Grammaticacontrole met behulp van Rscript

ing. B.J. Arnoldus
14 augustus 2005

Master Thesis in Software Engineering

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Universiteit van Amsterdam, Hogeschool van Amsterdam en Vrije Universiteit Amsterdam

Afstudeerdocent: prof. dr. P. Klint
Begeleider: dr. M.G.J. van den Brand
Opdrachtgever: Centrum voor Wiskunde en Informatica, Amsterdam

Inhoudsopgave

1 Samenvatting

De ASF+SDF Checker zoals deze aanwezig is binnen de MetaEnvironment dient vervangen te worden. Een nieuw ontwikkelde technologie genaamd Rscript lijkt hiervoor zeer geschikt te zijn. Aangezien dit de eerste grote toepassing van Rscript is komen er een aantal vragen naar boven.

- Wat is het verschil in prestatie en welke onderdelen zijn daarvoor verantwoordelijk?
- De broncode dient simpel en begrijpbaar te zijn in verband met toekomstig onderhoud. Is dit haalbaar met een zeer krachtige taal zoals Rscript?
- Welke controles zijn beter te beschrijven in Rscript en welke in een herschrijf-formalisme zoals ASF+SDF?

In deze scriptie wordt uiteengezet hoe de nieuwe ASF+SDF Checker is opgebouwd en welke technieken zijn toegepast. Het feitenextractie proces is verder geoptimaliseerd. De losse feiten kunnen in de toekomst gebruikt worden voor andere analyses of het visualiseren van deze gegevens.
Naast de ASF+SDF Checker zelf is er een gehele omgeving gebouwd waarin zich ook een testomgeving bevindt. Een programma om de uitvoer van de nieuwe checker te herschrijven naar het error-summary formaat zijn ook aanwezig.
De uiteindelijke code omvang van de nieuwe checker is kleiner dan de oude. Hierbij is opvallend dat het grootste aandeel van de code nog te vinden is in de feitenextractie.
Rscript is geschikt voor het checken van een SDF definitie. De meeste checks zijn beschreven in Rscript en maar een enkele zijn gespecificeerd in ASF+SDF.
De prestaties van de nieuwe ASF+SDF Checker vallen op dit moment nog zwaar tegen. De Rscript gebaseerde oplossing is hierdoor nog niet geschikt voor industriële toepassing. Een betere implementatie van RscriptCalculator zou dit probleem kunnen verhelpen.

2 Inleiding

2.1 Introductie

Aan het Centrum voor Wiskunde en Informatie (CWI) te Amsterdam wordt er al lange tijd gewerkt aan ASF+SDF. ASF+SDF is een combinatie van het algebraïsche specificatie formalisme ASF en het syntax definitie formalisme SDF [HHKR89]. Deze formalismen maken het mogelijk om de syntax, en semantiek van een (programmeer) taal te beschrijven en om analyses en transformaties te beschrijven. Rondom deze formalismen is een gehele gereedschapsset gegroeid welke gebundeld is in de ASF+SDF MetaEnvironment [met].
De ASF+SDF Checker is een onderdeel van de MetaEnvironment. Dit programma controleert de SDF definitie voordat deze gebruikt wordt voor het genereren van de parseertabel. De ASF+SDF Checker is bedoeld om de ontwikkelaar te helpen zijn specificatie beter te formuleren. De implementatie van deze checker is volledig beschreven in de ASF+SDF formalismen.
De huidige checker voldoet echter niet meer geheel aan de eisen die er in de toekomst aan gesteld kunnen worden. Het toevoegen van complexere controles vormt een probleem met de huidige opzet. En de complexiteit van de huidige ASF+SDF Checker is groot, waardoor de onderhoudbaarheid wordt belemmerd.
Een vervanging door een ander concept is op zijn plaats en hiervoor is gedacht aan een op Rscript [Kli03] [Kli05] gebaseerde oplossing. Rscript is een nieuw ontwikkelde scripting taal gebaseerd op relatie calculus. De taal is bedoeld voor het analyseren van de broncode van software systemen. Met deze technologie zijn op dit moment nog weinig grote industriële projecten uitgevoerd. Een van de weinige industriële toepassingen is een deelproject binnen het CALCE [cal] project van de VU.
De toepassing van Rscript zou moeten zorgen voor een overzichtelijkere oplossing en zou beter geschikt moeten zijn voor verdere evolutie van de checker. De toepassing van Rscript zou het ook mogelijk moeten maken om gemakkelijk controles te implementeren die vroeger bijna onmogelijk waren.
Deze checker, zoals deze aanwezig is in de MetaEnvironment, bestaat uit twee lagen. De eerste laag voert SDF[1] specifieke controles uit en de tweede laag de controles voor SDF die gebruikt wordt voor een ASF specificatie. Er worden zowel waarschuwingen (warnings) als fouten (errors) gegenereerd.
De vraag achter dit onderzoek is of het mogelijk is om deze checks uit te voeren op basis van Rscript.

2.2 Onderzoeksvraag en de doelstellingen

Het doel achter dit project is om te onderzoeken of Rscript geschikt is voor industriële toepassingen en of het mogelijk is om de huidige ASF+SDF Checker te vervangen door een checker die is gebaseerd op Rscript. Aan de hand van het bouwen van een werkend prototype kan worden aangetoond of Rscript geschikt is voor de taak en wat de problemen zijn.
Vanuit dit doel vallen verschillende vragen te formuleren. De hoofdvraag die geformuleerd is uit het doel, is de vraag of het mogelijk is om een adequate ASF+SDF Checker te formuleren op basis van een feitenextractie in samenwerking met Rscript. Adequaat geeft aan dat de nieuwe checker op zijn

[1] Eigenlijk wordt hier SDF2 bedoeld. Dit is de opvolger van SDF.

minst hetzelfde moet zijn op het gebied van prestatie en niet mag groeien in omvang.
Vanuit dit perspectief ontstaan een aantal deelvragen over de volgende kwaliteitsaspecten in vergelijk met de oude ASF+SDF Checker:

- Wat is het verschil in prestatie en welke onderdelen zijn daarvoor verantwoordelijk?
- De broncode dient simpel en begrijpbaar te zijn in verband met toekomstig onderhoud. Is dit haalbaar met een zeer krachtige taal zoals Rscript?

Een ander onderdeel is de vraag of Rscript wel geschikt is voor alle soorten controles.

- Welke controles zijn beter te beschrijven in Rscript en welke in een herschrijf-formalisme zoals ASF+SDF?

2.3 Type of well-formedness checker

SDF is een taal waarin grammatica's beschreven kunnen worden. SDF is gebaseerd op contextvrije grammatica's uitgebreid met character classes, sorts, literals, reguliere expressies, renamings, aliasen en modules. SDF combineert ook de definitie van lexicale syntax en contextvrije syntax in één formalisme. SDF is geschikt om zowel grammatica's te definiëren als de definities voor het gebruik met ASF [Vis97a].
Om de kwaliteit van specificaties te waarborgen en bugs te voorkomen controleert de ASF+SDF Checker of een SDF specificatie goed geformuleerd (well-formed) is. Dit houdt in dat een SDF definitie gespecificeerd moet zijn volgens de regels in [vdBK04] en [Vis97a]. De controles zijn vooral gebaseerd op het gebruik van dubbele declaraties, ongebruikte soorten, illegale attributen en illegale traversal attributen. De checks die uitgevoerd worden verschillen wezenlijk van een typechecker aangezien er niet wordt gecontroleerd of een operator is toegepast op een incompatibele operand [ASU88]. Er kan niet echt gesproken worden over een echte typechecker, maar van een well-formedness check gecombineerd met uniqueness checks [ASU88].

2.4 Relationele algebra als analyse gereedschap

De meeste controles die uitgevoerd moeten worden om een SDF definitie te checken zijn relationeel van aard. Dit maakt het mogelijk om de checks te realiseren in een relatie geörienteerde taal zoals Rscript. Het gebruik van Rscript voor het analyseren van software heeft een bijna standaardaanpak. Eerst dienen de feiten geëxtraheerd te worden en daarna moet van deze feiten een Rstore opgebouwd worden. Een Rstore is een taalonafhankelijk uitwisselingsformaat om complexe relationele data tussen programma's uit te wisselen [Kli05].
De checks staan beschreven in een Rscript. De comprehensies maken het mogelijk complexe verbanden binnen feiten op korte wijze op te schrijven.

2.5 Rscript

Rscript biedt de mogelijkheid om bewerkingen uit te voeren op sets en relaties. Een set bestaat uit een unieke lijst met elementen. Een set heeft

de volgende vorm: $set[T]$ $\{E_1, E_2, \ldots, E_n\}$ waarvan de elementen van het type T zijn. Een relatie is een set waarvan de elementen uit tupels bestaan. Een relatie heeft de volgende vorm: $rel[T_1, T_2]$ $\{< E_{11}, E_{12} >, < E_{21}, E_{22} > , \ldots, < E_{n1}, E_{n2} >\}$.
Op deze sets en relaties kan de relatie algebra worden toegepast. Voorbeelden hiervan zijn de ingebouwde operatoren vereniging, verschil, carthesisch produkt of intersectie. De intersectie $\{1, 2, 3\} \cap \{2, 3, 4\} = \{2, 3\}$ ziet er in Rscript als volgt uit: $\{1, 2, 3\}$ $inter$ $\{2, 3, 4\}$ reduceert tot $\{2, 3\}$.
Naast deze ingebouwde operatoren biedt Rscript de mogelijkheid om gebruik te maken van comprehensies. Een comprehensie wordt als volgt genoteerd: $\{E_1, \ldots, E_m | G_1, \ldots, G_n\}$. Het resultaat van de comprehensie bestaat uit een vereniging van de opeenvolgende elementen $E_1, \ldots, E_n$. E wordt bepaald aan de hand van alle mogelijke combinaties die door de generatoren $G_1, \ldots, G_n$ geproduceerd kunnen worden. De generators kunnen alle elementen van set of relatie opsommen, een test uitvoeren of een waarde aan een variabele toekennen. Een voorbeeld van een comprehensie: $\{X | int X : \{1, 2, 3, 4, 5\}, X >= 3\}$ geeft als resultaat $\{3, 4, 5\}$.
Deze constructies maken het mogelijk om kort en bondige complexe software analyses te beschrijven. De gehele Rscript omgeving is verder beschreven in [Kli05].

