze procedures verschillend beantwoord wordt. Brown en Burton beweren dat het mogelijk is met een serie van 20 opgaven 1200 bugs te onderscheiden, waarbij elke niet-samengestelde bug tenminste drie maal voorkomt.

De procedurele netwerken werden gecodeerd in LISP, ieder knoop uit het netwerk werd een LISP-procedure. Elke variant van de correcte procedure werd een LISP-procedure op zich. Het programma had bij elke knoop daardoor de keuze tussen een aantal procedures, één correcte en verschillende (uiteenlopend van één tot vijftien) incorrecte.

2.3.2 BUGGY

Voordat men begon aan het automatiseren van het stellen van diagnoses, werd daarom eerst gedacht aan het gebruik van de buggy netwerken voor het simuleren van leerlingen. Zo ontstond het 'spel' BUGGY. BUGGY *kiest* een bug en maakt vervolgens aftreksommen volgens het bijbehorende procedurele netwerk. De spelers moeten de bug zien te raden en kunnen BUGGY aftreksommen opgeven om 'bewijsmateriaal' te verzamelen.

BUGGY werd allereerst voorgelegd aan studenten in de lerarenopleiding, met een positief resultaat. Veel studenten bleken moeite te hebben met het vinden van bugs, vooral omdat men geneigd was bevestiging te zoeken voor hypothesen en omdat veel studenten steeds opgaven van een bepaald type intikten, die niet voldoende onderscheidend vermogen hadden.

Vervolgens werd BUGGY aan leerlingen voorgelegd Ook hier waren de ervaringen positief. Aanvankelijk reageerden leerlingen in de trant van "Wow, how dumb en stupid this kid must be.", maar later gingen ze ook patronen in de fouten zien. Belangrijk hierin is ook het element van de rolwisseling: de leerling treedt nu als beoordelaar op.

2.3.3 Het naieve diagnostische systeem

De volgende stap was het gebruik van de procedurele netwerken in een half geautomatiseerd diagnostisch programma. Dit programma werkte volgens de methode van de *exhaustive search*: een reeks door één leerling gemaakte opgaven werd vergeleken met de resultaten van *alle* buggy procedures. Daarbij werden zogenaamde *bug comparison tables* gegenereerd. Figuur 3.6 bevat een voorbeeld van een bug comparison table.

Het programma werd gebruikt om reeksen opgaven van enkele duizenden leerlingen te analyseren. In eerste instantie werden de resultaten met de hand geanalyseerd, wat leidde tot het vinden van nieuwe primitieve bugs en combinaties van bugs. Het aantal primitieve bugs groeide daardoor aan tot 60. Daarnaast bleken er 270 combinaties van bugs voor te komen.
Later werd het programma uitgebreid; het moest ook enige diagnose kunnen plegen. Dit gebeurde door van elke bug het aantal keren te tellen dat in de tabel ***, *, ! of een afwijkend antwoord voorkwam. Op grond van berekeningen, die door Brown en Burton [15] 'involved' genoemd worden, werden leerlingen in groepen ingedeeld. Daarbij bleek dat een belangrijk deel van de fouten bij 40% van de leerlingen door één bug verklaard konden worden. Dit systeem werd het *Naive Diagnostic System* genoemd.

31	53	23	2003	203	1043	520	10214	909	10300	70001
-11	-20	-8	- 24	-109	-505	-467	-8375	-458	- 546	- 391
20	33	15	1979	94	538	53	1839	451	9754	69610
Student Answers:										
+	30	5	1	106	2	140	161	501	200	0
smaller from larger & 0-n=0 & n-0=0 & stop working when the bottom number runs out:										
*	***	***	***	***	***	***	***	***	***	***
n-0=0:										
*	***	!	!	4	508	!	!	!	!	!
stop working when the bottom digit becomes a blank:										
*	!	***	79	!	!	!	!	!	754	610
smaller from larger:										
*	!	25	2021	***	1542	147	18161	551	10246	70390
0-n=0:										
*	!	!	!	!	1038	60	1909	***	9800	70000

*** - bug predicts student's incorrect answer
* - bug predicts student's correct answer
! - bug predicts the correct answer, but student's answer is incorrect
\+ - student's answer is correct

figuur 3.6: Voorbeeld van een bug comparison table. Uit: [13]

2.3.4 DEBUGGY: off-line diagnose

Een volgende stap was de ontwikkeling van een volledig geautomatiseerd diagnostisch systeem. Dit moest aanzienlijk subtieler te werk gaan dan het oorspronkelijke systeem: door analyse van door leerlingen gemaakte opgaven was het aantal primitieve bugs inmiddels opgelopen tot 110, en werden er leerlingen gevonden wier fouten door combinaties van vier verschillende bugs verklaard konden worden. Dit maakte de exhaustive search onmogelijk.
Er werden twee varianten van het diagnostische systeem ontwikkeld: DEBUGGY was een 'off-line' systeem. Dit analyseerde een serie opgaven, die van te voren door een leerling gemaakt waren. IDEBUGGY (Interactive DEBUGGY) was de interactieve variant. Dit programma kon de leerlingen opgaven opgeven om bepaalde hypothesen te toetsen.

Burton [13] beschrijft een aantal complicerende factoren bij het stellen van een diagnose:

1. *Fouten, die niets met bugs te maken hebben.*
   Fouten van leerlingen kunnen allerlei oorzaken hebben. Onoplettendheid, vermoeidheid, een kortstondig vervallen in oude (foute) technieken, overschrijven van buren, of van andere opgaven op het blaadje, het zijn allemaal veroorzakers van fouten, die niet aan een consistente bug te wijten zijn. Het gevolg hiervan is, dat een hypothetische bug niet uitgesloten kan worden, omdat deze één of enkele opgaven mist: een veel subtieler beoordelingsmechanisme is nodig.
   Verder zijn er allerlei fouten die niet bij de procedurefouten in het rekenen horen,

maar die wel een belangrijke rol kunnen spelen. Hieronder valt bijvoorbeeld dyslexie; problemen met lezen en schrijven. Een bekende uiting hiervan is het schrijven in spiegelschrift, of het omkeren van de volgorde van de cijfers in een getal. De uitwerking van dergelijke fouten in een algoritme kan verstrekkend zijn. Brown en Burton gaan niet in op dergelijke bronnen van fouten. Ze veronderstellen correcte vaardigheden op het gebied van het lezen en schrijven van getallen.

2. *Rekenfouten.*
Leerlingen maken ook rekenfouten. Veelal zijn ze toevallig en daardoor eenmalig. Sommige leerlingen kunnen ook consequent bepaalde simpele aftrekkingen fout doen. Er zijn echter redenen om deze rekenfouten niet in de diagnostiek op te nemen. Ten eerste is het aantal van dergelijke 'bugs' zeer groot: bij het aftrek-algoritme zijn honderd eenvoudige aftrekkingen nodig, die uit de *facts table* moeten komen: $n-m$, waarbij $0 \leq m \leq 9$ en $m \leq n \leq m+9$. Dit geeft dus 900 mogelijke bugs. Ten tweede kan elke fout gemaakte opgave wel uit een reeks rekenfouten verklaard worden, wat tot gevolg heeft dat het diagnostisch systeem met dergelijke hypothesen overvoerd kan worden. Ten derde zijn rekenfouten veelal inconsistent.

3. *In speciale gevallen hoeven bugs niet op te treden.*
Veel bugs hebben speciale gevallen, waarin ze *niet* optreden. Zo is het mogelijk dat leerlingen, die moeite hebben met lenen, opgaven als 109–64 toch goed maken. Bij de berekening van 10–6 hoeft immers niet geleend te worden, deze aftrekking komt direct uit de tafels!

4. *Samengestelde fouten.*
Samenstellingen van bugs veroorzaken een hele reeks complicaties. Allereerst kunnen de samenstellingen aanleiding geven tot een combinatorische explosie. Samenstellingen van bugs kunnen echter ook geheel andere complicaties geven. Burton noemt de volgende:

   a. *Bugs kunnen speciale gevallen zijn van andere.*
   Een voorbeeld hiervan is de bug *0–n=n*, bug 4 in figuur 2.2. Deze kan gezien worden als een speciaal geval van de bug *grootste–kleinste* (bug 1 in figuur 2.2). Bij een leerling met de eerst bug 'scoort' de tweede bug dus ook altijd.

   b. *Bugs kunnen andere bugs uitsluiten.*
   De bug *grootste–kleinste* sluit alle bugs die met lenen te maken hebben uit; je hoeft immers nooit te lenen! Dergelijke samenstellingen moeten dus niet in beschouwing genomen worden.

   c. *Twee bugs kunnen elkaar 'uitdoven'.*
   Het kan voorkomen dat een leerling met meer dan één bug bij toeval een goed antwoord vindt, terwijl de opgave fout gemaakt zou worden als de leerling maar één van deze bugs zou hanteren. Een voorbeeld is de opgave 313–208, met de bugs *grootste–kleinste* en *n–0=0*. Samen geven de bugs het correcte antwoord, apart niet! Het gevolg hiervan is dat bepaalde bugs

niet kunnen worden uitgesloten op grond van het feit dat een leerling een goed antwoord heeft, terwijl elke bug apart een fout antwoord voorspelt.

d. *Samenstellingen van bugs moeten soms geordend worden.*
Sommige bugs hebben overlappende precondities, in die zin dat ze beide op één kolom in een aftrekking van toepassing kunnen zijn. In deze gevallen wordt de volgorde van belang, omdat het effect van deze bugs op het deel links verschillend kan zijn. De samenstelling {A, B} verschilt dan van {B, A}.

e. *Twee bugs kunnen gelijke symptomen geven, tenzij ze in combinatie met een derde bug optreden.*
Bugs kunnen dezelfde symptomen hebben. Zo maakt het voor het antwoord niet uit of je de minuend vermindert, dan wel de subtrahend vermeerdert. Zulke bugs kunnen, samengesteld met andere bugs, bijvoorbeeld met de bug *0–n=n*, wel verschillen geven.

DEBUGGY gebruikt heuristieken om de combinatorische explosie bij het samenstellen van bugs te vermijden. Het stellen van de diagnose gebeurt in een aantal stappen, waarbij in elke stap heuristieken gebruikt worden om het aantal mogelijkheden te beperken.

1. *Verzameling initiële hypothesen.*
DEBUGGY begint met het vergelijken van de antwoorden van de leerling met de antwoorden die door elk van de 110 primitieve bugs plus een twintigtal veel voorkomende samengestelde bugs gegeven worden. De bugs die één foutief antwoord van de leerling volledig voorspellen komen in de verzameling van initiële hypothesen.[2] Deze verzameling is de basis voor het beschouwen van samenstellingen. Zie ook figuur 3.6 voor een voorbeeld.

2. *Reductie van de initiële hypothesen.*
De ervaring leert dat de verzameling initiële hypothesen regelmatig zo'n 20 bugs bevat. Dit is teveel voor een volledig onderzoek van de samenstellingen; er zijn immers al 20^4 mogelijke combinaties van vier bugs. De verzameling initiële hypothesen wordt daarom gereduceerd door middel van *subsumption*: als bug A dezelfde fouten verklaart als bug B, en er zijn nog extra aanwijzingen voor A, dan wordt B verwijderd uit de verzameling hypothesen.

3. *Samenstellingen van initiële hypothesen.*
Van de resterende hypothesen worden samenstellingen bekeken, die uit maximaal vier primitieve bugs mogen bestaan. Het aantal samenstellingen wordt beperkt door de eerdergenoemde restricties te gebruiken (volgorde, uitsluiting).

2. Burton noemt dit *full-problem evidence*. Het is zo natuurlijk mogelijk dat er kansrijke hypothesen gemist worden. Burton relativeert dit door te hameren op een goede samenstelling van de testserie van opgaven. Er moeten voldoende eenvoudige opgaven tussen zitten, zodat bugs zich afzonderlijk kunnen manifesteren. Een andere mogelijkheid is *single-column evidence*, waarbij het optreden van een bug in één kolom van een aftrekking voldoende is om deze bug tot de hypothesen toe te laten.

Bovendien worden combinaties van bugs, die variaties zijn van een en dezelfde subprocedure, uitgesloten; dit zijn immers alternatieven.
Alle combinaties, die minder dan een bepaald percentage van de fouten voorspellen, vallen af. In DEBUGGY is dat percentage 40%.

4. *Coërcie.*
   Vervolgens wordt geprobeerd de missers van de overgebleven hypothesen te verklaren. Hierbij gaat de ruis – rekenfouten, verschrijvingen, en dergelijke – een rol spelen. De techniek wordt *coërcie* genoemd, omdat deze probeert de onverklaarde fouten in het keurslijf van een hypothese te dwingen. Er zijn drie soorten coërcies:
   a. Fouten in simpele aftrekkingen (tafels). Een afwijking van 2 wordt toegeschreven aan een rekenfout: 13–7=8 kan dus, 13–7=2 niet. De achtergrond hiervan is dat er veel rekenfouten gemaakt worden omdat de tafels niet volledig beheerst worden. Een leerling moet dan gaan tellen en komt dan makkelijk iets naast het juiste antwoord uit.
   b. Er wordt gekeken of verschillen verklaard kunnen worden uit de speciale gevallen, zoals de 10 links in het bovenste getal.
   c. Soms kunnen er tengevolge van bugs 'onmogelijke' situaties optreden. Als een leerling *altijd* leent, kan de uitkomst in een kolom groter dan 10 worden. In dergelijke gevallen worden er door de leerling allerlei reparaties toegepast, veelal zonder consistentie. Deze gevallen worden door het systeem gemarkeerd als voorspelbare afwijkingen.

5. *Kiezen van de diagnose.*
   De diagnose wordt gekozen op grond van een berekening, waarin het aantal missers en correcte voorspellingen gewogen wordt. Brown geeft geen precieze berekening, maar noemt de volgende punten: juiste voorspellingen tellen het zwaarst in *voordeel* van een hypothese, coërcies tellen minder. Als de leerling een goed antwoord gaf, terwijl de bug een fout voorspelde, telt dat het zwaarst in het *nadeel* van een hypothese.