3 Gerelateerd werk

SDF is enkel het formalisme om een grammatica te beschrijven en wordt pas zinvol wanneer deze gebruikt wordt samen met een parser. SGLR [Vis97b] is de parser die aanwezig is in de MetaEnvironment en deze maakt gebruik van SDF. Naast SGLR bestaan er nog vele andere parsers en parsergeneratoren. Enkele voorbeelden van andere formalismen die gebruikt worden door andere parsers zijn (E)BNF [ebn] [ea60] of dialect daarvan, GDML [Cle03], van Wijngaarden formalisme [GJ98].
De meeste parsers en parsergeneratoren zijn gebaseerd op één van deze formalismen voor het specificeren van grammatica's. De opgestelde grammatica zal gecontroleerd moeten worden op de formulering ervan, zodat er geen onzin wordt gegeneerd.
De meeste parsergeneratoren zoals YACC en ANTLR [ea03] controleren de specificatie gedurende het proces van het opbouwen van de parsercode of de ontleedtabel. Deze aanpak zorgt voor zeer snelle code, maar is zeer ondoorzichtig verweven met de parser code. Deze manier van werken komt de onderhoudbaarheid niet ten goede. Ten tweede zijn complexere controles moeilijk te implementeren en blijft de foutterugkoppeling hierdoor zeer oppervlakkig.
ANTLR gaat echter verder dan YACC met zijn checks. Er wordt een LL(k) analyzer aangeroepen. Dit programma controleert of de grammatica voldoet aan het LL(k) principe.
In [HK95] wordt een well-formedness checker gepresenteerd voor μCRL[2]. Deze checker is net als de oude ASF+SDF Checker gebaseerd op ASF+SDF.
De controle of bepaalde declaraties uniek zijn wordt op dezelfde wijze uitgevoerd als in de oude ASF+SDF Checker. Er wordt een lijst opgebouwd van het te controleren element en elk moment dat er een element moet worden

[2] μCRL is een proces algebra taal ontwikkeld voor onderzoek naar communicerende processen.

toegevoegd aan deze lijst wordt er gekeken of het element nog niet aanwezig is in de lijst. Daarnaast bevat de μCRL Checker, in tegenstelling tot de ASF+SDF Checker, wel routines die de typecorrectheid van de specificatie controleert. Deze routines zijn in de ASF+SDF Checker niet nodig.
De μCRL Checker heeft hetzelfde probleem als de ASF+SDF Checker. De feitenextractie is niet consequent gescheiden van de controles die er plaats vinden. Deze techniek maakt het moeilijker om de checkers te onderhouden omdat er in elke fase van het proces fouten gegenereerd kunnen worden.
Ook de Pascal typechecker in [vD91] valt binnen de groep dat de typecheck functie toegepast is op de abstract syntax tree [HK95]. Het programma wordt hierdoor volgens de syntactische structuur van de taal stuk voor stuk gecontroleerd.
De Eiffel checker voorgesteld in [Vis92] maakt net als de nieuwe ASF+SDF Checker gebruik van een vorm van feitenextractie. De Eiffel checker maakt gebruik van een symbol table CONTEXT genaamd. Deze valt te vergelijken met de Rstore. Gedurende het controleren van de CONTEXT is het oorspronkelijke Eiffel programma niet meer nodig. Om de bewerkingen uit de voeren op deze CONTEXT is er de Context Modification Language ontwikkeld.

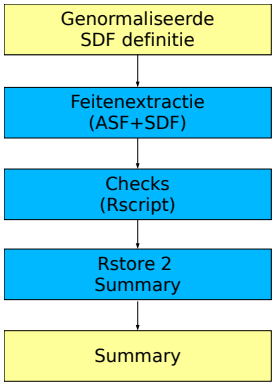
4 De ASF+SDF Checker

4.1 De architectuur

Het gebruik van Rscript zorgt voor een bijna standaard architectuur die hier ook is toegepast. Deze architectuur bestaat uit drie lagen. De eerste laag is verantwoordelijk voor de feitenextractie en is beschreven in ASF+SDF. Deze laag verwacht een geparseerde SDF definitie en levert aan de uitgang een Rstore op. In deze Rstore zitten de geëxtraheerde feiten welke geschikt zijn voor het controleren van een definitie.
De tweede laag doet de werkelijke well-formedness check en is geformuleerd in Rscript [Kli05]. De RscriptCalculator kan de hierboven genoemde Rstore verwerken. Het Rscript is verantwoordelijk voor het extraheren van de fouten en levert een Rstore op met de fouten.
De derde laag maakt het mogelijk om de Rstore uit de checker om te zetten in een error-summary. Dit is in wezen de visualisatie van de Rstore. Deze converter van Rstore naar error-summary is zo opgebouwd dat deze niet afhankelijk is van de controles die uitgevoerd worden in de checker. Het is mogelijk extra controles toe te voegen zonder deze converter aan te passen.
De functionaliteit van elke laag dient zo goed mogelijk gescheiden te blijven. De feitenextractie laag mag in principe geen fouten genereren. Dit is echter niet geheel mogelijk aangezien Rscript niet geschikt is om bijvoorbeeld op efficiënte wijze syntactische fouten te extraheren.
De voordelen van het zoveel mogelijk onafhankelijk maken van de componenten zijn dat de onderhoudbaarheid verbeterd wordt en fouten sneller geïsoleerd kunnen worden.
In Figuur 1 is de basisarchitectuur weergegeven.

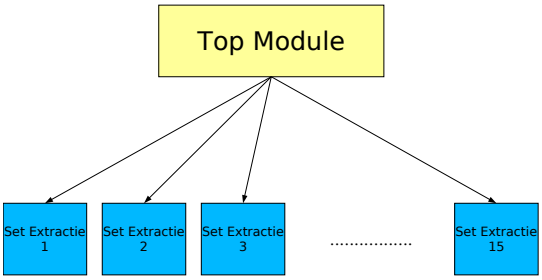
4.2 Feitenextractie

De feitenextractie is beschreven in ASF+SDF. De structuur is opgebouwd zoals is aanbevolen in [Kli05] en is in Figuur 2 weergegeven. Een voordeel



Figuur 1: De basisarchitectuur

van deze structuur is dat er een goede scheiding aanwezig is in de extractie van verschillende soorten feiten. Ook voorkomt het scheiden van de extracties dat de specificatie erg complex wordt [Kli05]. Bovendien kunnen er in de toekomst gemakkelijker extracties worden verwijderd of toegevoegd.
Het nadeel is echter dat er een klein snelheidsverlies optreedt, aangezien de SDF definitie meerdere keren doorlopen moet worden [Kli05]. De extractie modules zijn zo gedefinieerd dat ze onafhankelijk van de topmodule kunnen functioneren. Dit maakt het mogelijk om extracties alleenstaand te gebruiken of te hergebruiken in andere programma's.



Figuur 2: De structuur van de feitenextracter

4.3 De extractiecomponenten

De extractiecomponenten zijn uniform opgebouwd. Een extractie component bestaat uit één geëxporteerde functie en de rest is verborgen. De geëxporteerde functie is van de vorm *extract-<name>-facts(SDF, RSTORE) -> RSTORE*. Deze functie verwacht een SDF definitie en een (lege) Rstore. Het resultaat van deze functie is de Rstore waarbij een relatie is toegevoegd. De verschillende extract-functies kunnen op deze wijze gemakkelijk in cascade geschakeld worden en zijn niet afhankelijk van hun top module. Het nadeel is echter dat elke module op dit moment de gehele boom moet doorlopen van de geparseerde SDF definitie.
De extractie is opgebouwd uit een traversal op een gehele SDF definitie die

geactiveerd wordt op de plek waar een te extraheren feit zich bevindt.
Tijdens de ontwikkeling van de extract heeft de traversaloplossing voor enorme snelheidsproblemen geleid. Dit probleem is verholpen door de relatie op een andere manier te extraheren.

De relaties die worden toegevoegd aan de Rstore hebben dezelfde standaardvorm, namelijk *<NAME>, rel [loc , str] , { }*. De tupels hebben door het gebruik van een locatie aan de linkerzijde een unieke sleutel. De rechterzijde van de tupel vormt het gegeven welke bij deze locatie hoort. Door het gebruik van locaties kunnen feiten makkelijk aan elkaar gekoppeld worden. Een voorbeeld hiervan is het destilleren van alle start-symbols. Deze kunnen worden verzameld uit de relatie met sectie informatie en alle gedeclareerde soorten. De soorten waarvan de locatie past binnen de locatie van een start-symbol declaratie sectie zijn de start-symbols.

4.4 Nieuwe extractie aanpak

Het gebruik van traversals [vdBKV01] in de extractie modules heeft in het begin tot schokkende resultaten geleid. Deze techniek bleek zeer traag. Er is tijdelijk teruggeschakeld naar de manier waarop de oude checker zijn feiten extraheert. De oude methode houdt in dat de module handmatig wordt herschreven tot het niveau Sections. Vanaf de losse Sections werd er dan een traversal gebruikt om de feiten te extraheren. Eén experiment met een de oude methode A, één compleet gebaseerd op een enkele traversal B en de nieuwe traversal methode C geven het volgende resultaat[3]:

	Test-Term1 (209 loc)	Test-Term2 (1771 loc)
Oude Traversal A	0,32s	12m52,3s
Oude checker B	0,21s	8,88s
Nieuwe Traversal C	0,21s	4,72s

Tabel 1: Resultaten traversal prestatie meting

Uit Tabel 1 valt af te lezen dat de oude traversal code meer tijd nodig heeft. Bij grotere termen wordt dit verschil nog vele malen groter.
Het probleem zit hem echter niet in de werking van de traversal maar is verborgen in de volgende ASF vergelijking zoals voorgesteld in [Kli05] voor extractie:
extract-declared-symbols($Sort , $Rel) = $Rel union {< get-location($Sort) , symbol2str($Sort) >}
Door het gebruik van de union van Rscript ontstaat er het probleem dat er op een inefficiënte wijze een unieke set wordt opgebouwd. De traversal zal gedurende zijn executie bij elke valide knoop een *union* uitvoeren van het geëxtraheerde feit en de al bestaande relatie. Dit houdt in dat er bij elke toevoeging een controle plaats vindt of alle elementen in de relatie uniek zijn.
De nieuwe traversal extracter (voorbeeld in Appendix A3) voorkomt het elke keer controleren op de uniciteit van het element. De relatie wordt eerst geëxtraheerd als een lijst van tupels zonder gebruik te maken van de Rscript *union* functie. Dit houdt in dat er dubbele elementen aanwezig kunnen zijn

[3]Metingen zijn verricht op een FreeBSD 5.3 systeem draaiend op een Pentium 4 2.40GHz machine met 1024MB RAM

aangezien er niet bij het toevoegen van een tupel op uniciteit wordt getest. Deze lijst wordt door middel van de deprecated functie *unique* omgezet in een set van tupels. De uniekheid van de elementen wordt pas getoetst na de traversal en niet continu tijdens de traversal. Dit zorgt voor de enorme snelheidswinst.

4.5 Stringconversie

Het is op dit moment binnen ASF+SDF niet mogelijk om een terminal om te zetten naar een *strcon*. Dit zorgt binnen de module SdfOperations[4] voor extra code. Er dient voor bijvoorbeeld *attribute2str* voor elke mogelijke terminal een herschrijfregel te zijn. Het zou handig zijn als er een mogelijkheid zou bestaan om een gedeelte van de boom op een willekeurig moment weer om te zetten naar een *strcon*.

4.6 Extractie in praktijk: de interface

De opzet van checker is er nu op gebaseerd dat bij elke controle de gehele SDF definitie opnieuw ingelezen en geparseerd moet worden. Deze functionaliteit wordt verwezenlijkt door een ASF+SDF definitie die de volgende geëxporteerde functies realiseert: *extract-facts(SDF) -> RSTORE* en *extract-facts(SDF, ModuleId) -> RSTORE* . Deze topmodule biedt dezelfde mogelijkheden als de oude checker. Er kan enkel een SDF definitie worden herschreven naar een Rstore of er kan een SDF definitie inclusief een ModuleId worden meegegeven aan de functie.
Het telkens opnieuw extraheren van de feiten van de gehele SDF definitie is niet ideaal. Naarmate de definities groter worden is het behoorlijk tijdrovend om alle modules opnieuw na te lopen, zeker wanneer er maar enkele modules zijn gewijzigd. Dit kan vaak in de MetaEnvironment voorkomen. Een mogelijkheid om dit op te lossen is door per module een Rstore van zijn feiten bij te houden en enkel bij te werken wanneer de SDF module is gewijzigd. Wanneer de SDF definitie gecontroleerd moet worden zullen de relaties in de Rstores van de verschillende modules gefuseerd moeten worden. De huidige opbouw van de Rstore laat dit toe. Een programma om dit te verwezenlijken is echter nog niet gerealiseerd.
Aangezien de relaties van de modules onderling geen enkele invloed hebben op elkaar gedurende de extractie is deze oplossing mogelijk. Dit zal bij grote definities binnen de MetaEnvironment de extractietijd aanzienlijk kunnen verkorten.

4.7 De feiten

Voor de volledige ASF+SDF Checker worden momenteel de volgende feiten geëxtraheerd:

- Modules - De beschikbare modules, en de locatie bevat het gebied van de hele module.
- Imports - De imports, en de locatie waar deze zich bevinden.

[4]Deze module verzorgt de conversie van productieregel, charclass, literal, enzovoorts naar string binnen het extractie-proces.

- Sections - De sectie set bestaat uit een gebied, en het type gebied. Voorbeelden hiervan zijn: exports, hiddens, context-free syntax, aliases, en function production.
- Rules - De productieregels, en de locatie waar deze zich bevinden.
- Rule rhs - Het rhs gebied van een productieregel.
- Sorts - De soorten, en de locatie waar deze zich bevinden.
- Aliases - De rhs van de aliassen, en de locatie bevat het gebied van de gehele alias regel.
- Labels - De labels, en de locatie waar deze zich bevinden.
- Attributes - De attributen van een productieregel, en de locatie waar deze zich bevinden.
- Function literals - De functienaam, en de locatie bevat het gebied van de gehele productieregel.
- Function arguments - De *tweede ... n-de* argumenten van een functie, en de locatie van deze argumenten.
- Rule rhs literals - Literals die zich in de rhs van een productieregel bevinden, en de locatie waar deze zich bevinden.
- Renamings - De productieregel renamings, en de locatie waar deze zich bevinden.
- CharClasses - De CharClasses, en de locatie waar deze zich bevinden.
- Not supported symbols - De symbool constructies die niet ondersteund worden door ASF, en de locatie waar deze zich bevinden.

De sectie set heeft een sleutelrol binnen de opzet van deze extractie. Via de feiten in deze set kan de relatie gelegd worden tussen de andere feiten. Het is op deze wijze bijvoorbeeld mogelijk om te bepalen of een soort zich bevindt in een declaratie sectie of in een productieregel sectie.
Deze manier van opbouw van feitenextractie vermindert het effect, zoals geconcludeerd in [Kli05], dat de feitenextractie geoptimaliseerd is voor het Rscript. Dit is het effect dat de geëxtraheerde feiten volledig geoptimaliseerd zijn voor deze specifieke relationele verwerking, maar een extra last leggen bij de feitenextractie. Daarnaast kan dit effect de feiten minder generiek maken.
De sets kunnen algemener geformuleerd worden zonder dat Rscript te weinig informatie heeft. Het resultaat is ook zichtbaar in het aantal feiten dat geëxtraheerd moet worden. Bij de eerste versies liep het aantal feiten op tot 28 stuks. Door het toepassen van de set Sections is dit weer teruggedrongen tot 15 stuks.
In deze versie van de extracter wordt er geen onderscheid gemaakt tussen de feiten die geëxtraheerd worden voor de SDF Checker of ASF+SDF Checker. Dit zal voor een gering aantal onnodige sets zorgen wanneer alleen de SDF Check uitgevoerd moet worden.

4.8 Andere toepassingen van de feiten

Deze feiten zijn niet alleen zinvol voor de checker maar kunnen ook andere toepassingen dienen. Het is bijvoorbeeld mogelijk de importrelatie uit deze informatie te halen. Deze zou gecombineerd met hiddens en exports secties herschreven kunnen worden naar een dot-script [GKN02]. Het is met deze gegevens mogelijk om de importrelaties die verborgen zijn met een andere kleur weer te geven. In de huidige weergave is niet duidelijk te zien welke modules verborgen zijn.
Het is ook mogelijk om te controleren of de variabelen die gedeclareerd zijn in SDF gebruikt zijn in de bijbehorende ASF definitie. Op deze wijze kan er melding gemaakt worden van ongebruikte variabelen.

4.9 De ASF+SDF Checker Rscript

De checker zelf is beschreven in Rscript. Deze Rscript voert twee taken uit, namelijk het extraheren van extra feiten uit de Rstore en het uitvoeren van de controles. De volgende feiten worden geëxtraheerd in de Rscript:

- Bracket production - Productieregels die opgebouwd zijn als een *bracket*[5] regel.
- Function production - Productieregels die opgebouwd zijn als een *functie*[6] regel.
- Binary production - Productieregels die opgebouwd zijn als een *binary*[7] regel.
- Lexical syntax sections - De secties met lexicale syntax definities.
- Hidden sections - Verborgen secties.
- Export sections - Geëxporteerde secties.
- Traversal production - De productieregels die een opbouw hebben die geschikt is voor een traversal.
- Def parameter - Soorten die gedeclareerd zijn via module parameters.
- Def alias - Soorten die gedeclareerd zijn via een alias definitie.
- Used sort - Alle soorten die gebruikt zijn in syntax, prioriteiten, parameter en alias secties.
- Def sort - Alle soorten die gedeclareerd zijn.
- Rhs sorts rule - De rhs soorten van elke productieregel.
- Def start symbols - De soorten die gedefiniëerd zijn als start-symbol.
- Prior rule - Alle productieregels in een prioriteiten sectie.
- Syntax rule - Alle productieregels in een syntax sectie.

[5] Productieregel met de opmaak $Literal\ Sort\ Literal -> Sort$
[6] Productieregel met de opmaak $Literal(Sort, *) -> Sort$
[7] Productieregel met de opmaak $Sort\ Sort*\ Sort -> Sort$

- Rhs prior - Alle rh-zijdes van de prioriteiten secties met de prioriteiten sectie locatie als sleutel.
- Context free syntax sections - De Context Free Syntax secties.

Er zijn twee Rscripts, waarvan één de SDF Checker is en de ander de ASF+SDF Checker. Binnen deze Rscripts worden bijna alle testen uitgevoerd. Een klein aantal testen wordt nog uitgevoerd in de feitenextracter. Dit zijn de tests die vooral op syntax niveau zijn en erg lastig zijn te definiëren in Rscript. De volgende tests worden uitgevoerd inclusief de technologie waarin deze geïmplementeerd zijn:

- Waarschuwingen SDF
  - undeclared sort (Import afhankelijk, Rscript)
  - double declared sort (Import afhankelijk, Rscript)
  - double declared start-symbol (Import afhankelijk, Rscript)
  - illegal attribute (Rscript)
  - used in priorities but undefined (Importafhankelijk, Rscript)
  - inconsistent right-hand side in priorities (Rscript)
  - char in rule (Rscript)
  - aliased symbol already declared (Importafhankelijk, Rscript)
  - unused sort definitions (Rscript)
- Fouten SDF
  - module not available (Importafhankelijk, Rscript)
  - start-symbol not defined in any right hand (Importafhankelijk, Rscript)
  - literal in right-hand side not allowed in (ASF+SDF)
  - only sort allowed in right-hand-side of lexical-function in (Rscript)
  - double used label in (Rscript)
  - used symbol but not defined in any right-hand side (Rscript)
  - syntax definition contains no start symbols for module in definition (Importafhankelijk, Rscript)
- Waarschuwingen ASF+SDF
  - unknown constructor used in priorities (Rscript)
  - exported variables section in (Rscript)
  - kernel syntax construction in (Rscript)
  - production renamings not supported (ASF+SDF)
  - not supported symbol (ASF+SDF)
- Fouten ASF+SDF
  - constructor has already been used (Rscript)
  - traversal attributes in non-prefix function (Rscript)
  - illegal traversal attribute (Rscript)

missing bottom-up or top-down attribute (Rscript)

missing trafo and/or accu attribute (Rscript)

missing break or continue attribute (Rscript)

accutrafo should return tuple of correct types (Rscript)

accu should return accumulated type (Rscript)

trafo should return traversed type (Rscript)

inconsistent arguments of traversal attributes

(Importafhankelijk, Rscript)

inconsistent traversal attributes (Importafhankelijk, Rscript)

asf equation sort must not be used (Rscript)

charclasses not allowed in context-free syntax (Rscript)

Binnen de Rscript worden de uitkomsten van de tests samengevoegd in de relaties WARNINGS en ERRORS van de volgende vorm *rel* [*str , rel* [*loc , str*]]. De eerste string is de foutmelding. De relatie die erbij hoort zijn alle fouten en de locatie daarvan. Deze opzet heeft als voordeel dat binnen de Rscript definitie bepaald wordt welke fouten errors zijn of warnings. Eventuele herschrijvers naar andere representaties hoeven niet aangepast te worden wanneer er een wijzing van de indeling is van de resultaten of als er extra controles bij komen.
Hieronder staat een voorbeeld van deze aanpak.

```
rel[ str , rel[ loc , str ] ] WARNINGS = {
        <"undeclared sort: " , UNDECLARED_SORTS >,
        <"double declared sort: " , DOUBLE_DECLARED_SORTS >
}
```

UNDECLARED_SORTS en DOUBLE_DECLARED_SORTS zijn de relaties die de desbetreffende waarschuwingen bevatten. Een extra foutmelding toevoegen is een kwestie van een extra tupel toevoegen aan WARNINGS of ERRORS.