Uit het bovenstaande wordt duidelijk dat het systeem naast de informatie over bugs *zelf*, de buggy subprocedures ook een aanzienlijke hoeveelheid informatie over samenstellingen van bugs moet bevatten, zoals informatie over uitsluiting en ordening. Verder moet geregistreerd worden welke bugs speciale gevallen hebben. De artikelen geven echter geen beschrijving van de wijze waarop deze informatie is gerepresenteerd.

2.3.5 IDEBUGGY: Interaktieve diagnose

Bij IDEBUGGY ontstaan geheel nieuwe mogelijkheden doordat het systeem interactief is en dus extra informatie kan opvragen. De belangrijkste manier van informatie vragen is het opgeven van een opgave die onderscheid maakt tussen twee concurrerende hypothesen. Dit legt echter wel beperkingen op aan het aantal hypothesen die er door het systeem op na gehouden worden; je kunt immers geen eindeloze reeks onderscheidende opgaven opgeven. Daarom is de verzameling hypothesen gesplitst in een 'actief' deel, dat onderzocht wordt, en een passief deel, met hypothesen die

(tijdelijk) terzijde gelegd zijn.
IDEBUGGY werkt in een cyclus en bij elke doorgang wordt een van de volgende operaties uitgevoerd:

1. Geef een willekeurige opgave op aan de leerling.
2. Geef de leerling een opgave die onderscheid maakt tussen twee actieve hypothesen.
3. Genereer nieuwe hypothesen door actieve hypothesen samen te stellen met andere mal-rules.
4. Neem ter zijde gelegde hypothesen opnieuw in beschouwing.
5. Neem nieuwe (zeldzame) bugs in beschouwing.
6. Geef een diagnose en stop.
7. Geef 't op.

Burton geeft een weinig overzichtelijke beschrijving van de procedure die een keuze maakt tussen de bovenstaande acties. Hij geeft vooral voorbeelden: als er twee actieve hypothesen zijn, geef dan een onderscheidende opgave. Als er slechts één is, probeer deze samen te stellen met terzijde gelegde hypothesen, of met veel voorkomende bugs, of probeer deze hypothese te falsificeren, enz.

Interessant aan IDEBUGGY is het probleem van het genereren van opgaven. IDEBUGGY heeft een aantal verschillende opgaven-generatoren. Elk van deze generatoren produceert opgaven van een bepaald type, zoals 'opgaven waarbij lenen nodig is', 'opgaven met een nul in de minuend', 'opgaven waarbij van twee nullen op rij geleend moet worden' en 'willekeurige opgaven'. Met elke primitieve bug worden een aantal generatoren geassocieerd, namelijk degene die opgaven genereren waarin de bug tot uiting komt. Onderscheidende opgaven worden gegenereerd door deze generatoren opgaven te laten produceren totdat er een opgave verschijnt die door de twee bugs verschillend gemaakt wordt.

2.3.6 Andere representaties: productiesystemen

Naast het werk van Brown en Burton aan BUGGY is er nog meer onderzoek gedaan naar de problematiek van fouten in het aftrek-algoritme. Het gaat hier vooral om verschillende representaties van het algoritme en de bugs. Deze onderzoeken zijn niet zo diep gegaan als het onderzoek aan BUGGY, in die zin dat er in de literatuur geen in de praktijk werkende systemen worden beschreven.

Young en O'Shea [16] beschrijven het gebruik van productiesystemen voor het ontwikkelen van een model voor aftrekkingen. Hun productieregels zijn geschreven in OPS2, een taal voor het schrijven van data-driven (forward-chaining) productiesystemen. Figuur 3.7 bevat de productieregels voor het aftrekken en een voorbeeld van een productie in OPS2.

Het productiesysteem verdient enige toelichting. OPS2 bestaat uit drie onderdelen: *working memory*, dat de data bevat, *production memory*, dat de productieregels bevat, en een *inference engine*, die in een cyclus een toepasbare productieregel selecteert en afvuurt. Bij de selectie van een productieregel worden door OPS2 verschillende criteria

```
FD:     M =m, S =s    ⇒  FindDiff, NextColumn
B2A:    S > M         ⇒  Borrow
BS1:    Borrow        ⇒  *AddTenToM
BS2:    Borrow        ⇒  *Decrement
CM:     M =m, S =s    ⇒  *Compare
IN:     ProcessColumn ⇒  *ReadMandS
TS:     FindDiff      ⇒  *TakeAbsDiff
NXT:    NextColumn    ⇒  *ShiftLeft, ProcessColumn
WA:     Result =x     ⇒  *Write =x
DONE:   NoMore        ⇒  *HALT
B2C:    S = M         ⇒  Result 0, NextColumn
AC:     Result 1 =x   ⇒  *Carry, Result =x
```

Step	WM contents (in order of recency)	Rule fired	Action taken, or element asserted
	(PROCESSCOLUMN)		
1.		IN	Do *ReadMandS
			Assert (M 4)
			Assert (S 8)
	(S 8) (M 4) (PROCESSCOLUMN)		
2.		CM	Do *Compare
			Assert (S > M)
	(S > M (S 8) (M 4) (PROCESSCOLUMN)		
3.		B2A	Assert (BORROW)
	(BORROW) (S > M) (S 8) (M 4) (PROCESSCOLUMN)		
4.		BS1 (say)	Do *AddTenToM
5.		BS2 (say)	Do *Decrement
6.		FD	Assert (NEXTCOLUMN)
			Assert (FINDDIFF)
	(FINDDIFF) (NEXTCOLUMN) (BORROW) (S > M) (S 8) . . .		
7.		TS	Do *TakeAbsDiff
			Assert (RESULT 6)
	(RESULT 6) (FINDDIFF) (NEXTCOLUMN) (BORROW) (S > M) . . .		

figuur 3.7: OPS2-productieregels voor aftrekking en voorbeeld van een afleiding. Uit: [16]

gehanteerd, waaronder *recency* (probeer een regel af te vuren die op het laatst geproduceerde data-element werkt) en *specificity* (specifieke regels gaan vóór algemene). Zie Brownston [7] voor details.

Het conditiedeel van een productieregel bevat referenties aan data in het working memory. Als de conditie unificeert met data in het working memory, kan de regel afvuren. Het actiedeel bevat procedures (acties) die uitgevoerd kunnen worden. Deze procedures zijn in figuur 3.7 gemerkt met een *. Daarnaast bevat het actiedeel data-elementen, die bij afvuren aan het working memory toegevoegd worden.

In principe volgt OPS2 een data-driven strategy bij inference. In de praktijk is er echter eerder sprake van een gemengde data- en goal-driven strategy. Een aantal van de data-elementen, zoals (M 3),[3] (RESULT 1) en (M>=S), zijn gegevens, die betrekking

hebben op de aftrekking die uitgevoerd wordt. Andere data-elementen, zoals (Borrow), (FindDiff) en (ProcessColumn), zijn eerder te zien als goals, omdat ze er alleen toe dienen het afvuren van andere regels mogelijk te maken.

Young en O'Shea geven een verzameling van ongeveer twintig productieregels waarmee het volledige aftrek-algoritme gesimuleerd kan worden. Ze beweren dat veel van de 60 oorspronkelijke primitieve bugs van Brown en Burton en de samenstellingen daarvan gesimuleerd kunnen worden door één of meer productieregels weg te laten en enkele 'buggy' productieregels toe te voegen. Ongeveer tien van deze buggy productieregels zijn voldoende om een groot deel van de bugs van Brown en Burton te kunnen 'maken'. Op deze manier wordt een eenvoudiger representatie van de bugs gegeven.

Daarnaast geven Young en O'Shea ook kritiek op BUGGY van meer fundamentele aard. De procedurele netwerken kunnen weliswaar aangepast worden door een subprocedure te vervangen door een buggy variant, maar de structuur van het netwerk, de indeling in doelen en subdoelen, wordt daardoor niet gewijzigd. Ze merken verder op dat kinderen vaak niet werken volgens een structuur van doelen, maar volgens herkenning van patronen in de opgaven. Daardoor kunnen allerlei 'kortsluitingen' ontstaan, die delen van het netwerk overslaan. Young en O'Shea geven daarbij het voorbeeld dat gelijke cijfers boven en onder soms direct het reultaat 0 geven, zonder dat daar bugs als *altijd lenen* tussen kunnen komen.
Verder twijfelen zij aan de bewering van Brown en Burton dat de buggy procedures verklaren *waarom* leerlingen bepaalde fouten maken. De bugs *simuleren* weliswaar de fouten, maar geven geen redenen. Dit idee wordt versterkt door het feit dat BUGGY zeer veel bugs kent, waarvan een groot deel sterk op elkaar lijkt.

2.3.7 Andere representaties: repair-theory.

De repair theory van Brown en van Lehn [17] gaat uit van een geheel andere benadering. De onderliggende gedachte is dat leerlingen zelden ophouden te werken aan een opgave, als ze niet meer verder kunnen. In plaats daarvan gebruiken ze een 'lapmiddel' om er toch maar een antwoord uit te krijgen. Deze lapmiddelen worden vaak verkregen uit analogieën met andere technieken, of andere opgaven. Ze kunnen sterk variëren en zijn vaak ook niet consistent.
Deze theorie geeft ook een verklaring voor het grote aantal, vaak ietwat exotische, fouten, met name bij het lenen van nullen. Verder is het in deze theorie niet voldoende om een buggy procedure te maken die de leerling simuleert: de buggy procedure is immers op zichzelf niet zo belangrijk, het gaat om de reden van het inzetten van 'lapmiddelen'. Over een concretisering van deze theorie in een geautomatiseerd systeem is in de

3. M staat voor minuend, het cijfer boven, S voor substrahend, het cijfer onder. Een = in de conditie van een duidt productieregel een variabele aan. Verder lijkt het dat de gegeven set productieregels wat slordig omgaat met getallen die uit verscheidene cijfers bestaan: zo is het onduidelijk waar de actie *decrement* de cijfers links vandaan moet halen.

literatuur (nog) niets te vinden.

2.4 Een ITS voor eenvoudige algebra

De tweede case study heeft betrekking op het oplossen van lineaire vergelijkingen in één variabele. Het Leeds Modelling System (LMS) wordt door Sleeman in verschillende artikelen beschreven [14, 18, 19]. LMS is ontwikkeld om fouten op te sporen in de door leerlingen gevolgde procedures bij het oplossen van dergelijke vergelijkingen. LMS is een interactief systeem: een leerling die met LMS werkt krijgt een lineaire vergelijking voorgeschoteld en kan herschrijvingen van de vergelijking intikken, zodat ook de tussenstappen in de oplossing geregistreerd kunnen worden. Dit stopt als de leerling een oplossing intikt (x=...), of aangeeft dat hij of zij klaar is. LMS voorziet de stappen niet van commentaar, maar beperkt zich tot een 'thank you' na afloop.
In de representatie van de correcte en foute procedures wordt gebruikt gemaakt van *rules* en *mal-rules*. Vanwege het onderwerp en de representatie is dit onderzoek bijzonder relevant.

2.4.1 Productieregels voor lineaire vergelijkingen

Het Leeds Modelling System is gebaseerd op productieregels. Alle productieregels hebben in hun conditie-deel een vorm van (een deel van) een vergelijking, het actiedeel bevat een herschrijving van deze vergelijking. Het working memory bevat slechts één data-element, de te onderzoeken vergelijking.
Een productiesysteem (PS) is een geordende lijst van productieregels. In een *recognize-act cyclus* wordt steeds de *eerste* productieregel in het PS opgezocht, waarvan het conditie-deel op de vergelijking in het working memory past. Deze wordt afgevuurd, de oude vergelijking wordt uit het working memory verwijderd, de door het actiedeel van de productieregel herschreven vergelijking wordt erin gezet. Als de conditie van een productieregel op verschillende plaatsen in de vergelijking past, wordt de meest linkse match genomen.
Figuur 3.8 bevat een overzicht van een aantal productieregels. De in de tabel voorkomende *niveaus* en *task-sets* zullen in paragraaf 4.4.2 besproken worden.
De FAIL-regel staat achteraan de lijst, dit heeft tot gevolg dat deze de cyclus stopt als geen enkele regel meer kan worden afgevuurd. De ANS-regel staat direct voor de FAIL-regel en stopt de cyclus als de vergelijking is gereduceerd tot een antwoord (een getal of een breuk). Deze twee regels worden verondersteld in alle productiesystemen aan het eind van de lijst met regels opgenomen te zijn, daarom worden ze in de notatie veelal weggelaten.

naam	niveau	conditie	actie	typische 'task-set'
FAIL	-	[willekeurig]	[geef 't op]	-
ANS	-	(M) of (M/N)	[geef antwoord]	-
FIN	1	(X = M/N)	(M/N)	*x=7/5*
SOLVE	2	(M*X = N)	(X = N/M) or (INFINITY)	*5*x=7*
ADDSUB	3	(lhs M{+|-}N rhs)	(lhs [evaluated] rhs)	*5*x=4+3*
MULT	4	(lhs M*N rhs)	(lhs [evaluated] rhs)	*3*x=2*2*
XADDSUB	5	(lhs M*X{+|-}N*X rhs)	(lhs (M{+|-}N)*X rhs)	*2*x+3*x=10*
NTORHS	6	(lhs {+|-}N = rhs)	(lhs = rhs {-|+}N)	*2*x+4=16*
REARRANGE	7	(lhs {+|-}M{+|-}N*X rhs)	(lhs {+|-}N*X{+|-}M rhs)	*4+2*x=16*
XTOLHS	8	(lhs = {+|-}M*X rhs)	(lhs {-|+}M*X = rhs)	*4*x=2*x+3*
BRA1	9	(lhs (N) rhs)	(lhs N rhs)	*2*x=5*(3+1)*
BRA2	10	(lhs M*(N*X{+|-}P) rhs)	(lhs M*N*X{+|-}M*P rhs)	*6*x=4*(2*x+3)*

figuur 3.8: Enkele productieregels voor het oplossen van lineaire vergelijkingen. Hierin staan L, M en P voor getallen, X is de variabele. Accolades duiden een keuze aan. Uit [18]

Door de eenvoud van het working memory is het niet nodig deze vorm van inferentie te zien als een data-driven productiesysteem. Een en ander kan eenvoudig gesimuleerd worden door een paar Prolog-clauses, waarbij vergelijkingen als argument van het predicaat los_op worden meegegeven:

```
los_op(Vergelijking, Oplossing):-
        vindt_herschrijving(Vergelijking, NVergelijking),
        los_op(NVergelijking, Oplossing).

los_op(Vergelijking, Vergelijking):-
        antwoord_vorm(Vergelijking).
```

De ANS-regel zit in de tweede clause, de FAIL-regel kan worden weggelaten, zodat Prolog vanzelf 'failure' uitspreekt als geen enkele herschrijving meer toegepast kan worden.