4.10 Opbouw en werking checks

Om de kracht van Rscript aan te tonen is hier de check afgebeeld die controleert op inconsequente right hand side van een prioriteiten sectie. De regel controleert op tupels met hetzelfde linker element.
De relatie RHS_PRIOR is een relatie van de locatie van de gehele prioriteitensectie en de right hand sids van de productieregels die in deze sectie voorkomen. Het resultaat van deze test moeten alle prioriteitensecties zijn die verschillende right hand sides bevaten. Aangezien elke tupel uniek is in een relatie [Kli05] mag elke prioriteitssectie maar vertegenwoordigd worden door één tupel. Wanneer een prioriteitssectie verschillende right hand sides bevat zal deze meerdere tupels hebben met dezelfde left hand side. De Rscript analyse regel:

```
rel[ &T1 , &T2 ] duplicatedLHS(rel[ &T1 , &T2 ] R) =
        { < V , X > | < &T1 V , &T2 X > : R, < &T1 W , &T2 Y > : R,
        X != Y, V == W }

rel[ loc , str ] INCONSISTENT_RHS_IN_PRIOR =
        duplicatedLHS( RHS_PRIOR )
```

De controle wordt uitgevoerd door de RHS_PRIOR feiten mee te geven aan de functie *duplicatedLHS*. Deze functie controleert of er dezelfde left hand sides aanwezig zijn voor andere right hand sides. Dit wordt gedaan door twee generatoren die de meegegeven relatie opsommen. Bij deze opsommingen wordt gekeken of er wordt voldaan aan de vergelijking of de linkerzijdes gelijk zijn $V == W$ en de rechterzijdes ongelijk $X! = Y$. Een voorbeeld hiervan is:

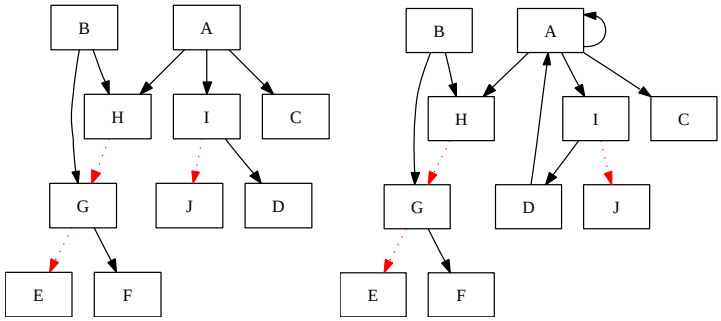
```
duplicatedLHS( { < "b" , "a" >, < "b" , "b" >,
        < "a" , "b" >, < "c" , "b" >,
        < "c" , "d" > } )
```

resulteert in

```
        { < "c" , "d" > , < "c" , "b" > ,
        < "b" , "b" > , < "b" , "a" > }.
```

4.11 Importafhandeling

SDF ondersteunt de mogelijkheid om andere modules te importeren. Daarbij heeft dit importsysteem de mogelijkheid gedeeltes verborgen te houden voor de andere modules. Dit zijn de zogenaamde hiddens secties. Alle geëxporteerde entiteiten van de geïmporteerde modules en alle indirect geïmporteerde modules worden zichtbaar voor de importerende module [vdBK04].
Een goede afhandeling van de importrelaties is essentieel voor het goed functioneren van deze checker. De checker dient alle modules die geïmporteerd worden op correcte wijze te controleren en rekening te houden met de importrelatie per module en welke conflicten er zijn tussen de modules. In Figuur 3 is de importstructuur te vinden van enkele definities die zijn beschreven ten behoeve van de regressietests. Zoals in Figuur 3 te zien is moet er rekening gehouden worden met de verborgen importrelaties en cyclische importrelaties.



Figuur 3: Twee importgraven die in de regressietests gebruikt worden. Verborgen imports zijn weergegeven door middel van een rode gestippelde pijl.

Het basis concept achter de hier gepresenteerde oplossing is om een relatie op te bouwen van de gebieden die voor een module zichtbaar zijn. Deze relatie bestaat in de praktijk uit tupels met aan de rechterzijde de module naam en aan de linkerzijde een zichtbaar gebied.
In deze specificatie worden twee functies toegepast *getContainedR1R2* en

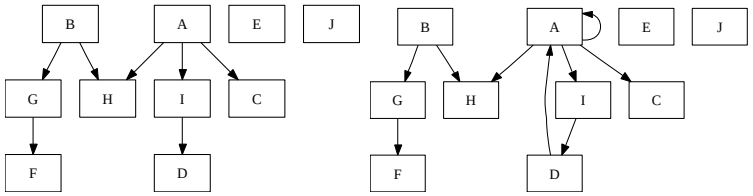
getNotContainedL2R2. De functie *getContainedR1R2*(R_1, R_2) heeft als argumenten twee relaties van de vorm $rel[loc, T]$ en het resultaat is van diezelfde vorm. Deze functie levert tupels op met de vorm $< T_1, T_2 >$. T_1 is de rechterkant van het tupel van R_1 en T_2 de rechterkant van het tupel van R_2. De voorwaarde gesteld in de comprehensie is dat het locatie element van het tupel van R_1 het locatie element van het tupel van R_2 moet omvatten.
Dezelfde functie is gedefiniëerd in *getContainedL2R2* met als verschil dat de tupels van het resultaat een andere vorm hebben. De overgebleven tupels zijn de tupels van R_2 waarvan de locatie wordt omvat door een locatie van een tupel uit R_1. De functie *getNotContainedL2R2* is de inverse van deze functie en levert de tupels op die zich niet in het gebied van R_2 bevinden. Hieronder staan de comprehensies weergegeven:

```
rel[ loc , &T2 ] getContainedL2R2(rel[ loc , &T1 ] R, rel[ loc , &T2] S) =
       { < W , Y > | < loc V , &T1 X > : R, < loc W , &T2 Y > : S, W <= V }
rel[ loc , &T2 ] getContainedL1R2(rel[ loc , &T1 ] R, rel[ loc , &T2] S) =
       { < V , Y > | < loc V , &T1 X > : R, < loc W , &T2 Y > : S, W <= V }
rel[ loc , &T2 ] getNotContainedL2R2(rel[ loc , &T1 ] R, rel[ loc , &T2] S) =
       S \ getContainedL2R2( R , S )
```

De net besproken functies zijn nodig voor de verdere analyse van de geïmporteerde secties. De eerste stap om de zichtbare gebieden te bepalen is om de transitieve afsluiting vast te stellen van de geëxporteerde importrelaties:

```
rel[ loc , str ] IMPORTS_VISIBLE = getNotContainedL2R2(HIDDEN_SECTIONS, IMPORTS)
rel[ str , str ] IMPORT_REL_VISIBLE = getContainedR1R2(MODULES, IMPORTS_VISIBLE)
rel[ str , str ] CLOSURE_IMPORT_REL_VISIBLE = IMPORT_REL_VISIBLE+
```

De eerste regel IMPORT_VISIBLE heeft als resultaat alle geëxporteerde importregels. Dit wordt verwezenlijkt door alle imports in de sectie hiddens te negeren.
De tweede regel IMPORT_REL_VISIBLE maakt een relatie aan van de vorm $rel[str, str]$ van de module en welke module deze importeert. Deze koppeling wordt gemaakt door te vergelijken met de functie *getContainedR1R2* in welke module een import valt.
De laatste vergelijken zorgt voor de transitieve afsluiting van de relatie IMPORT_REL_VISIBLE. Het is nu bekend welke modules indirect bereikbaar zijn. Figuur 4 geeft de geëxporteerde importrelatie weer.



Figuur 4: De importgraaf van enkel geëxporteerde importrelaties.

De volgende stap is het bepalen van de relatie waarin is opgeslagen wat de bereikbare modules zijn per module. Deze set bestaat uit de importrelaties van de module zelf en de modules die bereikbaar zijn via de transitieve afsluiting van de geëxporteerde importrelaties. Deze extracties zijn als volgt gedefiniëerd:

```
rel[ str , str ] IMPORT_REL = getContainedR1R2(MODULES, IMPORTS)
rel[ str , str ] MODULE_REACHABLE =
                        IMPORT_REL union
                        { < A , Y > | < str A , str X > : IMPORT_REL,
                        < str R , str Y > : CLOSURE_IMPORT_REL_VISIBLE, R == X}
```

De eerste query IMPORT_REL stelt een relatie vast tussen de module en welke deze allemaal importeert. Hierbij worden ook de imports opgenomen die verborgen zijn.
De relatie MODULE_REACHABLE is de relatie waarin per module alle bereikbare import modules staan vermeld. Dit is een vereniging van alle eigen geïmporteerde modules (IMPORT_REL) en de modules die zichtbaar zijn via de geïmporteerde modules. Het laatste gedeelte van de *union* verzamelt alle indirect bereikbare modules.
De laatste stap in het proces is het vaststellen van de relatie met de tupels met de module naam en de locatie die zichtbaar is. De relatie met de bereikbare modules wordt omgezet in een relatie waarbij de modulenaam wordt gekoppeld aan de bereikbare export gebieden. De verborgen gebieden worden niet meegenomen. Ook wordt deze set verenigd met het gehele gebied van de eigen module. De comprehensie ziet er als volgt uit:

```
rel[ loc , str ] MODULE_VISIBLE_AREAS =
                        { < A , X > | < str X , str Y > : MODULE_REACHABLE,
                        < loc P , str Q > : MODULES,
                        < loc A , str Z > : EXPORT_SECTIONS,
                        Y == Q, A <= P  }
                        union
                        MODULES
```

De relatie MODULE_VISIBLE_AREAS wordt samen met de functie *getVisible* gebruikt in een comprehensie die de losse modules afloopt. Dit is bijvoorbeeld te zien in Appendix D bij het vaststellen van de dubbel gedeclareerde soorten.
Het nadeel van deze methode is dat deze processorintensief is en deze voor extra code zorgt in de vorm van comprehensie en aanroepen van *getVisible* bij de regels die de werkelijke controle uitvoeren.

4.12 Rstore2summary

De Rstore die uit de Rscript gebaseerde checker komt is niet geschikt om verder verwerkt te worden door de MetaEnvironment. De MetaEnvironment verwacht een error-summary. In ASF+SDF is een programma beschreven die de Rstore kan herschrijven naar een error-summary. Op dit moment kunnen alleen errors en warnings worden herschreven, maar fatals en info zullen ook gemakkelijk geïmplementeerd kunnen worden.
De applicatie houdt zich niet bezig met welke controle een waarschuwing of fout is. Er worden twee relaties verwacht WARNINGS en ERRORS welke omgezet zullen worden naar een Summary.

5 Vergelijking met oude checker

Naast het feit of het überhaupt mogelijk is om een ASF+SDF Checker te beschrijven met behulp van Rscript is het interessant hoe deze nieuwe checker zich verhoudt ten opzichte van de oude checker. In de komende secties zullen de deelvragen hierover beantwoord worden.

5.1 Prestaties

Tabel 2 laat de gegevens zien van de tijdsduur van de tests van de regressietest omgeving. Zoals uit de tabel valt af te lezen is de gecompileerde oude

checker verreweg het snelst. Aangezien er voor Rscript nog geen gecompileerde versie gegenereerd kan worden is alleen de geïnterpreteerde versie van de oude checker interessant. De nieuwe checker lijkt sneller als er naar het gemiddelde wordt gekeken en dit klopt ook bij de simpelere SDF definities. Maar de standaard deviatie en het maximum zijn hoger dan bij de oude checker. Wanneer de SDF definitie complexer en groter is zal het controleproces bij de nieuwe checker steeds meer tijd gaan kosten. De oude checker heeft echter minder last van dit proces en presteert op snelheid beter.
De checker gebaseerd op Rscript presteert op snelheid zo slecht dat op dit moment een industriële toepassing niet haalbaar is. Binnen het huidige prototype zijn de feitenextractie en de bash-scripts geoptimaliseerd. De prestatieproblemen zitten in de ASF+SDF Checker Rscript specificatie en de RscriptCalculator zelf. Het vermoeden is dat met een gecompileerde versie van de Rscript een behoorlijke inhaalslag gemaakt kan worden.

	μ	maximum	σ
Oude checker(gecompileerd)	0,26s	0,42s	0,69s
Oude checker(geïnterpreteerd)	2,61s	4,37s	0,82s
Nieuwe checker(geïnterpreteerd)	2,48s	7,41s	1,93s

Tabel 2: Resultaten tijd meting

5.2 Omvang code

Naast de prestatie is de omvang van de code ook een interessante parameter. In tabel 3 is het aantal regels[8] code te zien van de oude en de nieuwe checker. De nieuwe checker bestaat uit de feitenextractie module en het Rscript die te samen 2169 regels vertegenwoordigen. Het verschil met de oude checker is ongeveer 1000 regels code.
Het valt op te merken dat de feitenextractie het leeuwendeel van de regels code op zich neemt. Zoals opgemerkt in [Kli05] is de vraag of er iets gedaan kan worden aan deze scheve verhouding tussen extractie en Rscript door een generieke extractie methode.
Uit het aantal regels code van de op Rscript gebaseerde checker blijkt wel dat Rscript een zeer krachtig middel is voor analyse gebaseerd op relatie calculus. Met het gebruik van Rscript is op dit gebied al een flinke stap gezet.

	regels code
Oude checker	3257
Feitenextractie	1761
ASF+SDF Checker Rscript	408

Tabel 3: Aantal regels code oude en nieuwe checker.

[8] Geïmporteerde bibliotheek definities niet meegerekend.

5.3 Toepasbaarheid Rscript

De resterende vraag is nog in hoeverre Rscript geschikt is voor de taak van een well-formedness checker. Uit dit onderzoek is gebleken dat Rscript voor de meeste opgestelde checks geschikt is. Veel checks zijn op relationele verhoudingen gebaseerd en kunnen goed beschreven worden in Rscript. Er is in ieder geval één gebied waar Rscript minder geschikt voor is. Dat zijn de syntactische controles. Deze controles zijn in ASF+SDF in enkele regels te definiëren terwijl ze in Rscript haast onmogelijk zijn. Een voorbeeld hiervan is de not supported symbol controle. Deze fouten worden door de feitenextractie verzameld in een relatie en Rscript voegt weinig aan de controle toe. In de checker wordt bij dit soort controles hoogstens gefilterd op sectie niveau (prioriteiten, contextvrije syntax enz).
Er moet goed worden afgewogen welke controles in ASF+SDF gedefiniëerd dienen te worden en welke in Rscript. Wanneer de juiste balans gevonden is kunnen beide omgevingen elkaar goed aanvullen. Aangezien de meeste controles voornamelijk zijn beschreven in Rscript is de toepasbaarheid van Rscript bij dit soort problemen goed.

6 Teststructuur

De correctheid en kwaliteit van de checker dient gewaarborgd te blijven. Het is hierdoor noodzakelijk dat er een vorm van testen aanwezig is. Hiervoor is een kleine testset opgebouwd. Een diepgaande testanalyse zoals voorgesteld in [PTvV00] was niet haalbaar binnen dit project.

6.1 Regressietests

De tests zijn onderverdeeld in de 32 foutmeldingen die door de checker gegenereerd kunnen worden. Elke foutmelding heeft twee tot drie tests. Een aantal van deze tests bevat een complexe importrelatie en modulestructuur om het import gedeelte van de checker te kunnen testen.
Voor elke SDF definitie in de testset is een Rstore gegenereerd. Deze is met de hand gecontroleerd en opgeslagen in een bestand. Door het genereren van een tweede Rstore kan met het commando Rstorediff worden gecontroleerd of beide Rstores hetzelfde zijn. Er is ook een bash script *regress-terms* aanwezig waarmee alle tests automatisch doorlopen kunnen worden.
De opzet van een regressietest systeem [PTvV00] heeft het mogelijk gemaakt om de Rscripts van de checker te verbeteren, terwijl de uitkomst gemakkelijk gecontroleerd kan worden. Ook biedt het opzetten van tests de ruimte om na te denken over hoe foutmeldingen geïnterpreteerd moeten worden en wat een goede SDF definitie is en wat niet. In [Vis97a] staat veel beschreven over hoe SDF definities opgebouwd horen te zijn.

6.2 Rstores vergelijken

Het vergelijken van Rstores kan niet zomaar gedaan worden door het MetaEnvironment commando atdiff of het unix commando diff. Deze vergelijkingsprogramma's werken goed wanneer de Rstores geheel identiek zijn, maar wanneer de volgorde anders is zullen de genoemde programma's een foutmelding geven. Binnen de ASF+SDF Checker is het niet van belang in

welke volgorde de fouten zijn opgesomd in de relaties.
Stel er zijn twee verzamelingen; A $\{a, b, c\}$ en B $\{b, c, a\}$. De authentieke vergelijkingsprogramma's zullen deze als ongelijk betitelen terwijl in dit geval het correct is als $A \setminus B \bigcup B \setminus A$ geen elementen meer bevatten. De volgorde van de elementen is nu niet meer van belang.
De Rstorediff implementeert deze functionaliteit specifiek voor de vergelijking van twee Rstores uit de ASF+SDF Checker. Rstorediff is gespecificeerd in ASF+SDF aangezien het noodzakelijk is om twee Rstores in te kunnen lezen. Dit is gerealiseerd door een functie te definiëren die twee Rstores als argumenten heeft. De interne vergelijking tussen de Rstores is opgezet door middel van ingebedde Rscripts in de ASF+SDF specificatie. Deze opzet werkt voor kleine specificaties goed en geeft een zeer krachtige toevoeging aan ASF+SDF. De code is ook zonder problemen te compilen naar een direct uitvoerbaar programma. Toch is het bij grote Rscript specificaties verstandig deze te scheiden van ASF+SDF aangezien de complexiteit van de code waarschijnlijk erg snel toeneemt.

7 Ontwikkelingsproces

Het ontwikkelingsproces zoals voorgesteld in [Kli05] sloot goed aan bij dit project. Het proces lijkt de omgekeerde wereld aangezien eerst met de Rscript moet worden begonnen alvorens de feitenextractie wordt gedefiniëerd. Het eerst uitdenken van de Rscript-queries geeft een goed beeld van de feiten die nodig zijn.
Nadat de eerste versie van de ASF+SDF Checker gerealiseerd was bleek het mogelijk om weer opnieuw de feitenextractie te herzien. Het aantal feiten kon gereduceerd worden tot 15 stuks.

7.1 Ontwikkelomgeving

Er is op dit moment nog geen Rscript ontwikkelomgeving aanwezig. Het schrijven en testen van een feitenextractie en Rscript systeem zou niet gemakkelijk zijn als er niet een aantal voorzieningen waren getroffen.
Aangezien de snelheid van de geïnterpreteerde feitenextractie te hoog oploopt bij grote definities wordt de feitenextractie omgezet in een uitvoerbaar programma door middel van asfc [vdBHKO02]. Rondom deze compiler is een bash-script (*extract*) geschreven waarin een SDF definitie wordt geparseerd en doorgestuurd naar de uitvoerbare feitenextracter. Er zijn ook bash-scripts (*sdfcheck*, *asfsdfcheck*) aanwezig die na dit proces de Rscript uitvoeren. Deze twee scripts vormen een uitvoerbare checker waarbij een SDF definitie wordt meegegeven en een Rstore als resultaat wordt overgehouden. Het checken van een definitie bestaat uit het invoeren van het commando *sdfcheck foo.sdf* of *asfsdfcheck foo.sdf*.
Het opzetten van deze losse programma's gebeurt in een make-proces en is volledig geautomatiseerd. De vergelijkingen, de parseertabellen en de programma's worden allemaal gebouwd gedurende dit proces. Het aanpassen van de feitenextractie houdt in dat een aantal van deze bestanden bijgewerkt moeten worden. Via het make-proces kan dit geautomatiseerd plaatsvinden. Het onwikkelproces is hierdoor veel efficiënter geworden. Het aanpassen van de feitenextractie vergt enkel een *make* en kan direct getest worden in samenwerking met de Rscripts. Een verandering van de Rscripts is direct te

testen, ook al vraagt de nieuwe opzet uit hoofdstuk 7.2 dat deze met de hand geparseerd moet worden.
In de oude omgeving is er geen manier aanwezig waarmee de verschillende programma's (semi)-geautomatiseerd aangeroepen worden en vernieuwd worden. Dit vele handwerk is in de huidige ASF+SDF Checker distributie weggewerkt. De bash-scripts voeren de losse stappen uit zoals parseren en herschrijven met de juiste parseertabellen. Hierdoor is het ontwikkelen en testen veel van de checkers gemakkelijker en sneller verlopen.