Het is van belang op te merken dat matching in LMS plaatsvindt op het niveau van strings; vergelijkingen zijn geen termen met functoren en argumenten, maar strings. De regels voor ADDSUB en MULT kunnen dus allebei toegepast worden op de formule 3*5+8! Dit maakt de volgorde van de productieregels uitermate belangrijk. Ook kunnen de variabelen *lhs* en *rhs* rustig beginnen of eindigen met een infix-operator. Redenen voor deze opmerkelijke aanpak worden niet gegeven, naar mijn mening is het een zeer zwak punt in LMS.

LMS gebruikt *mal-rules* bij het genereren van leerlingmodellen (zie paragraaf 2.2.2). Met elke productieregel wordt een aantal mal-rules geassociëerd die een foute herschrijving van een vergelijking produceren. Een aantal van deze mal-rules is weergegeven in figuur 3.9. Merk op dat de conditie-delen van de mal-rules overeenstemmen met die van de correcte productieregels.
Daarnaast kunnen er productiesystemen, die fouten maken bij het oplossen van vergelijkingen, gegenereerd worden door te knoeien met de volgorde van de

naam	niveau	conditie	actie
MSOLVE	2	(M*X = N)	(X = M/N)
MNTORHS	6	(lhs {+/-}N = rhs)	(lhs = rhs {+/-}N)
M2NTORHS	6	(lhs1 {+/-}N*X lhs2 = rhs)	(lhs1 + *X lhs2 = rhs {-/+}N)
M3NTORHS	6	(lhs1 {+/-}N*X lhs2 = rhs)	(lhs1 *X lhs2 = rhs {+/-}N)
MXTOLHS	8	(lhs = {+/-}M*X rhs)	(lhs {+/-}M*X = rhs)
M1BRA2	10	(lhs M*(N*X{+/-}P) rhs)	(lhs M*N*X{+/-}P rhs)
M2BRA2	10	(lhs M*(N*X{+/-}P) rhs)	(lhs M*N*X{+/-}M{+/-}P rhs)

figuur 3.9: Enkele mal-rules voor het oplossen van lineaire vergelijkingen. Uit [18]

productieregels. Een productiesysteem dat de regel ADDSUB voor MULT heeft, zal optellen voor vermenigvuldigen laten gaan. Figuur 3.10 bevat een aantal afleidingen, zowel van een correct productiesysteem als van enkele foute. De afleidingen moeten gelezen worden als 'PS *<regels>* EVALUATES *<formule>* TO *<afleiding>* *<eindtoestand>*. Merk op dat het 'correcte' productiesysteem, in het eerste voorbeeld, alleen werkt voor vergelijkingen van het soort dat in het voorbeeld genoemd wordt, of eenvoudiger vergelijkingen. Vergelijkingen met haakjes worden niet door dit productiesysteem opgelost.

```
PS (MULT ADDSUB XADDSUB NTORHS SOLVE FIN2 FIN) EVALUATES
(2*X+4*X+4=16) TO
RULE XADDSUB FIRES-RESULT IS (6*X+4=16)
RULE NTORHS FIRES-RESULT IS (6*X=16-4)
RULE ADDSUB FIRES-RESULT IS (6*X=12)
RULE SOLVE FIRES-RESULT IS (1*X=12//6)
RULE FIN2 FIRES-RESULT IS (2)
COMPLETE ANSWER IS 2

PS (MNTORHS MULT ADDSUB XADDSUB SOLVE FIN2 FIN) EVALUA
(2*X+4*X+4=16) TO
RULE MNTORHS FIRES-RESULT IS (2*X+4*X=16+4)
RULE ADDSUB FIRES-RESULT IS (2*X+4*X=20)
RULE XADDSUB FIRES-RESULT IS (6*X=20)
RULE SOLVE FIRES-RESULT IS (1*X=20//6)
RULE FIN2 FIRES-RESULT IS ((20 6))
COMPLETE ANSWER IS (20 6)

PS (M3NTORHS MULT ADDSUB XADDSUB SOLVE FIN2 FIN) EVALUATES
(2*X+4*X+4=16) TO
RULE M3NTORHS FIRES-RESULT IS (2*X*X+4=16+4)
RULE M3NTORHS FIRES-RESULT IS (2*X*X=16+4+4)
RULE ADDSUB FIRES-RESULT IS (2*X*X=20+4)
RULE ADDSUB FIRES-RESULT IS (2*X*X=24)
UNABLE TO FULLY EVALUATE THIS PROBLEM—FINAL STATE IS
(2*X*X=24)
```

figuur 3.10: Enkele producties. Uit: [14]

In principe kunnen er nu leerlingmodellen gegenereerd worden door lijsten van correcte productieregels en mal-rules te genereren. De aldus verkregen productiesystemen kunnen op opgaven worden losgelaten om te onderzoeken of ze hetzelfde 'gedrag' produceren. Het aantal productiesystemen is echter bijzonder groot. Aan deze combinatorische explosie, en de beheersing ervan, zal in de volgende paragraaf aandacht

besteed worden.

2.4.2 Task-sets: het reduceren van de zoek-ruimte

Sleeman en Smith [19] besteden enige aandacht aan de combinatoriek die gepaard gaat met het beschouwen van productiesystemen met mogelijk grote aantallen mal-rules. Ze baseren hun berekeningen op de volgende vooronderstellingen:

1. Op het domein zijn precies r productieregels van toepassing.
2. Voor elk van de r productieregels zijn er m mal-rules.
3. Een productiesysteem bestaat uit ten hoogste r productieregels.
4. De volgorde van de productieregels in een productiesysteem is van belang.

Het totaal aantal mogelijke productiesystemen, $N_{tot}(r,m)$, wordt nu eenvoudig gegeven door de formule,

$$N_{tot}(r,m)=\sum_{k=0}^{r}\frac{(r(1+m))!}{(r(1+m)-k)!}$$

waarin k het aantal regels in het productiesysteem is. Dit aantal is natuurlijk bijzonder groot, ook voor kleine r en m en daarom dient het aantal mogelijkheden beperkt te worden. Een eerste reductie wordt bereikt door de extra veronderstelling dat uit elke groep van één correcte regel met de m daarmee geassociëerde mal-rules slechts één regel in het productiesysteem kan worden opgenomen. Dit betekent dat er na het kiezen van de eerste regel 'slechts' $(r-1)(1+m)$ mogelijkheden over zijn, in plaats van $r(1+m)-1$. Het aantal productiesystemen wordt nu gereduceerd tot:

$$N_{groep}(r,m)=\sum_{k=0}^{r}\frac{r!}{(r-k)!}*(1+m)^k$$

waarin het linkerdeel van het product het aantal combinaties met volgorde van k correcte regels is. De factor $(1+m)^k$ komt voort uit het feit dat we voor elk van deze correcte regels kunnen kiezen uit de regel zelf en zijn m mal-rules. Deze beperking is aanzienlijk, maar ontoereikend: bijvoorbeeld $N_{tot}(6,2)=14472900$ en $N_{groep}(6,2)=732528$. Ook in systemen waar de volgorde van de regels niet van belang is blijft het aantal aanzienlijk: in beide gevallen scheelt dat een factor die ongeveer gelijk is aan 6!!

Deze aantallen maken een andere aanpak noodzakelijk. In LMS werd een enorme reductie bereikt door het introduceren van *niveaus*. Deze aanpak veronderstelt dat met een deel van de r correcte regels een PS gemaakt kan worden dat een deel van de taken, eenvoudiger taken, uit kan voeren. Zo ontstaat een reeks productiesystemen (PS_i), waarbij PS_{i+1} uit PS_i ontstaat door toevoeging van een aantal regels en PS_{i+1} een grotere klasse van problemen aankan dan PS_i. Deze klasse van problemen wordt door Sleeman een *task-set* genoemd. In LMS kon zo'n reeks productiesystemen gemaakt worden, waarbij steeds één regel wordt toegevoegd. Daardoor werd elke regel geassocieerd met een bepaald niveau; het niveau van een regel en een representant van de bijbehorende task-set zijn in de tabel in figuur 3.8 opgenomen.

In feite is dit een zeer rigoreuze manier van 'pruning': op niveau 1 wordt vastgesteld

welke van de $1+m$ regels gebruikt wordt, alleen in deze tak van de zoekboom wordt verder gezocht, alle PS-en die een van de m andere regels bevatten worden uit de zoekboom gesnoeid. Het is eenvoudig in te zien dat bij de toevoeging van één regel aan een PS dat al k regels bevat er $(k+1)(1+m)$ mogelijkheden zijn, het ligt immers vast welke *correcte* regel moet worden toegevoegd. De plaats waar deze regel of mal-rule wordt ingevoegd is verantwoordelijk voor de factor $k+1$. Bij het genereren van een PS met r regels dient deze toevoeging r maal herhaald te worden, zodat:

$$N_{niveau}(r,m)=\sum_{k=0}^{r-1}(k+1)(1+m)=\tfrac{1}{2}(1+m)r(r+1)$$

Dit reduceert $N_{niveau}(6,2)$ tot 63.

Het stellen van de diagnose gebeurt nu stapsgewijs, dat wil zeggen dat LMS pas opgaven van niveau L opgeeft als het model voor niveau $L-1$ bekend is. Bij het vinden van het model wordt gebruik gemaakt van de *generate and test* methode. De leerling krijgt opgaven van een bepaald niveau totdat er maar één kandidaat-model over is, of de aanwezige 'experimenter' aangeeft dat de resterende kandidaten gelijk zijn. Modelling gaat niet verder dan het niveau waarop een mal-rule optreedt. Er worden geen pogingen ondernomen om *ruis* (rekenfouten, 'performance lapses', e.d.) te verklaren.[4]

Het hanteren van een model met niveaus stelt een aantal eisen aan het domein. Allereerst moet er een stapsgewijze opbouw in het domein zijn, die het idee van het toevoegen van regels, om een grotere klasse van problemen op te kunnen lossen, mogelijk maakt. Verder moeten mal-rules goed plaatsbaar zijn, ze moeten kunnen worden gekoppeld aan een bepaald niveau. De mal-rules M2NTRHS en M3NTORHS zijn in dit kader interessant: hoewel ze betrekking hebben op termen die x bevatten, horen ze bij het niveau van het 'getallen naar rechts brengen'. Verder wordt er van uitgegaan dat een leerling die de vaardigheden van een bepaald niveau lijkt te beheersen, daarmee op een hoger niveau – een ingewikkelder probleem – ook geen fouten maakt. Dit probleem bleek in een aantal gevallen echter wel op te treden. Dit leidde tot een niet beschreven 'reformulation of the algorithm'.

2.4.3 Experimenten met leerlingen

Met LMS zijn verschillende experimenten met leerlingen uitgevoerd, waarbij het systeem voortdurend uitgebreid en verbeterd werd. Deze experimenten betroffen groepen van enkele tientallen leerlingen. Met iedere leerling vonden twee sessies plaats; een sessie met LMS en een sessie met een expert. Deze tweede sessie had tot doel de diagnose van LMS en toetsen en zonodig tot een uitbreiding van de verzameling mal-rules te komen. Net als bij BUGGY, leidden de experimenten ook hier tot een aanzienlijke uitbreiding van de verzameling mal-rules.

4. Ook bij LMS is de beschrijving van het algoritme voor het genereren van het leerlingmodel zeer onduidelijk. De beschreven procedure is meer afgeleid uit 'circumstancial evidence' dan uit de beschrijving.

Op grond van deze experimenten geeft Sleeman [18] een indeling van de mogelijke fouten in vier categorieën: *manipulatieve fouten, parseerfouten, administratieve fouten* en *willekeurige fouten.*
Onder de manipulatieve fouten vallen fouten als het herleiden van $mx = n$ tot $x = m/n$, het vergeten van de tekenwisseling bij het overbrengen van een term naar de andere kant van de vergelijking en fouten waarbij verwisseling of verwarring van operanden optreedt.
In de categorie parseerfouten vindt men een grote verzameling, vaak exotische, fouten. Hieronder valt het lezen van $2x$ als $2+x$ met afgeleide varianten, fouten in de volgorde van bewerkingen, fouten bij het 'wegwerken' van haakjes en complexe fouten als het herleiden van $3x = 2x + 5$ tot $x+x = 2+5+3$.
Deze bug is in meer dan een opzicht interessant. Hij kan op procedureel niveau gesimuleerd worden door de mal-rules *herleidt nx tot n+x* en *vergeet tekenwisseling bij het overbrengen van termen naar de andere kant van de vergelijking.* Deze mal-rules simuleren weliswaar de procedure van de leerling, maar geven geen verklaring voor de bug. Deze moet waarschijnlijk op een conceptueel niveau gezocht worden: de leerling voert de opdracht *breng de x'en naar links en de getallen naar rechts* letterlijk uit en mist elementaire begrippen van de algebra.

Reken- en schrijffouten worden tot de administratieve fouten gerekend, en de categorie willekeurige fouten bevat die fouten, die niet consequent gehanteerd worden.

Verder werden er ook 'moeilijk plaatsbare' fouten geconstateerd, zoals varianten van het 'oplossingen zoeken door waarden voor x te proberen' waarbij in vergelijkingen met twee x'n beide x'n een verschillende waarde kregen: $3x+4x=98$ werd herleid tot $3*22+4*8=66+32=98$.
Als laatste voorbeeld geven we een fout die een goed beeld geeft van de exotische procedures die soms toegepast worden:

$3x + 2x = 12$
$3 * 2 + 2 + 4 = 12$
$x=2$
$x=4$

De leerling die deze oplossing gaf motiveerde deze als volgt: "Ik maak de eerste x gelijk aan 2 en bereken 3*2+2, dan moet de tweede x gelijk zijn aan 12-8, dat is dus 4." Deze leerling paste die procedure zeer consequent toe.