7.2 De bash-scripts

De prestatie is één van de belangrijke aspecten van een nieuwe ASF+SDF Checker om deze werkbaar te maken. De prestaties van de bash-script konden verbeterd worden. Het commando *rscript* van de Rscript distributie wordt niet meer gebruikt in de bash-scripts en de RscriptCalculator wordt nu direct aangeroepen na de feitenextractie.
Het bleek hierdoor mogelijk te zijn om een aantal parser stappen over te slaan. De Rstore van de feitenextractie wordt als geparseerde boom aangeleverd aan de RscriptCalculator. De Rscript definitie wordt gedurende het bouwen van de ASF+SDF Checker distributie geparseerd, zodat deze niet elke keer bij het uitvoeren van de checker opnieuw geparseerd hoeft te worden. Daarnaast wordt deze Rscript niet meer gecontroleerd op fouten. Door het overslaan van deze stappen is er een snelheidswinst van ongeveer één seconde[9] geboekt. Vooral bij het controleren van kleine SDF specificaties scheelt deze aanpak aanzienlijk.
Het aanpassen van de Rscript van de ASF+SDF Checker moet nu echter met de hand geparseerd worden. Ook moet de Rscript voor het parsen gecontroleerd worden of deze type correct is. Deze nadelen zijn echter alleen van toepassing bij onderhoud of ontwikkelingswerkzaamheden. Het bash-script *parseRscript* voert deze taak uit en slaat de geparseerde Rscript in de juiste map op.

8 RscriptCalculator aanbevelingen

Tijdens het ontwerpen zijn er een aantal zaken naar voren gekomen binnen Rscript die voor verbetering vatbaar zijn. Deze verbeteringen zullen het gemak en de prestatie positief kunnen beïnvloeden.

8.1 Samenvoegen van strings

Binnen Rscript ontbreekt er een functie waarmee strings aan elkaar gekoppeld kunnen worden. Het ontbreken van deze mogelijkheid maakt het bijvoorbeeld moeilijk om nette foutmeldingen op te bouwen, zoals wel mogelijk is binnen ASF+SDF. Een ander punt is dat twee of meerdere string feiten die aan elkaar gekoppeld moeten worden nu aan een set toegevoegd dienen te worden, terwijl deze feiten ook als één string aan elkaar gekoppeld kunnen worden. Een voorbeeld hiervan is constructor van een productieregel en de rhs van deze productieregel. Op dit moment wordt de koppeling tussen

[9] Metingen zijn verricht op een FreeBSD 5.3 systeem draaiend op een Pentium 4 2.40GHz machine met 1024MB RAM

deze twee gegevens opgeslagen in de vorm van een tupel. Een aaneenkoppeling van de twee gegevens in één string zou ook voldoen en zou de code overzichtelijker maken.

8.2 Rscript imports

Het verschil tussen de SDF Checker en de ASF+SDF Checker is nu gerealiseerd door twee losse definities. Het zou een verbetering zijn als Rscript de mogelijkheid zou bieden om andere Rscripts te importeren. De voordelen die geboekt kunnen worden zijn vooral de beheersbaarheid van grote projecten en het overzicht.
Het is in de huidige opzet wel mogelijk om Rscripts in cascade te schakelen. Deze optie is in dit geval niet haalbaar aangezien de functies die zowel in de SDF Checker als de ASF+SDF Checker niet meegegeven worden in de Rstore.

8.3 Snelheid

De huidige methode om een Rscript uit te voeren is op basis van een interpreter. Deze oplossing zorgt nog voor veel prestatieverlies en maakt de nieuwe ASF+SDF Checker ongeschikt om toegepast te worden binnen de MetaEnvironment. Waar de oude checker met maximaal enkele seconden klaar is met de check van een grote specificatie, kan de Rscript gebaseerde checker enkele tot tientallen minuten aan het rekenen zijn[10].
De oude checker is ook wanneer deze geïnterpreteerd wordt uitgevoerd iets sneller dan de nieuwe Rscript versie. De oude checker controleert echter minder compleet dan de nieuwe Rscript gebaseerde checker. Vooral de intensieve importrelatie checks zijn beter geïmplementeerd in de nieuwe checker. De vraag die echter blijft is of een gecompileerde Rscript versie de snelheid van de oude checker kan evenaren. Het vermoeden is dat dit het geval is aangezien de relatie calculus implementaties van moderne programmeertalen snel zijn.
In deze checker wordt er veel gebruik gemaakt van vergelijkingen tussen locaties. Deze vergelijking bestaat uit vier getalvergelijkingen en de stringvergelijking van de bestandsnamen. De vergelijking van de vier getallen is een snelle en efficiënte methode. Het controleren van de bestandsnamen kan ook op efficiënte wijze plaats vinden aangezien er gebruik gemaakt wordt van maximal subterm sharing [vdBdJKO00]. Het is nu alleen noodzakelijk om de twee pointers met elkaar te vergelijken in plaats van elke losse karakter van de string.

9 Toekomst

Het meest opvallende knelpunt in het opzetten van een Rscript systeem is de feitenextractie. Het definiëren van een generiek raamwerk zou dit probleem kunnen verbeteren waardoor de verhouding Rscript en de extractie beter komt te liggen.
Naast dit probleem is de importanalyse nog erg complex. Er is veel extra code nodig om deze functionaliteit toe te voegen. Rscript biedt nog geen mogelijkheden om de importanalyse op eenvoudige wijze toe te voegen aan een

[10] Metingen zijn verricht op een FreeBSD 5.3 systeem draaiend op een Pentium 4 2.40GHz machine met 1024MB RAM

query. Aangezien in veel moderne talen gewerkt wordt met geïmporteerde modules of bibliotheken zou het handig zijn als hier een extra functionaliteit voor zou komen.
Het is waarschijnlijk mogelijk dat deze checker gedeeltelijk ook geschikt is om andere grammatica's te controleren. De typische SDF checks zouden daarvoor verwijderd moeten worden uit Rscript en de feitenextractie module zou hiervoor aangepast moeten worden. Het biedt in ieder geval mogelijkheden om verder te bouwen aan een generieke checker in Rscript welke ook gebruikt kan worden voor andere grammatica definitie formalismen.

9.1 Nieuwe checks

Naast de huidige tests welke allemaal zijn overgenomen van de oude checker is er nu ruimte voor nieuwe checks. Rscript biedt de mogelijkheden om complexere controles uit te voeren. Hierbij valt te denken aan een controle naar de bereikbaarheid van productieregels op basis van transitieve afsluiten of het gebruik van geïmporteerde modules. Dit soort analyses maken het refactoren van een grammatica eenvoudiger.

9.2 Generieke Rstore vergelijking

De rstorediff die toegepast is in de regressietests van dit project is alleen geschikt voor de Rstores van deze toepassing. Deze specificatie is echter niet geschikt om willekeurige Rstores te vergelijken. Het zou een mooie aanvulling zijn als een generieke rstorediff opgenomen zou worden in de Rscript distributie. Het opzetten van regressietesten [PTvV00] of tussentijds controleren van veranderingen aan een extractie of een Rscript wordt hierdoor gemakkelijker.

9.3 SGLR foutmeldingen

Een andere opvallende zaak is het feit dat de parser SGLR zelf weinig terugkoppeling geeft wanneer er iets mis gaat. De gebruiker moet het meestal doen met een *Parse error: character ' ' unexpected*. Dit maakt het soms lastig om bepaalde fouten te herstellen, terwijl de fouten vaak simpel van aard zijn.
Een goede terugkoppeling van de fouten met de informatie wat de parser wel verwacht zou het foutherstellen van ASF+SDF specificaties vereenvoudigen.

10 Conclusie

De Rscript methode is geschikt voor het analyseren van grammatica definities en biedt een gestructureerd raamwerk om de controles uit te voeren. Rscript maakt complexere checks mogelijk en de code is overzichtelijker. De betere scheiding tussen feitenextractie en checks maakt de nieuwe ASF+SDF Checker overzichtelijker.
Er is echter nog een groot nadeel. De snelheid van de RscriptCalculator is een belemmering om grote Rscripts en/of Rstores door te rekenen. Dit maakt het momenteel nog niet mogelijk om deze nieuwe checker in te zetten als vervanging voor de huidige checker. Een RscriptCompiler zou dit probleem waarschijnlijk kunnen oplossen.
Wanneer dit probleem is verholpen biedt de Rscript gebaseerde ASF+SDF

Checker een hoop extra mogelijkheden en een veel geavanceerder analyse mechanisme dan de oude checker.

11 Projectopzet

Het onderzoek is opgedeeld in een aantal onderdelen:

- Onderzoeksbeschrijving.
- Literatuuronderzoek naar typechecking, ASF+SDF en Rscript.
- Aanpak en ontwerp-definitie nieuwe ASF+SDF Checker.
- ASF+SDF definitie voor feitenextractie.
- ASF+SDF/Rscripts definities voor checker.
- Scriptie.

De eerste drie onderdelen zijn uitgevoerd in het Voorbereidings Master Project. Deze vormen de basis van het onderzoek. De laatste drie zijn uitgevoerd gedurende de drie maanden durende onderzoek bij het Centrum voor Wiskunde en Informatica.

12 Fasering

De gedetailleerde tijdlijn van het masterproject is als volgt:

- Voorbereiding Master Project: Inlezen en ontwerp typechecker
- Week 1: Paper Voorbereidings Master Project en ontwerp checker
- Week 2: Ontwerp eerst feitenextractie(soorten) + testset en basis opzet projectfiles
- Week 3: Rscript voor double declared sort en output compatible maken met MetaEnvironment
- Week 4: Extra feitenextractie traversals en test-files
- Week 5: Rscript definities behorende bij deze feiten
- Week 6: Extra feitenextractie traversals en test-files
- Week 7: Rscript definities behorende bij deze feiten
- Week 8: Extra feitenextractie traversals en test-files
- Week 9: Rscript definities behorende bij deze feiten
- Week 10: Uitloop om laatste zaken af te maken en eventueel start scriptie
- Week 11: Scriptie
- Week 12: Scriptie
- Week 13: Scriptie

Gedurende het Voorbereidings Master Project is de onderzoeksvraag helder en concreet gemaakt en is inzicht verkregen hoe het probleem aangepakt moest worden.
De uitvoering van het project is begonnen met een klein experiment om gevoel te krijgen voor de feitenextractie en Rscript. Hierbij is meteen de ontwikkelomgeving opgezet en een versiebeheer server.
Hierna is gestart met een proces waarbij eerst gewerkt wordt aan de feitenextractie en vervolgens aan het Rscript en dit herhaalt zich drie keer. In dit proces wordt rekening gehouden met het voorgestelde proces in [Kli05]. Het beschrijven van de Rscript geeft namelijk beter inzicht in de benodigde feiten en deze kunnen verwerkt worden in de feitenextractie. In deze tabel komt het proces van eerst Rscript queries schrijven en dan de feitenextractie niet naar voren. Dit is veroorzaakt door het feit dat er eerst een vorm van extractie nodig is voordat een Rscript getest kan worden[11]. Echter voordat begonnen is met het opstellen van de eerste feitenextractie is wel nagedacht over de vorm van de Rscript queries en vanuit dat beeld zijn de feiten die nodig zijn bepaald.