2.4.4 Kritische beschouwing van LMS

LMS is een geslaagd experiment in die zin, dat men er in geslaagd is een systeem te ontwikkelen dat een groot deel van de fouten bij het oplossen van lineaire vergelijkingen in één variabele kan modelleren, terwijl de combinatorische explosie in de hand kan worden gehouden. Er zijn echter een aantal kritische kanttekeningen bij te plaatsen. We bespreken eerst de kennisrepresentatie, daarna de niveaus en task-sets.

Representatie van algebraische kennis
Als we LMS beschouwen aan de hand van de door Dede gegeven driedeling in soorten

kennis – declaratieve, procedurele en metacognitieve (strategische) kennis – dan blijkt LMS alleen over een beperkte procedurele kennis te beschikken. Declaratieve kennis – *Wat zijn variabelen? Wat is de bedoeling van het oplossen van vergelijkingen? Wat is de structuur van algebraische expressies?* – ontbreekt volledig, mede omdat de vergelijkingen met behulp van strings gerepresenteerd worden, in plaats van een structuur zoals de termen van Prolog.
LMS bevat alleen de bouwstenen van procedures, namelijk de ondeelbaar kleine manipulaties die nodig zijn bij het oplossen van lineaire vergelijkingen. Procedures van een hoger niveau, die deze bouwstenen in een (doel)structuur passen ontbreken. Er is dus geen procedure *gelijksoortige termen optellen* of *term naar de andere kant brengen.* In plaats daarvan zijn de bouwstenen geordend en verloopt het proces volgens het principe 'wie het eerst komt, die het eerst maalt'.
Omdat het domein zeer beperkt is, is ook de metacognitieve kennis beperkt. De oplossingsstrategie – isolatie – valt al samen met een procedure voor het bepalen van een oplossing. Andere ideeën ontbreken echter ook, zoals het idee dat het niet uitmaakt of je eerst de *x*'en naar links brengt en daarna de getallen naar rechts, of andersom, of het idee dat links en rechts in de oplossingsprocedure verwisseld mogen worden. Dergelijke ideeën zijn echter wel vaak belangrijk.

Een deel van deze gebreken is een direct gevolg van de reeds gewraakte representatie: vergelijkingen als strings en daarnaast regels, waarin volgorde van belang is.
De lineaire vergelijkingen worden door middel van strings gerepresenteerd en de conditiedelen van de productieregels bevatten stringpatronen. Pattern-matching bepaalt of een regel afgevuurd kan worden. Dit leidt tot allerlei merkwaardige beperkingen. Zo kan de regel NTORHS (zie figuur 2.9) alleen getallen die direct voor de '=' staan naar rechts brengen. Het systeem kan geen lineaire vergelijkingen in de variabele *y* aan. Dit maakt de toevoeging van merkwaardige regels zoals REARRANGE en BRA1 noodzakelijk.
Het is ook pattern-matching dat de volgorde van de productieregels zo belangrijk maakt. Beschouw bijvoorbeeld de eenvoudige 'formule' *3+4*5*. De regels ADDSUB en MULT passen allebei op deze formule, de matches zijn respectievelijk *3+4* en *4*5*. Alleen door te specificeren dat MULT voor ADDSUB gaat, kan een foute evaluatie worden voorkomen. In een representatie die de structuur weerspiegelt, zoals de termen van Prolog, geeft de regel ADDSUB geen match; rechts van de '+' staat immers geen getal, maar een product!
Introductie van haakjes maakt het geheel nog vreemder. Beschouw de 'formule' *3+4*(5+6)*. MULT past nu niet en dus vuurt ADDSUB af. Deze neemt de meest linker match, wat resulteert in *7*(5+6)*. Dit maakt introductie van een nieuwe regel nodig. Sleeman voert de regel PADD op die match't met *lhs (N+M rhs*. Deze dient in de volgorde tussen MULT en ADDSUB te komen. Soortgelijke problemen dienen zich ook aan bij de lineaire vergelijkingen. Op niveaus hoger dan 10 verschijnen er regels met exotische namen als ADDSUB&BRA2 en MULT&XADDSUB. Het geheel wordt op deze manier natuurlijk uitermate *on*wiskundig. Het probleem schuilt in het ontbreken van de eerdergenoemde declaratieve en metacognitieve kennis.
Het belang van de volgorde van regels leidt ook tot andere moelijkheden. Soms leidt een verwisseling van twee regels tot een fout PS, zoals de verwisseling van MULT en

ADDSUB en soms ook niet, bijvoorbeeld bij verwisseling van XTOLHS en NTORHS. Het maakt immers niet uit of je eerst de x'en naar links brengt of eerst de getallen naar rechts. In de praktijk wordt er maar één PS als het ideale, correcte, gezien, waardoor de oplossingsmethode veel te nauw lijkt te worden vastgelegd.

Concluderend lijkt het dat een representatie met termen die de structuur van de vergelijking weerspiegelen en regels waarbij de volgorde niet van belang is de voorkeur verdient boven de representatie in LMS. Hiervoor zijn de volgende redenen aan te voeren:

1. In wiskundige problemen is de volgorde van handelingen vaak onbelangrijk. Nondeterminisme in de volgorde waarin regels worden toegepast leidt dan tot een representatie van de verschillende manieren waarop een oplossing gevonden kan worden.

2. In een representatie met (Prolog)termen staan regels voor algebraische operaties. In LMS is dit vaak niet het geval. Veel regels zijn nodig om het afvuren van andere regels mogelijk te maken, of juist te voorkomen. Feitelijk voegen ze niets toe aan het arsenaal algebraische manipulaties. Dit geldt in het bijzonder voor regels als PADD en BRA1. Dit soort regels en de invloed, die de volgorde van regels heeft, maken de werking van de productiesystemen van LMS zeer ondoorzichtig.

3. De introductie van nieuwe regels is in LMS problematisch, omdat de plaats in de lijst van bestaande regels gezocht moet worden. Dit heeft tot gevolg dat voor elke nieuwe task-set het nieuwe PS volledig gegeven moet worden; men kan niet volstaan met de toevoeging alleen. Dit geldt niet voor systemen waarin de volgorde niet van belang is.

4. In LMS zijn er twee soorten fouten: mal-rules en 'mal-order' van correcte regels. In een systeem zonder volgorde moeten alle fouten gerepresenteerd worden door mal-rules. Dit leidt tot een grotere uniformiteit in de representatie van fouten. Zo kan de verkeerde volgorde (ADDSUB MULT) in een systeem zonder volgorde gerepresenteerd worden door de mal-rule *(de leerling) herleidt a+b*c tot (a+b)*c.*

5. Door eliminatie van volgorde wordt de combinatoriek van het zoeken zeer beperkt. Weliswaar moeten er mal-rules worden toegevoegd (punt 4), maar er worden ook regels verwijderd (punt 2).

Aan een representatie met gestructureerde termen kleven ook problemen. Zo kunnen leerlingen syntactisch incorrecte herleidingen geven. Dit is bijvoorbeeld het geval bij de mal-rule M2NTORHS.[5] Daarnaast zijn er natuurlijk een aantal fouten die gebaseerd

5. Mogelijk wordt deze fout veroorzaakt door LMS zelf. LMS schrijft producten met een '*', terwijl de maaltekens tussen getallen en variabelen normaliter niet geschreven worden!

zijn op de *uitgeschreven* representatie van formules, zoals fouten waarbij (ongeschreven) maaltekens een rol spelen. Een voorbeeld hiervan is de bekende fout dat het substitueren van $x=3$ in de formule $2x$ als resultaat *23* oplevert. Mogelijk zijn een aantal van dergelijke fouten moeilijker te representeren.

Niveaus en task-sets.
Het gebruik van niveaus lijkt een goede methode om het zoeken te beperken. De uitvoering in LMS lijkt me echter te gedetailleerd. Omdat per niveau slechts één regel toegevoegd wordt, en de diagnose op niveau 1 begint, zal een leerling een lange reeks opgaven door moeten worstelen, waarbij van een niveau naar het volgende slechts een klein stapje genomen wordt. Dit lijkt niet alleen vervelend, maar het kan leerlingen ook te veel fixeren op een algoritme voor het oplossen van vergelijkingen, waarin elk stapje precies is vastgelegd.
Een betere benadering lijkt het apart testen van subprocedures als daar aanleiding toe bestaat. Zo zou een systeem apart kunnen testen of een leerling gelijksoortige termen op kan tellen, of termen naar de andere kant van de vergelijking kan brengen. Misschien is het ook mogelijk om het testen van subprocedures pas plaats te laten vinden als er in een probleem op hoger niveau fouten optreden. Deze taktiek wordt immers ook veel door leraren gebruikt!

2.4.5 Opmerkingen over didaktiek

Tot besluit van de case studies besteden we aandacht aan een blinde vlek in de bespreking tot dusver. Steeds is het onderwijs dat leerlingen genoten hebben vóór ze met BUGGY en LMS geconfronteerd werden buiten beschouwing gebleven. Het spreekt voor zich dat dit van grote invloed is op de bugs die leerlingen kunnen ontwikkelen.

In het wiskundeonderwijs kan onderscheid gemaakt worden tussen twee onderwijsopvattingen, namelijk *mechanistisch* en *realistisch* wiskundeonderwijs. Deze twee opvattingen kunnen als volgt omschreven worden.

Mechanistisch wiskundeonderwijs gaat uit van algoritmen. Daarbij wordt leerlingen een algoritme bijgebracht dat exact is vastgelegd in stappen en regels. Het aanleren gebeurt ook in stappen; steeds wordt door toevoegen van nieuwe regels een grotere klasse van problemen oplosbaar gemaakt. De aanpak in LMS weerspiegelt deze opvatting.

In *realistisch* wiskundeonderwijs wordt uitgegaan van een contekst uit de 'leefwereld van de leerling'. Wiskunde wordt opgehangen aan problemen uit de realiteit. Leerlingen worden gestimuleerd om zelf verkortingen te vinden in arbeidsintensieve procedures en pas later worden daaruit algoritmen geformaliseerd. In alle stadia blijft de binding met problemen uit de realiteit belangrijk.

Er zijn sterke aanwijzingen dat de realistische aanpak veel bugs voorkomt, door de binding met problemen uit de realiteit.[6] Bij de mechanistische aanpak ontstaan allerlei

varianten op regels juist omdat (varianten op) regels niet ingeperkt worden door zulke bindingen. Zo lijkt het zeer onwaarschijnlijk dat de leerling die de opgave 2003–24=1 maakte (zie de tabel in figuur 3.6), ook werkelijk gelooft dat als je van 2003 snoepjes er 24 opeet, er dan nog maar één overhebt! Toch produceert deze leerling het antwoord 1, hij heeft waarschijnlijk alleen maar regeltjes toegepast, de verbinding met realistische problemen is zoek.

Bij het ontwerp van een ITS voor algebra dient men *niet* uit te gaan van de mechanistische aanpak. Dit heeft anders iets van het paard achter de wagen spannen: er wordt een 'intelligent' systeem ontwikkeld om leerlingen te leren op een 'domme' manier een procedure af te werken.
De conteksten van de realistische aanpak zijn echter ook nog van ondergeschikt belang, omdat we ons richten op een diagnostisch systeem, waarin geen tutoring is opgenomen. Wel van belang daarbij is de representatie van de algebraische kennis: door af te stappen van de mechanistische aanpak moet deze kennis meer 'structuur' bevatten dan 'regeltjes', zoals in de beschouwing van LMS al uitgebreid is besproken.

2.5 Andere intelligent tutoring systems

Tot slot van deze beschrijving van ontwikkelingen op het gebied van intelligent tutoring systems geven we een lijst van een aantal aanverwante systemen op andere gebieden.

— WEST is een voorloper van intelligente onderwijssystemen. In feite is het de simulatie van een bordspel, *how the west was won*, waarbij het aantal zetten dat gedaan mag worden bepaald wordt door de uitkomst van een som, die de speler met drie gegeven getallen kan maken. Het systeem bevat een expert-speler, zodat de leerling zijn som kan vergelijken met die van het systeem.

— SOPHIE is een van de eerste intelligente tutors. Het domein is het opsporen van fouten in elektrische circuits. Sophie gaat uit van 'learning by doing'; de student dient fouten in een gegeven circuit op te sporen. Het systeem heeft een electronic troubleshooter en een didactisch module, dat vragen van de student over het te onderzoeken circuit van commentaar kan voorzien en hulp kan geven bij het zoeken naar de fout.

— MACSYMA ADVISOR is een tutor bij MACSYMA, een programma dat algebraische formules kan manipuleren. De gebruiker geeft daarin aan wat voor soort operatie plaats moet vinden (bijv. factoriseer naar x, bepaal coëfficiënt van x^2), het programma voert ze uit. Tutoring vindt ook plaats op het niveau van deze operaties, in termen van strategiën voor het vinden van oplossingen.

---

6. Onderzoek naar het leren van de staartdeling heeft opmerkelijke verschillen aangetoond in het voordeel van de realistische aanpak. Zie Rengerink [23].

— SPADE is een coach bij het schrijven van programma's in LOGO en assisteert bij het ontwerpen van modulaire turtle-graphics programma's. Daarbij gaat het vooral om de opeenvolging van subprocedures en de 'interfacing' tussen deze procedures.

— De THERMODYNAMICA COACH is door Bierman en Kamsteeg ontwikkeld voor het onderwijs van thermodynamica in het voortgezet onderwijs. Het richt zich vooral op conceptueel moeilijke zaken als het verschil tussen energie en warmte. In een voortzetting van dit project wordt gewerkt aan een simulatie van praktikumproeven. Met behulp van PCE-Prolog (een Prolog van waaruit een object-georiënteerd tekenprogramma bestuurd kan worden) wordt een bank uit een praktikumlokaal getekend waarop de leerling door middel van een muis allerlei 'handelingen' kan verrichten. De coach stelt het probleem en kan (tussentijdse) antwoorden en vragen van de leerling van commentaar voorzien.

2.6 Conclusies

Uit het overzicht van intelligente onderwijssystemen kunnen een aantal conclusies worden getrokken ten aanzien van de ontwikkeling van diagnostische systemen. Allereerst lijkt het dat het ontwikkelen van een ITS uitermate complex is. Het is een open vraag of het mogelijk is empty shells voor dergelijke systemen te ontwikkelen. Het lijkt me daarom verstandig de ontwikkeling te beperken tot een diagnostisch systeem voor fouten in algebra. Ook is een uitgebreid onderzoek naar voorkomende fouten in de algebra niet haalbaar. Wat dit betreft zal het een experimenteel systeem moeten worden, dat een aantal veel voorkomende fouten herkent. Dit is geen bezwaar, zolang de verzameling fouten makkelijk uit te breiden is. Vooralsnog kunnen LMS, de classificatie van Matz [21] en persoonlijke ervaringen als een voldoende rijke bron dienen.
Over het systeem is het volgende op te merken:

1. *De struktuur van het systeem voor diagnose van fouten in algebraische manipulaties.*
   Het systeem zal tenminste twee onderdelen moeten bevatten: een kunstmatige expert, die in staat is algebraische formules te manipuleren en een deel dat de leerling modelleert. De ervaring met LMS leert dat het belangrijk is dat de kunstmatige expert gestructureerd is op een manier die overeenstemt met de structuur in de algebra. De drie door Dede genoemde soorten van kennis zullen erin aanwezig moeten zijn.
   Mal-rules lijken een goede representatie van fouten die door leerlingen gemaakt worden. Omdat mal-rules variaties zijn op correcte regels, stelt het gebruik van deze representatie hoge eisen aan de kunstmatige expert. Niet alleen moeten de algebraische manipulaties gerepresenteerd kunnen worden, maar ook (foute) oplossingsstrategieën en fouten in begrippen, zoals het begrip 'variabele'.

6. Deze classificatie zal niet nader besproken worden.

2. *De geschiktheid van Prolog als taal om het systeem in te schrijven.*
In een eerste oogopslag lijkt Prolog uitermate geschikt voor de ontwikkeling van een intelligent onderwijssysteem. De goal-driven inferentie en Prolog-termen garanderen dat algebraische kennis gestructureerd kan worden gerepresenteerd. Meta-rules bieden mogelijkheden voor het representeren van strategiën. Mal-rules kunnen gerepresenteerd worden als gemerkte, alternatieve clauses voor een doel. Deze methode garandeert ook onafhankelijkheid van regels en daardoor uitbreidbaarheid.
De student-modeller kan gerepresenteerd worden als een meta-interpreter die bijhoudt welke 'mal-clauses' gebruikt worden om tot een bepaald antwoord te komen. In deze interpreter kunnen ook de heuristieken, die het zoekwerk moeten beperken, en technieken voor het behandelen van de ruis worden ingebouwd.
Mogelijk zijn de enorme aantallen gegevens, die bij het stellen van een diagnose moeten worden bijgehouden problematischer.

3. *De toepassingsmogelijkheden van het systeem.*
De experimenten met BUGGY leren dat er een breed scala van toepassingsmogelijkheden is voor (delen van) het systeem. Zo kan een met mal-clauses verrijkte expert nuttig gebruikt worden in de lerarenopleiding. Dergelijke toepassingen geven ook mogelijkheden voor het gescheiden testen van onderdelen.

4. *De volgorde waarin diverse problemen moeten worden aangepakt.*
Bij de ontwikkeling van het systeem lijkt het volgende traject voor de hand te liggen.

   a. Ontwikkeling van de kunstmatige algebra-expert. Dit moet resulteren in een programma dat algebraische manipulaties uit kan voeren.

   b. Verrijking van de manipulator met mal-clauses. Dit zou moeten leiden tot een systeem dat een leerling die consequente fouten maakt kan simuleren, of een naief diagnostisch systeem, zoals bij BUGGY.

   c. Ontwikkeling van een meta-interpreter, die diagnose kan stellen.

   Natuurlijk zijn de stappen nauwelijks los te zien. In elk deel zal al enigszins rekening moeten worden gehouden met de eisen die het volgende deel stelt. Je kunt immers geen mal-clauses voor niet bestaande clauses maken. De hier gegeven opbouw lijkt me echter het meest natuurlijk.
   De ontwikkeling van de onder c genoemde meta-interpreter, zonder dat er al ruime kennis is van de eisen die rules en mal-rules daaraan stellen, lijkt niet verstandig.

Hoofdstuk 3

Intelligente onderwijssystemen in Prolog

In dit hoofdstuk wordt de ontwikkeling van intelligente diagnostische systemen in Prolog besproken. Een dergelijk systeem omvat twee onderdelen: *domein-kennis* over het onderwerp dat onderwezen wordt, en een *shell* die deze kennis manipuleert. Wanneer er wordt gewerkt volgens het principe van de mal-rule, dient de domein-kennis drie soorten regels te bevatten: *correcte regels*, *mal-rules* en *heuristische regels*, die het zoekproces sturen. De shell bevat een *bug-generator* en een *bug-tester*[1].

De eerste paragraaf gaat in op de domein-kennis, en met name op de representatie van mal-rules in Prolog. De tweede paragraaf beschrijft een shell voor een diagnostisch systeem. Daarbij wordt vooral ingegaan op de bug-tester. Bug-generators en de bijbehorende heuristische regels zijn niet uitgebreid onderzocht; daarom wordt volstaan met het geven van een eenvoudige bug-generator.

Het aftrek-algoritme fungeert in dit hoofdstuk opnieuw als paradigma. De verzameling correcte regels is bij dit algoritme eenvoudig, terwijl het aantal mogelijke fouten al zeer groot is. Daarom leent het zich goed voor dit onderzoek, waarin mal-rules en een bug-tester centraal staan.

Bij dit hoofdstuk zal een groot aantal Prolog-programma's gegeven worden, in Edinburgh-Prolog. In de programmateksten worden clauses voor een bepaalde goal steeds voorafgegaan door een cursief gedrukt commentaar, waarin de declaratieve betekenis van de goal nader omschreven wordt.

3.1 Mal-rules in Prolog

In deze paragraaf wordt een Prolog-programma gegeven voor het aftrek-algoritme en vervolgens worden mal-rules gegeven door wijzigingen in dit algoritme aan te brengen. Het uitgangspunt is de beschrijving van BUGGY. Het algoritme is gebaseerd op het daar gegeven procedurele netwerk en de mal-rules zijn representaties in Prolog van de daar beschreven buggy procedures.

Omdat *representatie* in deze paragraaf voorop staat, worden de bugs steeds ingebed in

1. Overige functies van een shell, zoals het genereren van verklaringen en het geven van hulp bij de bouw van de knowledge base, worden in dit onderzoek buiten beschouwing gelaten.

een complete procedure voor het aftrekken. In de volgende paragraaf, over meta-interpreters, zal besproken worden hoe een bug genoteerd kan worden als afwijking van een correcte procedure (waarbij het deel dat overeenstemt met de correcte procedure dus kan worden weggelaten) en hoe samengestelde bugs uit losse clauses geconstrueerd kunnen worden. De gegeven buggy procedures werken steeds op een volledige aftreksom; ze geven dus alleen *full problem evidence.*
Om de verschillende procedures uit elkaar te houden, zijn ze alle voorzien van een nummer dat het soort fout aanduidt.

3.1.1 Het correcte algoritme

Figuur 3.1 bevat het correcte algoritme. Het is een directe afspiegeling van het procedurele netwerk dat in paragraaf 2.3 gegeven werd.

```
        minus(Type, Bs, Os, Us):
De aftrekking Bs-Os resulteert in Us volgens algoritme Type.
Getallen worden gerepresenteerd door lijsten en length(Bs)<=length(Us).

        leen(Type, Bs, NBs):
Lenen van Bs volgens algoritme Type resulteert in NBs.

        naam_type(Type, Naam):
Naam is een omschrijving van het algoritme met nummer Type.

naam_type(0, 'correct algoritme.').

minus(0, [B | Bs], [O | Os], [U | Us]):-
            kolom_kan(B, O),
            tafel(B-O, U),
            minus(0, Bs, Os, Us).

minus(0, [B | Bs], [O | Os], [U | Us]):-
            not kolom_kan(B, O),
            leen(0, Bs, NBs),
            tafel(10+B-O, U),
            minus(0, NBs, Os, Us).

minus(0, Bs, [], Bs).

leen(0, [B | Bs], [NB | Bs]):-
            not is_nul(B),
            tafel(B-1, NB).

leen(0, [0 | Bs], [9 | NBs]):-
            leen(0, Bs, NBs).

Clauses voor tafel, kolom_kan en is_nul zijn weggelaten.
```

```
  6  5
  7 4 4
    6 1 7
  -------
  6 8 4 7
```

figuur 3.1: het correcte aftrek-algoritme.

De getallen in de aftrekking worden gerepresenteerd als lijsten met cijfers. Omdat het algoritme van rechts naar links werkt, bevatten deze lijsten de cijfers in omgekeerde volgorde; het meest rechtse cijfer staat voor in de lijst. Het getal 2418 wordt dus gerepresenteerd als [8, 1, 4, 2].

Het predicaat *minus(Type, Boven, Onder, Uitkomst)* slaagt als *Boven–Onder* gelijk is aan *Uitkomst* volgens het algoritme *Type*. Er zijn drie clauses: De eerste clause beschrijft de aftrekking van een kolom waarin het bovenste cijfer groter of gelijk is aan het onderste, de tweede beschrijft de aftrekking in een kolom waar ingewisseld (geleend) moet worden. Beide eindigen met staartrecursie, om de rest van de aftreksom te maken. De laatste clause beschrijft de situatie waarin het onderste getal 'op' is en de aftrekking dus klaar is.

Het predicaat *leen(Type, Boven, NBoven)* slaagt als er volgens algoritme *Type* een 1 uit het getal *Boven* geleend kan worden, waarna er *NBoven* overblijft. Voor dit predicaat zijn er twee clauses, één voor de situatie waarin het getal *Boven* eindigt op een nul (de nul wordt een negen en er moet verderop geleend worden) en één voor de situatie waarin er direct geleend kan worden.

De predicaten *tafel*, *kolom_kan* en *is_nul* bevatten eenvoudige berekeningen en vergelijkingen die voor zich spreken en geen nadere uitwerking behoeven. Zo test *kolom_kan* of het cijfer boven groter of gelijk is aan het cijfer onder.

Naam_type bindt het typenummer aan een beschrijving.

Het programma kan zowel antwoorden controleren als genereren. Deze worden ook gegeven als lijsten. Het predicaat *minus* faalt als de aftrekking 'niet kan', dat wil zeggen als het onderste getal groter is dan het bovenste. Aftrekkingen met negatieve uitkomsten behoren (nog) niet tot de leerstof.
In figuur 3.1 zijn geen clauses opgenomen voor het 'opschrijven' van de aftreksom, in dit geval dus de conversie van getal naar lijst met cijfers. Deze zullen in paragraaf 3.1.4 besproken worden.

3.1.2 Enkele bugs

Buggy procedures kunnen worden gerepresenteerd door kleine wijzigingen in de predicaten aan te brengen. De programma's 3.2, 3.3 en 3.4 bevatten drie van deze buggy procedures. De afwijkingen van het correcte algoritme zijn voor alle duidelijkheid vet gedrukt.

De bugs worden gerepresenteerd als clauses voor de predicaten *minus* en *leen*. Wanneer deze als *extra* clauses aan figuur 3.1 worden toegevoegd, ontstaat een programma dat naast het correcte algoritme verschillende 'buggy' algoritmen bevat. Dit kan voor verschillende doeleinden gebruikt worden.

```
naam_type(1, 'grootste min kleinste.').

minus(1, [B | Bs], [O | Os], [U | Us]):-
        kolom_kan(B, O),
        tafel(B-O, U),
        minus(1, Bs, Os, Us).

minus(1, [B | Bs], [O | Os], [U | Us]):-
        not kolom_kan(B, O),
        tafel(O-B, U),
        minus(1, NBs, Os, Us).

minus(1, Bs, [], Bs).
```

predicaten voor leen zijn overbodig.

```
  7 4 6 4
    6 1 7
  -------  -
  7 2 5 3
```

figuur 3.2: grootste min kleinste.

```
naam_type(2, 'altijd lenen.').

minus(2, [B | Bs], [O | Os], [U | Us]):-
        kolom_kan(B, O),
        not Bs=[],
        leen(2, Bs, NBs),
        tafel(B-O, U),
        minus(2, NBs, Os, Us).

minus(2, [B | Bs], [O | Os], [U | Us]):-
        not kolom_kan(B, O),
        leen(2, Bs, NBs),
        tafel(10+B-O, U),
        minus(2, NBs, Os, Us).

minus(2, Bs, [], Bs).

leen(2, [B | Bs], [NB | Bs]):-
        not is_nul(B),
        tafel(B-1, NB).

leen(2, [0 | Bs], [9 | Bs]).
```

```
    6 3 5
  7 ~~4~~ 4
    6 1 7
  -------  -
  6 7 4 7
```

figuur 3.3: altijd 'lenen'.

Relaties

In een query dienen de getallen *Boven* en *Onder* altijd geinstantieerd te zijn (een predicaat als *tafel* rekent namelijk met de cijfers in beide getallen). *Type* en *Uitkomst* kunnen echter zowel input- als output-variabele zijn, waardoor het programma op verschillende manieren kan functioneren. We noemen twee soorten van queries expliciet.