[11] Er kan natuurlijk met een tijdelijke dummy Rstore gewerkt worden.

13 Evaluatie

In dit hoofdstuk wordt besproken hoe het project bevallen is en of de resultaten bevredigend zijn. Ook zal er gekeken worden hoe dit onderzoek beter aangepakt had kunnen worden.

13.1 Positief

Dit onderzoek is soepel verlopen en er waren weinig tegenvallers. De nieuwe ASF+SDF Checker werkt en kan als prototype ingezet worden voor verdere uitwerking naar een productierijpe checker.
Er waren geen problemen met de MetaEnvironment of Rscript zodat het ontwikkelproces niet werd vertraagd. De voorbeelden uit de Rscript handleiding [Kli05] werkten in enkele uren. Er is geen tijd verloren gegaan aan het werkend krijgen van de MetaEnvironment en de Rscript omgeving.
Naast de positieve resultaten van het eindproduct heb ik zelf ook veel geleerd van dit onderzoek. Ik heb gewerkt met een omgeving waar ik nog weinig ervaring mee had.

13.2 Negatief

Er zijn weinig dingen verkeerd gegaan gedurende dit onderzoek. Het enige obstakel is nu nog de traagheid van de RscriptCalculator. Dit vormt nog een belemmering om de nieuwe checker toe te passen in een productie omgeving.

13.3 Tevredenheid opdrachtgever

Dit onderzoek levert voor de opdrachtgever nieuwe inzichten in de manier hoe Rscript ingezet kan worden in een echte applicatie. Wat voor problemen komen hierbij naar voren of welke juist niet. Ook is er een ontdekking gedaan met betrekking tot het efficiënt feiten extraheren.
Daarnaast is er de Rscript gebaseerde ASF+SDF Checker. Deze checker voert de checks nauwkeuriger uit dan de oude checker en is momenteel geschikt voor testdoeleinden.

13.4 Reflectie op onderzoeksaanpak

Deze manier van onderzoeken is goed verlopen. Het liep allemaal redelijk vlekkeloos. Het enige punt dat in de toekomst beter aangepakt zou kunnen worden is het kijken naar werk van anderen. In dit onderzoek is dat een beetje in het verschiet geraakt door de dwang om een werkend prototype te bouwen.

14 Dankwoord

In een van de eerste colleges over ASF+SDF had ik het idee dat ik meer met deze technologie wilde doen. Na een inspirerend gesprek met Mark van den Brand is het een onderwerp geworden waarbij ik me verder kon verdiepen in ASF+SDF, de werking van de MetaEnvironment en Rscript. En als stagebegeleider op het CWI.
Ik wil Paul Klint bedanken voor het oplossen van het zwarte gat aan het begin van het project en samen met Tijs van der Storm als vraagbaak over

Rscript in het algemeen. Voor het oplossen van de prestatie problemen met de traversals heeft Jurgen een grote bijdrage geleverd.

Referenties

[ASU88] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[cal] http://www.cs.vu.nl/∼steven/calce/.

[Cle03] ClearJump. *ClearParse SDK Programmer's Guide*, 2003. ClearParseSDK.pdf.

[ea60] P. Naur et al. Report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, 1960.

[ea03] T. Parr et al. *ANTLR Reference Manual*, 2003. http://www.antlr.org/doc/index.html.

[ebn] iso/iec 14977:1996(e).

[GJ98] D. Grune and C. Jacobs. *Parsing techniques A Practical Guide*. Ellis Horwood Limited, 1998.

[GKN02] E. Gansner, E. Koutsofios, and S. North. Drawing graphs with dot, February 2002.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[HK95] J. Hillebrand and H. Korver. A well-formedness checker for μcrl. Technical Report P9501, Universiteit van Amsterdam, 1995.

[Kli03] P. Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC03)*, pages 2–12. IEEE Computer Society, 2003.

[Kli05] P. Klint. A tutorial introduction to rscript: a relation approach to software analysis, 2005. CWI.

[met] http://www.cwi.nl/htbin/sen1/twiki /bin/view/SEN1/MetaEnvironment.

[PTvV00] M. Pol, R. Teunissen, and E. van Veenendaal. *Testen volgens TMap*. Uitgeverij Tutein Nolthenius, 2 edition, 2000.

[vD91] A. van Deursen. An algebraic specification for the static semantics of pascal. In *201*, page 93. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 31 1991.

[vdBdJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[vdBHKO02] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The asf+sdf compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[vdBK04] M.G.J. van den Brand and P. Klint. Asf+sdf meta-environment user manual, 2004. CWI.

[vdBKV01] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. Technical report, Centrum voor Wiskunde en Informatica, 2001.

[Vis92] E. Visser. Syntax and static semantics of Eiffel. A case study in algebraic specification techniques. Unpublished technical report, December 1992.

[Vis97a] E. Visser. A family of syntax definition formalisms, 1997. UvA.

[Vis97b] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, Universiteit van Amsterdam, juli 1997.

A Oude extractie methode

Deze ASF+SDF specificatie is de oude methode waarop feiten werden geëxtraheerd. Binnen deze methode wordt een gedeelte van de boom met de hand uitgeschreven voordat er een traversal wordt toegepast. Deze techniek is gebaseerd op de manier waarop de oude ASF+SDF Checker de feiten extraheert. Deze methode is omslachtig en minder overzichtelijk.

A.1 SDF definitie

```
module SortsExtractor

imports BasicImports
imports utilities/PosInfo[Symbol]

exports
     context-free syntax
        extract-facts(SDF)      -> Rstore

hiddens
     context-free start-symbols
                                Rstore

     context-free syntax
        extract-parameter-symbols(Module, Rel[[Elem]])
                -> Rel[[Elem]] {traversal(accu, break, top-down)}
        extract-parameter-symbols(ModuleName, Rel[[Elem]])
                -> Rel[[Elem]] {traversal(accu, break, top-down)}

        extract-sort(Section, Rel[[Elem]])
                -> Rel[[Elem]] {traversal(accu, break, top-down)}
        extract-sort(Grammar, Rel[[Elem]])
                -> Rel[[Elem]] {traversal(accu, break, top-down)}
        extract-sort(Symbols, Rel[[Elem]])
                -> Rel[[Elem]] {traversal(accu, break, top-down)}
        extract-sort(Symbol*, Rel[[Elem]])
                -> Rel[[Elem]] {traversal(accu, break, top-down)}
        extract-sort(Symbol, Rel[[Elem]])
                -> Rel[[Elem]] {traversal(accu, break, top-down)}

     variables
        "$Rel" [0-9\']*         -> Rel[[Elem]]
        "$Rstore" [0-9\']*      -> Rstore
        "$Definition"[0-9]*     -> Definition
        "$ModuleId"[0-9]*       -> ModuleId

        "$Sort"[0-9]*           -> Sort
        "$Symbols"[0-9]*        -> Symbols

     context-free syntax
        extract-modules(Definition, Definition, Rel[[Elem]])
                -> Rel[[Elem]]
        extract-module(Module, Definition, Rel[[Elem]])
                -> Rel[[Elem]]
        extract-module-extracts(Module, Rel[[Elem]])
                -> Rel[[Elem]]
        extract-sections(Module, Rel[[Elem]])
                -> Rel[[Elem]] {traversal(accu, break, top-down)}
        extract-sections(Section, Rel[[Elem]])
                -> Rel[[Elem]] {traversal(accu, break, top-down)}

     variables
        "$Module"[0-9]*         -> Module
        "$Module*"[0-9]*        -> Module*
        "$Grammar"[0-9]*        -> Grammar
```

A.2 ASF definitie

```
equations

[esf-1]  $Definition2 := remove-labels($Definition1),
         $Rel := extract-modules($Definition2, $Definition2, {}),
         $Rstore2 := assign(SORTS, rel[area,varname], $Rel, Rstore())
         ====>
         extract-facts(definition $Definition1) = $Rstore2

[ems-1]  $Rel2 := extract-module($Module, $Definition, $Rel1)
         ====>
         extract-modules($Module $Module*, $Definition,  $Rel1) =
                extract-modules($Module*, $Definition, $Rel2)

[ems-2]  extract-modules(, $Definition, $Rel) = $Rel

[em-1]   $Rel2 := extract-module-extracts($Module, $Rel1)
         ====>
         extract-module($Module, $Definition, $Rel1) = $Rel2

[emes-1] $Rel3 := extract-parameter-symbols($Module, {}),
         $Rel4 := extract-sections($Module, $Rel3),
         $Rel2 := $Rel4 union $Rel1
         ====>
         extract-module-extracts($Module, $Rel1) = $Rel2

[es-1]   extract-sections( exports $Grammar, $Rel) =
                extract-sort( $Grammar , $Rel )

[es-2]   extract-sections( hiddens $Grammar, $Rel) =
                extract-sort( $Grammar , $Rel )

[eps-1]  extract-parameter-symbols($ModuleId[$Symbols], $Rel) =
                 extract-sort($Symbols, $Rel)

[eso-1]  extract-sort($Sort, $Rel) =
                 $Rel union {<get-location($Sort), symbol2str($Sort)>}
```

B Oude traversal extractie methode

Deze ASF+SDF specificatie is de oude traversal methode waarop feiten werden geëxtraheerd. Deze methode is minder omslachtig dan de vorige methode. Door het gebruik van *union* in de traversal is deze code erg traag.