1. *Uitkomst geinstantieerd, Type vrij.*
   Prolog zoekt het algoritme volgens welk een uitkomst verkregen is. Dit is dus het stellen van een *diagnose.*

```
naam_type(3, 'stoppen als onderste getal op is.').

minus(3, [B | Bs], [O | Os], [U | Us]):-
        kolom_kan(B, O),
        tafel(B-O, U),
        minus(3, Bs, Os, Us).

minus(3, [B | Bs], [O | Os], [U | Us]):-
        not kolom_kan(B, O),
        leen(3, Bs, NBs),
        tafel(10+B-O, U),
        minus(3, NBs, Os, Us).

minus(3, Bs, [], Zs):-
        niks_in_uitkomst(Bs, Zs).

leen(3, [B | Bs], [NB | Bs]):-
        not is_nul(B),
        tafel(B-1, NB).

leen(3, [0 | Bs], [9 | NBs]):-
        leen(3, Bs, NBs).
```

Het predicaat niks_in_uitkomst, *dat een lijst nullen genereert, is weggelaten.*

```
  6  5
 ~~4~~4
  6 1 7
 -------  -
  8 4 7
```

figuur 3.4: stoppen als onderste getal 'op' is.

Merk op dat het programma werkt volgens *full problem evidence.* De eerste applicatie van een clause van het predicaat *minus* bindt de variabele Type, zodat de rest van de som met de clauses voor dat type gemaakt moet worden!

2. *Uitkomst vrij, Type geinstantieerd.*
   Prolog genereert de uitkomst die het algoritme met het nummer Type zal geven. Daardoor kan het programma een leerling, die met een bepaalde bug werkt, simuleren. Het werkt dan als het spel BUGGY, dat in paragraaf 2.3.2 beschreven werd.

Een en ander is een direct gevolg van het feit dat Prolog met relaties werkt. In hoofdstuk 2 is aangetoond dat beide bovengenoemde toepassingen van nut kunnen zijn.

In tegenstelling tot bijvoorbeeld functionele talen, kan in Prolog één programma voor beide toepassingen gebruikt worden. Dit kan worden aangemerkt als een groot voordeel van Prolog boven andere programmeertalen met betrekking tot toepassingen zoals deze.

3.1.3 Patronen en samenstellingen

Veel fouten treden alleen op bij bepaalde patronen van cijfers. Vooral sommen met nullen zijn een bron van fouten. Figuur 3.5 bevat een bekende fout die betrekking heeft op 'lenen van een nul': De nul wordt bij het inwisselen overgeslagen, de leerling gaat verder bij het volgende cijfer.

Mal-rules voor bepaalde patronen vervangen een correcte regel vaak maar gedeeltelijk; ze zijn alleen toepasbaar in een deel van de situaties waarin de correcte regel werkt. Daarom moeten dergelijke mal-rules aan het algoritme worden *toegevoegd.* De correcte

```
naam_type(4, 'bij lenen nullen overslaan.').

minus(4, [B | Bs], [O | Os], [U | Us]):-
        kolom_kan(B, O),
        tafel(B-O, U),
        minus(4, Bs, Os, Us).

minus(4, [B | Bs], [O | Os], [U | Us]):-
        not kolom_kan(B, O),
        leen(4, Bs, NBs),
        tafel(10+B-O, U),
        minus(4, NBs, Os, Us).

minus(4, Bs, [], Bs).

leen(4, [B | Bs], [NB | Bs]):-
        not is_nul(B),
        tafel(B-1, NB).

leen(4, [0 | Bs], [0 | NBs]):-
        leen(4, Bs, NBs).
```

```
      6
  1 0 8 0 3
    6 5 2 9
  ---------
    4 1 8 4
```

figuur 3.5: nullen overslaan bij het lenen.

regel wordt dan niet verwijderd, maar voorzien van een extra conditie. Patronen kunnen ook aangeven dat een fout in uitzonderingssituaties juist *niet* gemaakt wordt.
In figuur 3.6 wordt van deze twee zaken een voorbeeld gegeven. Het 'nullen overslaan bij lenen' is gecombineerd met de fout '0–n=n', waardoor voorkomen wordt dat er twee keer van het cijfer vóór de nul geleend wordt. Bij getallen die beginnen met 10 treedt deze fout niet op, omdat dit gezien wordt als *tien*, zodat de aftrekking met behulp van de tafels gemaakt kan worden.

Determinisme

Samenstellingen en patronen roepen vragen op met betrekking tot het determinisme in rules en mal-rules. Wanneer in figuur 3.6 de extra voorwaarden in de correcte regels zouden zijn weggelaten, zou het niet meer deterministisch zijn: het programma had dan ook aftreksommen waarin de fout '0–n=n' *niet* gemaakt wordt, geaccepteerd, door toepassing van de vierde clause voor minus. Het programma zou kunnen slagen op elk van een reeks opgaven, zonder dat de genoemde bug één keer optreedt.

In het algemeen zal deze vorm van non-determinisme ongewenst zijn: Brown e.a. definiëren een bug immers als een fout die consequent wordt toegepast. Als de bug niet optreedt bij een bepaalde opgave, dient het programma te falen, ten teken dat die opgave de bug niet bevestigt.
Andere vormen van non-determinisme kunnen zeer gewenst zijn. Zo kan een leerling verschillende remedies hebben voor situaties, die hij niet beheerst (repair theory). In dat geval moet Prolog kunnen kiezen uit verschillende clauses.

```
naam_type(5, 'bij lenen nullen overslaan & 0-n=n & linker tien goed.').

minus(5, [0, 1], [O], [U, 0]):-
        tafel(10-O, U).

minus(5, [0 | Bs], [O | Os], [O | Us]):-
        not Bs=[1],
        minus(5, Bs, Os, Us).

minus(5, [B | Bs], [O | Os], [U | Us]):-
        not B=0,
        kolom_kan(B, O),
        tafel(B-O, U),
        minus(5, Bs, Os, Us).

minus(5, [B | Bs], [O | Os], [U | Us]):-
        not B=0,
        not kolom_kan(B, O),
        leen(5, Bs, NBs),
        tafel(10+B-O, U),
        minus(5, NBs, Os, Us).

minus(5, Bs, [], Bs).

leen(5, [B | Bs], [NB | Bs]):-
        not is_nul(B),
        tafel(B-1, NB).

leen(5, [0 | Bs], [0 | NBs]):-
        leen(5, Bs, NBs).
```

```
    7
1 0 ⁄ 0 3
  6 5 2 9
-------- -
  4 2 2 4
```

figuur 3.6: nullen overslaan bij lenen & 0–n=n & linker tien goed.

3.1.4 Het opschrijven van de opgave

Tot dusver is alleen de toepassing van het algoritme op reeds tot lijst herleide getallen ter sprake gekomen. In deze paragraaf worden predicaten besproken die het 'opschrijven' en 'voorlezen' van getallen – het herleiden tot een lijst en omgekeerd – uitvoeren en een predicaat dat de complete diagnose uitvoert. Figuur 3.7 bevat de noodzakelijke clauses.

Het predicaat *schrijf_op* beschrijft de omzetting van een getal in een lijst van cijfers, die bij het opschrijven van een opgave moet gebeuren; *lees_voor* beschrijft de omgekeerde bewerking, waarbij de resulterende lijst van cijfers weer als getal gelezen moet worden.
Deze predicaten kunnen gebruikt worden om schrijf- en leesfouten te representeren. Zo kan men nullen achten laten worden. Veel leerlingen hebben moeite met de richting van het opschrijven – vierendertig wordt opgeschreven als 43 –. Dit kan gerepresenteerd worden door (een deel van) de lijst om te keren. Het werken van links naar rechts kan gesimuleerd worden door beide lijsten om te keren. De mal-rules die dit weergeven zijn nogal recht-toe-recht-aan en daarom worden ze niet expliciet gegeven.

Diag is het predicaat dat bovenaan de hiërarchie staat. Ook dit kan voor verschillende doeleinden gebruikt worden. De query

```
        diag(Type, Naam, B-O=U):
De aftreksom B–O levert U op bij het hanteren
van algoritme Naam. Type is het bij Naam horende nummer.

diag(Type, Naam, B-O=U):-
        naam_type(Type, Naam),
        schrijf_op(B, Bs),
        schrijf_op(O, Os),
        minus(Type, Bs, Os, Us),
        lees_voor(Us, U).

        schrijf_op(Getal, Cijfers):
De representatie van Getal als lijst van cijfers is
Cijfers. Getal is input-variabele.

schrijf_op(0, []).

schrijf_op(G, [R | Gs]):-
        G > 0, !,
        R is G mod 10,
        D is G div 10,
        schrijf_op(D, LG).

        lees_voor(Cijfers, Getal):
Als schrijf_op, maar Cijfers is nu de input-variabele.

lees_voor([], 0).

lees_voor([C | Cijfers], G):-
        lees_voor(Cijfers, G1),
        G is 10*G1+C.
```

figuur 3.7: Notatie en uitvoering diagnose.

?- diag(1, Naam, 2304, 815, U).

simuleert de fout 'grootste min kleinste' en de query

?- diag(T, Naam, 2304, 815, 1419).

zoekt een bug die op 2304-815 het antwoord 1419 geeft.

Merk op dat het predicaat *naam_type* een tweede functie heeft, naast het voorzien van de bugtypes van een omschrijving. Bij de tweede query voorziet het in een universele kwantificatie van de bugtypes: bij backtracking worden alle bestaande bugtypes gegenereerd, die dan vervolgens uitgeprobeerd kunnen worden. Daardoor werkt het hier als een eenvoudige *bug-generator*.

Een laatste opmerking betreft de efficiëntie waarmee de diagnose gesteld wordt. *Diag* kiest een bugtype, genereert de uitkomst volgens de bijbehorende procedure en vergelijkt deze met het gegeven antwoord door middel van *lees_voor*. Eventuele failure treedt pas op bij de allerlaatste subgoal. Een efficiënter programma zou kunnen testen of *Uitkomst* geinstantieerd is en zo mogelijk vóór *minus* de omzetting naar lijst uitvoeren. Failure treedt dan op bij de eerste kolom die niet klopt.

Deze aanpassing moet wel met enige omzichtigheid doorgevoerd te worden, omdat er geen 1-1 afbeelding tussen getallen en lijsten bestaat. [3, 0] en [3] stellen beide het getal 3 voor. Er kan echter gebruik gemaakt worden van het feit dat *minus* uitkomsten genereert die evenveel cijfers bevatten als het getal boven: 1034–951 levert, in lijstnotatie, [3, 8, 0, 0] op. Dit is weliswaar equivalent met [3, 8], maar *minus* zal falen, als deze laatste lijst als argument wordt gegeven.

3.1.5 Rekenfouten

Fouten in elementaire berekeningen en vergelijkingen kunnen gerepresenteerd worden door foute varianten van de predicaten *tafel*, *kolom_kan* en dergelijke. De *coërcie* in DEBUGGY staat afwijkingen van ten hoogste 2 toe in de elementaire aftrekkingen. Deze worden gegeven door predicaten van de vorm:

```
tafel(Expr, Uitk):-
    Uitk is Expr+1.
```

In de paragraaf over shells wordt besproken hoe dergelijke verontreinigingen toegevoegd kunnen worden, zonder dat er weer een nieuwe verzameling volledige algoritmen gemaakt moet worden.

3.2 Een shell voor een systeem met mal-rules

In deze paragraaf zal aandacht besteed worden aan het belangrijkste onderdeel van de shell van een intelligent tutoring system. Het gaat om de shell die het zoeken door de rules en mal-rules stuurt. In het bijzondere gaat de aandacht uit naar de volgende zaken:

1. Mal-rules moeten gerepresenteerd worden als varianten van het correcte algoritme. In de voorgaande paragraaf werd steeds een compleet algoritme gegeven, dat de mal-rule bevatte. In plaats daarvan moeten we de mal-rule los kunnen geven. De shell moet dan correcte regels gebruiken in situaties waar de mal-rule niet toepasbaar is.
2. De shell moet uit de losse mal-rules samenstellingen van bugs kunnen genereren.
3. De shell moet met meer dan één som kunnen werken, dat is aanwijzingen voor een bug zoeken in een serie opgaven.

Deze zaken zullen in bovenstaande volgorde besproken worden.

3.2.1 Mal-rules als varianten van het correcte algoritme

In principe zouden we met losse mal-rules kunnen werken door de mal-rule uit het complete (foute) algoritme te lichten en de bug-tester te voorzien van deze mal-rule plus de correcte regels. De bug-tester kan dan waar mogelijk de mal-rule toepassen en in de andere gevallen correcte regels gebruiken.
Dit is echter niet zonder meer mogelijk. Allereerst veronderstellen we dat een leerling een fout consequent hanteert. Als een mal-rule van toepassing is, mogen er dus geen

correcte regels op dezelfde situatie worden toegepast. Daarnaast zijn er twee redenen waarom een mal-rule kan falen:

a. *De mal-rule is niet toepasbaar in de gegeven situatie.*
Bij de som 456–234 zal de mal-rule voor de bug *grootste min kleinste* steeds falen, omdat deze op geen enkele kolom van toepassing is.

b. *De mal-rule is toepasbaar in de gegeven situatie, maar faalt omdat het antwoord van de leerling er niet door voorspeld wordt.*
Dit kan alleen optreden bij diagnose, als de uitkomst van de opgave al geinstantieerd is. In dit geval faalt de mal-rule, omdat deze (blijkbaar) niet door de leerling gehanteerd wordt.

Een bug-tester zal deze gevallen moeten kunnen onderscheiden. Het eerste geval zegt immers niets over het al dan niet optreden van een bug en correcte regels mogen zonder meer gebruikt worden. Het tweede geval levert argumenten *tegen* het optreden van een bug. De bug-tester mag dan geen correcte regels toepassen en moet *failure* opleveren.

Een en ander is aanleiding om mal-rules te verdelen in een *conditie-deel* en een *actie-deel*. Het conditie-deel test of de mal-rule toegepast kan worden in de gegeven situatie en failure van dit deel correspondeert met het soort dat onder punt *a* genoemd wordt. Het actiedeel omschrijft de 'handelingen' van het algoritme, die uitgevoerd moeten worden, als de conditie slaagt. Failure van het actiedeel correspondeert met punt *b*.
Merk op dat *conditie* en *actie* hier gebruikt worden om een classificatie te geven van de unificaties en subgoals, die moeten slagen om een bepaalde regel te laten slagen. In geen geval mag er verwarring optreden met conditie en actie van een productieregel.

Condities en acties treden niet alleen op als subgoals in de body van een clause. Ze kunnen ook in de head staan in de vorm van bindingen van variabelen. Dit laat zich goed illustreren aan de hand van de volgende clauses van het correcte algoritme:

```
minus(0, [B | Bs], [O | Os], [U | Us]):-
    kolom_kan(B, O),
    tafel(B-O, U),
    minus(0, Bs, Os, Us).
```

In deze clause bestaat de conditie uit de test of de huidige goal *minus* is, of er nog cijfers boven en onder staan, en de subgoal *kolom_kan*. De actie is de aftrekking, de aanroep van *minus* en het samenstellen van de uitkomst [U | Us].

Conditie en actie zijn duidelijk aanwezig in de head van de volgende clause:

```
leen(0, [0 | Bs], [9 | NBs]):-
    leen(0, Bs, NBs).
```

De 0 in het tweede argument van de head is een conditie, die bepaalt dat deze clause toegepast moet worden als 'er van een nul geleend moet worden'. De 9 in het derde argument is een actie: het veranderen van de betreffende 0 in een 9.

Deze overwegingen leiden tot het volgende sjabloon voor de bug-tester:

diagnose(Type, Opgave...):-
conditie van mal-rule(Type, Opgave...),
actie van mal-rule(Type, Opgave...).

diagnose(Type, Opgave...):-
not *conditie van mal-rule(Type, Opgave...),*
probeer correcte regel(s).

Hierin duidt *Type* aan welke mal-rule er getest wordt. Deze mal-rule wordt toegepast wanneer mogelijk (clause 1). Correcte regels worden alleen toegepast als de mal-rule niet toegepast kan worden (clause 2).

Het probleem in dit sjabloon wordt gevormd door de test of er *niet* aan de condities van de mal-rule(s) voldaan is in de tweede clause. Feitelijk vinden we hier de aanpak van de eerste paragraaf terug, waar naast de mal-rules steeds correcte regels herhaald werden, voorzien van de juiste condities. Om deze negatie van condities weg te laten, kunnen we gebruik maken van een zogenaamde *red cut*. Red cuts brengen echter een aantal problemen met zich mee.

Problemen met 'cuts'

Bij de bespreking van de cut wordt gebruik gemaakt van een zeer eenvoudig predicaat, *minimum(X,Y,Z)*, dat slaagt als *Z* het minumum van *X* en *Y* is. De gegeven programma's zijn afkomstig uit Sterling [9].

minimum(X, Y, X):- X <= Y.
minimum(X, Y, Y):- X > Y.

Dit programma is deterministisch, het slagen van de ene clause sluit het slagen van de andere uit. Dit determinisme kan worden weergegeven door middel van een *green cut:*

minimum(X, Y, X):- X <= Y, !.
minimum(X, Y, Y):- X > Y, !.

Deze groene cuts veranderen niets aan de betekenis van het programma, ze geven alleen aan dat backtracking hier overbodig is. Eventueel kan de cut in de tweede clause worden weggelaten; na deze clause heeft Prolog immers geen keuzes meer.
Nu kan men beweren dat de voorwaarde in de tweede clause, *X>Y*, kan worden weggelaten. Immers als *X<=Y* in de eerste clause slaagt, wordt de cut bereikt en kan de tweede niet meer toegepast worden. Het programma wordt dan:

minimum(X, Y, X):- X <= Y, !.
minimum(X, Y, Y).

De cut in de eerste clause is nu rood geworden, omdat er expliciete voorwaarden zijn weggelaten. Het programma is echter niet correct: de query *minimum(1, 2, 2).* slaagt namelijk: *minimum(1, 2, 2).* unificeert niet met de head van de eerste clause en vervolgens slaagt de tweede clause! Deze versie van *minimum* is alleen correct wanneer het derde argument in de query een ongebonden variabele is. Het zijeffect van de cut is dus dat het predicaat niet meer een relatie is. Het kan alleen als functie gebruikt

worden (*bepaling* van het minimum van twee getallen).

Het bovenstaande programma kan ook bekeken worden in het licht van de verdeling in conditie en actie. De conditie in de eerste clause is de ongelijkheid, X<=Y, de actie is de unificatie van het eerste en het derde argument. Het programma is niet correct omdat de actie vóór de cut staat. Door de actie te verplaatsen wordt het programma weer correct:

```
minimum(X, Y, Z):- X <= Y, !, X = Z.
minimum(X, Y, Y).
```

Merk op dat elke query voor *minimum* nu unificeert met de head van de eerste clause. Daarom kan de tweede alleen bereikt worden als X<=Y faalt, dat wil zeggen als X>Y.

In de bug-tester hebben we behoefte aan een *red cut*, omdat we de condities voor toepassing van de correcte regels willen weglaten. Dit vereist een zorgvuldige notatie van de rules en mal-rules: acties zullen moeten worden uitgesteld tot na de volledige evaluatie van de conditie. In geen geval mogen ze in de head van een (mal-)rule voorkomen.

De regels

Rules en mal-rules zullen worden genoteerd door het predicaat *regel(Type, Head, Conditie, Actie)*. De shell zal voor evaluatie van berekeningen en vergelijkingen de standaard Prolog-interpreter aanroepen. Subgoals die op deze manier geëvalueerd moeten worden hebben het atom *built_in* als conditie:

```
regel( 0,
       tafel(Expr, Uitk),
       built_in,
       Uitk is Expr
     ).
```

Een volledig overzicht van de regels die voortvloeien uit de in de vorige paragraaf besproken bugs is opgenomen in figuur 3.8. Merk op dat de samengestelde bug (figuur 3.6) opgesplitst is in losse mal-rules, omdat de shell zelf samenstellingen zal genereren. Een aantal regels zijn geherformuleerd, om acties uit de head te verwijderen.[2]

2. Acties van de vorm [U | Us], die aangeven dat er tenminste één cijfer in de uitkomst moet verschijnen, zijn vanwege de leesbaarheid niet uit de heads van de regels gehaald. Ze leggen echter geen restrictie op, omdat de uitkomst door het in paragraaf 3.1.4 besproken probleem van de leidende nullen sowieso voldoende cijfers moet bevatten.

```
naam_type(0, 'correct algoritme.').

regel( 0,
       minus([B | Bs], [O | Os], [U | Us]),
       kolom_kan(B, O),
       ( tafel(B-O, U), minus(Bs, Os, Us) )
     ).
regel( 0,
       minus([B | Bs], [O | Os], [U | Us]),
       not kolom_kan(B, O),
       ( leen(Bs, NBs), tafel(10+B-O, U), minus(NBs, Os, Us) )
     ).
regel( 0,
       minus(Bs, [], Us),
       true,
       gelijk(Us, Bs)
     ).
regel( 0,
       leen([B | Bs], [NB | Bs]),
       not is_nul(B),
       tafel(B-1, NB)
     ).
regel( 0,
       leen([0 | Bs], [NB | NBs]),
       true,
       ( gelijk(NB, 9), leen(Bs, NBs) )
     ).
```

Een aantal mal-rules, die uit de complete programma's in paragraaf 4.1 gelicht zijn.

```
naam_type(1, 'grootste min kleinste.').
regel( 1,
       minus([B | Bs], [O | Os], [U | Us]),
       not kolom_kan(B, O),
       ( tafel(O-B, U), minus(Bs, Os, Us) )
     ).

naam_type(2, 'altijd lenen.').
regel( 2,
       minus([B | Bs], [O | Os], [U | Us]),
       ( kolom_kan(B, O), not gelijk(Bs, []) ),
       ( leen(Bs, NBs), tafel(B-O, U), minus(NBs, Os, Us) )
     ).
regel( 2,
       leen([0 | Bs], [NB | Bs]),
       true,
       gelijk(NB, 9)
     ).
```

figuur 3.8: rules en mal-rules voor het aftrek-algoritme.

```
naam_type(3, 'stoppen als onderste getal op is.').
regel( 3,
        minus(Bs, [], Us),
        true,
        niks_in_uitkomst(Bs, Us)
      ).

naam_type(4, 'bij lenen nullen overslaan.').
regel( 4,
        leen([0 | Bs], [NB | NBs]),
        true,
        ( gelijk(NB, 0), leen(Bs, NBs) )
      ).

naam_type(50, 'variant: linker tien goed.').
regel( 50,
        minus([0, 1], [O], [U, 0]),
        true,
        tafel(10-O, U)
      ).

naam_type(5, '0-n=n').
regel( 5,
        minus([0 | Bs], [O | Os], [U | Us]),
        true,
        ( gelijk(U, O), minus(Bs, Os, Us) )
      ).
```

figuur 3.8 (vervolg): rules en mal-rules voor het aftrek-algoritme.

De eerdergenoemde regel voor het 'lenen van een nul' wordt in deze notatie:

```
regel( 0,
       leen([0 | Bs], [NB | NBs]),
       true,
       ( gelijk(NB, 9), leen(Bs, NBs) )
     ).
```

Met uitzondering van de test op het cijfer nul, die in de head van de regel plaatsvindt, zijn er geen condities. Het vervangen van de nul door een negen vindt plaats bij de acties. In deze regel zijn er twee acties, wat is weergegeven door de conjunctie (A,B).

3.2.2 Het zoek-algoritme van de bug-tester

Doordat bugs als varianten op het correcte algoritme gerepresenteerd worden, kunnen samenstellingen van bugs eenvoudig worden weergegeven in een lijst. De samengestelde bug uit figuur 3.6 wordt dan gegeven door de lijst [50, 5, 4] (zie figuur 3.8 voor de bijbehorende mal-rules. De meest-specifieke regel staat daarbij voorop.

De bug-tester kan dan als volgt te werk gaan. Eén voor één worden de condities van de in de lijst genoemde mal-rules geëvalueerd. Als de eerste mal-rule in de lijst niet

```
        diagnose( Bugtypes, Goal ):
Goal kan worden waargemaakt door toepassing van regels van welke het type
in de lijst Bugtypes staat, aangevuld met correcte regels.
        diagnose_seq( Untried_Bugtypes, Goal, Bugtypes ):
Goal kan worden waargemaakt door directe toepassing van een regel waarvan
het type in de lijst Untried_Bugtypes of een correcte regel.
Bij het waarmaken van eventuele subgoals kan de volledige lijst Bugtypes
gebruikt worden.

diagnose( Types, true ):- !.

diagnose( Types, (A,B) ):-
        !,
        diagnose( Types, A),
        diagnose( Types, B).

diagnose( Types, not(Goal) ):-
        !,
        not diagnose( Types, Goal).

diagnose( Types, Goal ):-
        regel( 0, Goal, built_in, Actie),
        !,
        Actie.

diagnose( Types, Goal ):-
        diagnose_seq( Types, Goal, Types ).          % ( allowed goal full )

diagnose_seq( [ Type | MogelijkeTypes ], Goal, AlleTypes ):-
        regel( Type, Goal, Cond, Actie ),
        diagnose( AlleTypes, Cond ),
        !,
        diagnose( AlleTypes, Actie ).

diagnose_seq( [ Type | MogelijkeTypes ], Goal, AlleTypes ):-
        diagnose_seq( MogelijkeTypes, Goal, AlleTypes ).

diagnose_seq( [], Goal, AlleTypes ):-
        regel( 0, Goal, Cond, Actie ),
        diagnose( AlleTypes, Cond ),
        diagnose( AlleTypes, Actie ).
```

figuur 3.9: De kern van een meta-interpreter.

toepasbaar blijkt, wordt de tweede geprobeerd. Dit proces herhaalt zich tot er geen mal-rules meer zijn en pas dan worden correcte regels gebruikt. Merk op dat de volgorde zeer belangrijk is: correcte regels mogen pas toegepast worden als de voorwaarden van alle mal-rules falen.

Figuur 3.9 bevat een bug-tester die werkt volgens het beschreven schema. Het predicaat *diagnose* – de naam geeft hier niet een uniek gebruik van het predicaat aan; het zou ook *simulatie* kunnen zijn – heeft vijf clauses, voor *true*, conjunctie, negatie, *built_in*-functies en applicatie van regels. De cuts in deze clauses zijn groen, ze geven determinisme weer.

Voor applicatie van regels maakt *diagnose* gebruik van een hulp-predicaat,

diagnose_seq. Dit beschrijft het opeenvolgend proberen van de mal-rules, die in de lijst van types zijn opgenomen. *Diagnose_seq* heeft een extra argument: naast de lijst van bugtypes die 'nog uitgeprobeerd' moeten worden, wordt de volledige lijst meegegeven ten behoeve van de evaluatie van subgoals.
De eerste clause probeert een mal-rule, de eerste in de lijst van bugtypes, toe te passen en bevat de *red cut*. Als de conditie van deze mal-rule faalt, kan een eventuele tweede bug geprobeerd worden, of correcte regels, als de resterende lijst van mal-rules leeg is. Dit is weergegeven in de tweede en derde clause voor *diagnose_seq*. Hier is de voorwaarde dat alle voorgaande bugs niet toepasbaar zijn weggelaten.

Diagnose op losse opgaven

Allereerst kan de bug-tester verwerkt worden in het predicaat *diag*, waarmee de mal-rules op losse opgaven geprobeerd kunnen worden. Dit lijkt sterk op het gelijknamige predicaat uit paragraaf 3.1; de voornaamste verandering is dat samengestelde bugs nu weergegeven worden door een lijst van primitieve bugs. Het is gegeven in figuur 3.10.

```
        diag(Types, Namen, A-B=C):
De som A-B heeft als uitkomst C, wanneer regels uit de lijst Types
gebruikt worden. Namen is de lijst van omschrijvingen van de in Types
voorkomende mal-rules.

diag(Types, Namen, A-B=C):-
        naam_types(Types, Namen),
        schrijf_op(A, As),
        schrijf_op(B, Bs),
        diagnose(Types, minus(As, Bs, Cs)),
        lees_voor(Cs, C).
```

figuur 3.10: eenvoudige diagnose bij een enkele opgave.

Het predicaat *naam_types* fungeert wederom als bug-generator. Wanneer het aangeroepen wordt met twee variabelen, bij het stellen van een diagnose, moet het door middel van backtracking alle mogelijke (samenstellingen van) bugs kunnen genereren. *Naam_types* wordt gegeven in figuur 3.11. De volgorde van clauses en subgoals bepaalt de volgorde van de gegenereerde bugs. Deze is zo gekozen, dat korte lijsten gegenereerd worden vóór langere. Als eerste komt de lege lijst, wat staat voor 'geen bugs'; het correcte algoritme. Vervolgens komen alle lijsten met één mal-rule en daarna pas samenstellingen.

Aan de lijsten kunnen restricties opgelegd worden. De restrictie dat de toe te voegen bug *T* niet al in de lijst *Types* zit, is noodzakelijk om terminatie te garanderen. Daarnaast kan er een restrictie worden opgelegd aan de lengte van de lijst [T | Types]. Zo hanteert DEBUGGY een grens van vier bugs in een samenstellingen.
Andere restricties kunnen worden gegeven door heuristische regels die afhankelijk zijn van het domein. Er kan een database van relaties tussen bugs worden aangelegd. Een aantal van deze relaties zijn beschreven in hoofdstuk 2. Hier geven we een paar voorbeelden van representaties, waarbij voor de leesbaarheid de namen van de bugs gebruikt worden.

```
        naam_types( Types, Namen ):
Types is een lijst van types van mal-rules met omschrijvingen Namen,
die een zinnige samenstelling vormen.

naam_types([], []).

naam_types([T | Types], [N | Namen]):-
        naam_types(Types, Namen),
        naam_type(T,N),
        not ongewenst(T, Types).

ongewenst(T, Types):-
        member(T, Types).

ongewenst(T, Types):-
        member(OudT, Types),
        sluiten_elkaar_uit(T, OudT).

Er zijn nog veel meer mogelijkheden voor ongewenst.
```

figuur 3.11: het genereren en benoemen van bugs.

sluiten_elkaar_uit('grootste min kleinste', 'altijd lenen').
is_speciaal_geval_van('0-n=n', 'grootste min kleinste').
is_variant_van('linker tien goed', 'lenen van nul zonder verder te lenen').

In de eerste twee gevallen is het niet zinvol de samenstellingen te onderzoeken; *naam_types* kan lijsten die beide bugs bevatten dus weglaten. In het laatste geval wordt aangegeven dat het alleen zinnig is om een variant te onderzoeken als de bijbehorende bug al in de lijst staat. De variant moet ook vóór de bug staan, wat een restrictie legt op de volgorde in de lijst.
In een uitgebreidere bug-generator zouden ook regels opgenomen kunnen worden die de *volgorde* waarin (samenstellingen van) mal-rules gegenereerd worden bepalen.

Het behoeft geen toelichting dat het predicaat *naam_types* verantwoordelijk zal zijn voor de combinatorische explosie in het aantal te onderzoeken fouten.

3.2.3 Diagnose op een serie opgaven

Het volgende punt is diagnose op een serie opgaven. We veronderstellen dat van een leerling een serie opgaven gebruikt kan worden. Deze kunnen in de database van Prolog worden opgenomen als feiten van de vorm:

som(Leerling, Somnummer, A-B=C).

DEBUGGY begint met het zoeken naar enkelvoudige bugs die tenminste één *foute* som volledig voorspellen, *single problem evidence*. Het Prolog-programma dat dit uitvoert, wordt gegeven in figuur 3.12.

```
        som( Leerling, Somnr, A-B=C ):
C is het antwoord dat Leerling op de som A-B gaf.
Somnr is het nummer van de som.
        spe( Leerling, Buglist ):
Buglist is de lijst van alle mogelijke termen som( Somnr, Type, Naam )
welke aangeven dat het foute antwoord van Leerling op som Somnr
voorspeld word door toepassing van de mal-rule(s) (Type, Naam).

find_spe(Leerlnr, Somnr, Type, Naam):-
            som(Leerlnr, Somnr, A-B=C),
            schrijf_op(A, As),
            schrijf_op(B, Bs),
            length(As, N),
            schrijf_op_vl(N, C, Cs),
            not diagnose([], minus(As, Bs, Cs)),
            naam_type(Type, Naam),
            diagnose([Type], minus(As, Bs, Cs)).

spe(Leerlnr, Buglist):-
            bagof( som_bug(Somnr, Type, Naam), find_spe(Leerlnr, Somnr, Type, Naam), Buglist).
```

figuur 3.12: single problem evidence.

Het predicaat *spe*, genereert een lijst van termen *som_bug(Somnr, Type, Naam)*, die aangeven dat het antwoord op de som Somnr door de bug (Type, Naam) gesimuleerd wordt.
Find_spe kiest een foute som en een enkelvoudige bug en slaagt als de bug het antwoord voorspelt. Het predicaat *spe* verzamelt alle geslaagde instanties van *find_spe* in een 'bag'.

De resulterende lijst van bugs kan gebruikt worden om samenstellingen te onderzoeken. De shell kiest dan een aantal bugs uit de Buglijst en kijkt of deze voldoen aan de eerdergenoemde restricties. Vervolgens kunnen deze bugs weer onderzocht worden met behulp van het predicaat *diagnose*.

De uiteindelijke beoordeling van een mogelijke bug vergt meer dan alleen het onderzoeken of een bug op een bepaalde som *slaagt*, dat wil zeggen het juiste antwoord voorspelt. Zo telt het als een sterk argument tegen een bug als deze een fout antwoord voorspelt, terwijl de leerling de som goed gemaakt heeft. Uiteindelijk is een overzicht zoals in de *bug comparison tables* uit hoofdstuk 2 nodig.

Het is duidelijk dat het hierbij om aanzienlijke hoeveelheden gegevens gaat en het lijkt het eenvoudigst om hiervan, met behulp van *assert* een database aan te leggen. Deze moet dan feiten van de volgende vorm bevatten:

uitkomst(Leerling, Somnr, BugTypes, C).

waarin *BugTypes* weer de lijst is die een (samengestelde) bug en eventueel toegepaste coërcie representeert.
Het definitieve opmaken van de score kan daarna gebeuren door predicaten die weinig meer uitvoeren dan een database-query.

Mogelijke uitbreidingen

De gegeven clauses voor *diagnose* kunnen op verschillende punten uitgebreid worden om een betere diagnose te verkrijgen. We bespreken hier de coërcie (rekening houden met rekenfouten e.d.) en non-determinisme in het algoritme.

coërcie
Rekenfouten, schrijffouten en dergelijke kunnen worden gerepresenteerd door extra regels op te nemen voor de built_in predicaten als *tafel* en *kolom_kan*. Deze regels moeten dan gemarkeerd worden met (een code voor) coërcie, die wanneer dat gewenst is in de lijst van Types kan worden opgenomen. De clause van *diagnose* die de built_in predicaten behandelt, moet dan echter herschreven worden:

```
diagnose( Types, Goal ):-
    member( T, Types ),
    regel( T, Goal, built_in, Actie ),
    Actie.
```

De toepassing van regels voor coërcie verschilt essentiëel van de toepassing van andere mal-rules. Er is hier geen sprake van determinisme, in die zin dat de correcte aftrekking niet meer toegepast mag worden als een rekenfout gemaakt kan worden. Daarom dient de cut verwijderd te worden.

Enige variatie in de notatie is mogelijk. Men kan de built_in regels scheiden van de andere regels door ze op te nemen in een speciaal predicaat *built_in_regel(Type, Goal, Actie)*. Ook kan men stellen dat het toepassen van coërcie iets anders is dan een bug en daarom als een apart argument aan *diagnose* moet worden meegegeven, in plaats van het op te nemen in de lijst van bugtypes. Beide zaken kunnen door triviale herformuleringen gerealiseerd worden.

non-determinisme
Het aftrek-algoritme wordt weliswaar als voorbeeld gebruikt in deze bespreking van meta-interpreters, maar het kan niet model staan voor andere toepassingen. Het algoritme is namelijk deterministisch. In het algemeen, en in het bijzonder bij de algebra, zal dit niet het geval zijn. In de algebra kunnen er immers vele verschillende manieren zijn om een probleem aan te pakken, geassociëerd met even zovele toepasbare regels.
Een direct gevolg van het determinisme in het aftrek-algoritme is dat de noodzaak ontbreekt om bij een mal-rule aan te geven van welke correcte regel het een 'buggy' variant is. In iedere situatie is immers maar één correcte regel toepasbaar en een toepasbare mal-rule is dus impliciet een variant van die regel.

Veronderstel nu dat in een bepaalde situatie twee regels, A en B, van toepassing zijn en verder dat er getest moet worden of een leerling een buggy variant van B, M', hanteert. Veronderstel tenslotte dat M' in deze situatie ook toepasbaar is. De huidige versie van *diagnose* zal regel A niet kunnen gebruiken, omdat er een toepasbare mal-rule is, terwijl toepassing van A het gebruik van B (of M') mogelijk overbodig maakt.
In de genoemde situatie zal *diagnose* regel A vrij moeten kunnen gebruiken, terwijl voor B de oude restrictie blijft gelden dat de conditie van M' moet falen. Dit

onderscheid vereist de associatie van mal-rules met correcte regels en kan in Prolog gerealiseerd worden door regels een naam te geven en de bijbehorende mal-rules van dezelfde naam te voorzien.
In de clauses voor *diagnose* kan met een eenvoudige aanpassing volstaan worden: Voor de aanroep van *diagnose_seq* moet eerst een regel gekozen worden. *Diagnose_seq* zal dan mal-rules voor die regel proberen vóór de correcte regel wordt toegepast. Als *diagnose_seq* faalt voor een bepaalde regel, kan door middel van backtracking een andere regel geprobeerd worden.

3.2.4 Slotopmerkingen: algemeen toepasbare shells

Geconcludeerd kan worden dat het mogelijk lijkt om een shell te ontwikkelen die in verschillende intelligente onderwijssystemen toegepast kan worden. De rol van zo'n shell is echter beperkt. Een groot deel van het systeem zal gebonden zijn aan het domein. Niet alleen correcte regels en mal-rules zijn domein-afhankelijk, maar ook heuristische regels voor de bug-generator en regels die de onzekerheid weergeven (coërcie) zijn domein-afhankelijk.

De in dit hoofdstuk gegeven shell is zeer eenvoudig en vatbaar voor een groot aantal uitbreidingen en verfijningen. Zo is er nog geen aandacht besteed aan het apart testen van delen van het algoritme, zoals in LMS gedaan wordt. Ook is er nog nauwelijks aandacht besteed aan het tegen elkaar afwegen van verschillende hypotheses en met name het genereren van onderscheidende opgaven.

Daarnaast zal een reeks problemen van meer technische aard nog afdoende behandeld moeten worden. Het voornaamste probleem lijkt de beheersing van de combinatorische explosie in de bug-generator.

Het was echter een verrassing dat een bug-tester door een redelijk eenvoudig programma als het in paragraaf 3.2.2 gegeven predicaat *diagnose* gegeven kon worden.

Literatuur

[1] P. H. Winston,

[2] E. Rich, Artificial Intelligence,

[3] P. Jackson, Introduction to expert systems, Addison Wesley, 1985.

[4] B. Buchanan, E. Shortliffe (ed.), Rule Based Expert Systems - The MYCIN Experiments of the Stanford Heuristic Programming Project, Addison Wesley, 1984.

[5] P. J. F. Lucas en L. C. van der Gaag, Principes van Expertsystemen, Academic Service, 1988.

[6] Forsyth (ed.), Expert Systems - Principles and Case Studies, Chapman and Hall, 1984.

[7] L. Brownston, R. Farrell, E. Kant en N. Martin, Programming Expert Systems in OPS5. Addison-Wesley, 1985.

[8] I. Bratko, Prolog programming for artificial intelligence, Addison Wesley, 1986.

[9] L. Sterling en E. Shapiro, The art of prolog, The MIT Press, 1986.

[10] A. Walker (ed.) Knowledge systems and Prolog, a logical approach to expert systems and natural language processing, Addison Wesley, 1987.

[11] C. Dede, A review and synthesis of recent research in intelligent computer-assisted instruction, Int. Jnl. of Man-Machine Studies 24, 1986, pp. 329-353.

[12] D. Sleeman en J. S. Brown (ed.), Intelligent Tutoring Systems, Academic Press, 1982.

[13] R. R. Burton, Diagnosing bugs in a simple procedural skill, in: D. Sleeman en J. S. Brown (ed.), Intelligent Tutoring Systems, Academic Press, 1982, pp. 157-183.

[14] D. Sleeman, Assessing aspects of competence in basic algebra, in: D. Sleeman en J. S. Brown (ed.), Intelligent Tutoring Systems, Academic Press, 1982, pp. 185-199.

[15] J. S. Brown en R. R. Burton, Diagnostic models for procedural bugs in basic mathematical skills, in Cognitive Science 2, 1978, pp. 155-192.

[16] R. M. Young en T. O'Shea, Errors in Children's subtraction, in Cognitive Science 5, 1981, pp. 153-177.

[17] J. S. Brown en K VanLehn, Repair theory: A generative theory of bugs in procedural skills, in Cognitive Science 4, 1980, pp. 379-427.

[18] D. Sleeman, An attempt to understand students' understanding of basic algebra, in Cognitive Science 8, 1984, pp. 387-413.

[19] D. Sleeman en M. J. Smith, Modelling students' problem solving, in Jnl. of Artificial Intelligence 16, 1981, pp. 171-187.

[20] W. J. Clancey, Tutoring rules for guiding a case method dialog, in: D. Sleeman en J. S. Brown (ed.), Intelligent Tutoring Systems, Academic Press, 1982, pp. 201-225.

[21] M. Matz, Towards a process model for high school algebra errors, in D. Sleeman en J. S. Brown (ed.), Intelligent Tutoring Systems, Academic Press, 1982, pp. 25-50.

[22] A. Bundy en B. Welham, Using meta-level inference for selective application of multiple rewrite rule sets in algebraic manipulation, in Jnl. of Artificial Intelligence 16, 1981, pp. 189-212.

[23] J. Rengerink, De staartdeling, een geintegreerde aanpak volgens het principe van progressieve schematisering, onderzoekspublicatie van de vakgroep onderwijskunde en OW&OC, 1983.