B.1 SDF definitie

```
module SortsTraversal

imports BasicImports
imports utilities/PosInfo[Sort]

exports
     context-free syntax
        extract-facts(SDF)      -> Rstore

hiddens
     context-free start-symbols
                                Rstore

     context-free syntax
        extract-declared-symbols(Definition, Rel[[Elem]])
                -> Rel[[Elem]] {traversal(accu, break, top-down)}
        extract-declared-symbols(Sort, Rel[[Elem]])
                -> Rel[[Elem]] {traversal(accu, break, top-down)}

     variables
```

```
        "$Rel" [0-9\']*  -> Rel[[Elem]]
        "$Rstore" [0-9\']*      -> Rstore

        "$Definition"[0-9]*     -> Definition
        "$Sort"[0-9]*           -> Sort
```

B.2 ASF definitie

```
equations

[esf-1]  $Definition2 := remove-labels($Definition1),
         $Rel := extract-declared-symbols($Definition2, {}),
         $Rstore2 := assign(SORT, rel[ loc , str ], $Rel, Rstore())
         ====>
         extract-facts(definition $Definition1) = $Rstore2

[edss-1] extract-declared-symbols($Sort, $Rel) =
                $Rel union {<get-location($Sort), symbol2str($Sort)>}
```

C Nieuwe traversal extractie methode

Deze ASF+SDF specificatie is de huidige traversal methode waarop feiten werden geëxtraheerd. Deze methode is iets minder compact dan de vorige, maar stukken compacter dan de eerste. Daarnaast heeft deze methode geen last van prestatieverlies en is deze zelfs de snelste.

C.1 SDF definitie

```
module SortsNieuw

imports BasicImports
imports utilities/PosInfo[Sort]

exports
     context-free syntax
        extract-facts(SDF)      -> RSTORE

hiddens
     context-free start-symbols
                                RSTORE

     context-free syntax
        extract-declared-symbols(Definition, {Elem ","}* )
                -> {Elem ","}* {traversal(accu, break, top-down)}
        extract-declared-symbols(Symbol, {Elem ","}*)
                -> {Elem ","}* {traversal(accu, break, top-down)}

     variables
        "$Rel" [0-9\']*         -> Rel[[Elem]]
        "$Rstore" [0-9\']*      -> RSTORE
        "$Elems" [0-9\']*       -> {Elem ","}*

        "$Definition"[0-9]*     -> Definition
        "$Sort"[0-9]*           -> Sort
```

C.2 ASF definitie

```
equations

[esf-1]  $Elems := extract-declared-symbols($Definition, ),
         $Rel := unique({ $Elems }),
         $Rstore2 := assign(SORT, rel[ loc , str ], $Rel, rstore())
         ====>
```

```
         extract-facts(definition $Definition) = $Rstore2

[edss-1] extract-declared-symbols($Sort, $Elems) =
                $Elems , <get-location($Sort), symbol2str($Sort)>
```

D Rscript van de SDF Checker

Het Rscript zoals toegepast is voor de SDF Checker.

```
%%BEGIN declared sets

set[ str ] ASF_SORTS_DEF = {"ASF-Tag","ASF-TagId","ASF-ConditionalEquation",
        "ASF-Equation","ASF-Implies","ASF-Condition","ASF-Equations"}
set[ str ] LEGAL_TRAVERSAL_ATTRIBUTES_DEF = {"accu","trafo","break",
        "continue","top-down","bottom-up"}
set[ str ] KERNEL_SYNTAX_SECTIONS_DEF = {"syntax","priorities"}

set[ str ] BRACKET_SECTIONS_DEF = {"bracket production"}
set[ str ] FUNCTION_PRODUCTION_SECTIONS_DEF = {"function production"}
set[ str ] BINARY_PRODUCTION_SECTIONS_DEF = {"binary production"}
set[ str ] LEXICAL_SECTIONS_DEF = {"lexical priorities","lexical syntax"}
set[ str ] HIDDEN_SECTIONS_DEF = {"hiddens"}
set[ str ] EXPORT_SECTIONS_DEF = {"exports"}
set[ str ] TRAVERSAL_SECTIONS_DEF = {"accu","trafo","accu-trafo",
        "not-implemented"}
set[ str ] PARAMETER_SECTIONS_DEF = {"parameter"}
set[ str ] RENAMING_SECTIONS_DEF = {"renaming"}
set[ str ] ALIAS_SECTIONS_DEF = {"aliases"}
set[ str ] VARIABLES_SECTIONS_DEF = {"variables","lexical variables"}
set[ str ] SYNTAX_SECTIONS_DEF = {"context-free syntax","lexical syntax",
        "syntax"}
set[ str ] PRIORITY_SECTIONS_DEF = {"priorities","lexical priorities",
        "context-free priorities"}
set[ str ] DECL_SORT_SECTIONS_DEF = {"sort declaration"}
set[ str ] DECL_START_SECTIONS_DEF = {"start-symbol declaration"}
set[ str ] CONTEXT_FREE_SYNTAX_SECTIONS_DEF = {"context-free syntax"}
set[ str ] ATTRIBUTE_DEF = {"bracket","left","assoc","right","non-assoc"}
rel[ loc ,str ] BUILTIN_SORTS_DEF =
        {< area-in-file ( "" , area ( 0 , 0 , 0 , 0 , 0 , 0 ) ) , "CHAR">,
        < area-in-file ( "" , area ( 0 , 0 , 0 , 0 , 0 , 0 ) ) , "LAYOUT" >}

%%END declared sets

%%BEGIN functions

rel[ &T1 , &T2 ] duplicatedRHS(rel[ &T1 , &T2 ] R) =
        { < V , X > | < &T1 V , &T2 X > : R, < &T1 W , &T2 Y > : R,
        V != W, X == Y }

rel[ &T1 , &T2 ] duplicatedLHS(rel[ &T1 , &T2 ] R) =
        { < V , X > | < &T1 V , &T2 X > : R, < &T1 W , &T2 Y > : R,
        X != Y, V == W }

rel[ &T1 , &T2 ] getContainedR1R2(rel[ loc , &T1 ] R, rel[ loc , &T2] S) =
        { < X , Y > | < loc V , &T1 X > : R, < loc W , &T2 Y > : S, W <= V }

rel[ loc , &T2 ] getContainedL2R2(rel[ loc , &T1 ] R, rel[ loc , &T2] S) =
        { < W , Y > | < loc V , &T1 X > : R, < loc W , &T2 Y > : S, W <= V }

rel[ loc , &T2 ] getContainedL1R2(rel[ loc , &T1 ] R, rel[ loc , &T2] S) =
        { < V , Y > | < loc V , &T1 X > : R, < loc W , &T2 Y > : S, W <= V }

rel[ loc , &T2 ] getNotContainedL2R2(rel[ loc , &T1 ] R, rel[ loc , &T2] S) =
        S \ getContainedL2R2( R , S )

rel[ &T1 , &T2 ] relDifference(rel[ &T1 , &T2 ] R, rel[ &T1 , &T2 ] S) =
        rangeR( R , range( R ) \ range ( S ) )

rel[ loc , &T1 ] getVisible(rel[ loc, str ] M_V_AREAS, rel[ loc, &T1 ] R,
         set[str] T_MODULE) =
```

```
        { < W , Y > | < loc V , str X > : M_V_AREAS, < loc W , &T1 Y > : R,
         str M : T_MODULE, M == X, W <= V }

%%END functions

%%Imports from rstore

str MODULE
rel[ loc , str ] MODULES
rel[ loc , str ] SECTIONS
rel[ loc , str ] IMPORTS
rel[ loc , str ] RHS_RULE
rel[ loc , str ] RULE
rel[ loc , str ] LABELS
rel[ loc , str ] ATTRIBUTES
rel[ loc , str ] RHS_LITERAL_RULE
rel[ loc , str ] SORTS
rel[ loc , str ] RULE_RENAMING
rel[ loc , str ] CHARCLASS
rel[ loc , str ] NOT_SUPPORTED_SYMBOLS
rel[ loc , str ] FUNCTION_LITERAL
rel[ loc , str ] FUNCTION_ARGUMENTS

rel[ loc , str ] BRACKET_PRODUCTION =
        rangeR( SECTIONS , BRACKET_SECTIONS_DEF )
rel[ loc , str ] FUNCTION_PRODUCTION =
        rangeR( SECTIONS , FUNCTION_PRODUCTION_SECTIONS_DEF )
rel[ loc , str ] BINARY_PRODUCTION  =
        rangeR( SECTIONS , BINARY_PRODUCTION_SECTIONS_DEF )
rel[ loc , str ] LEXICAL_SYNTAX_SECTIONS =
        rangeR( SECTIONS , LEXICAL_SECTIONS_DEF )
rel[ loc , str ] HIDDEN_SECTIONS =
        rangeR( SECTIONS , HIDDEN_SECTIONS_DEF )
rel[ loc , str ] EXPORT_SECTIONS =
        rangeR( SECTIONS , EXPORT_SECTIONS_DEF )
rel[ loc , str ] TRAVERSAL_PRODUCTION =
        rangeR( SECTIONS , TRAVERSAL_SECTIONS_DEF )
rel[ loc , str ] DEF_PARAMETER =
        getContainedL2R2( rangeR( SECTIONS , PARAMETER_SECTIONS_DEF ),
                                SORTS)

rel[ loc , str ] DEF_RENAMING =
        getContainedL2R2( RHS_RULE, getContainedL2R2( rangeR( SECTIONS ,
        RENAMING_SECTIONS_DEF ), SORTS))

rel[ loc , str ] DEF_ALIAS =
        getContainedL2R2(RHS_RULE, getContainedL2R2( rangeR( SECTIONS ,
        ALIAS_SECTIONS_DEF ), SORTS))

rel[ loc , str ] USED_SORT =
        getContainedL2R2( rangeR( SECTIONS , ( SYNTAX_SECTIONS_DEF
        union VARIABLES_SECTIONS_DEF union PRIORITY_SECTIONS_DEF union
        PARAMETER_SECTIONS_DEF union RENAMING_SECTIONS_DEF union
        ALIAS_SECTIONS_DEF)), SORTS ) \
        ( DEF_ALIAS union DEF_PARAMETER union DEF_RENAMING )

rel[ loc , str ] DEF_SORT =
        BUILTIN_SORTS_DEF union DEF_ALIAS union getContainedL2R2(
        rangeR( SECTIONS, DECL_SORT_SECTIONS_DEF ),
        SORTS)

rel[ loc , str ] RHS_SORTS_RULE = getContainedL2R2( RHS_RULE , USED_SORT )

rel[ loc , str ] DEF_START_SYMBOLS =
        getContainedL2R2( rangeR( SECTIONS , DECL_START_SECTIONS_DEF ),
        SORTS )

rel[ loc , str ] PRIOR_RULE =
        getContainedL2R2(rangeR( SECTIONS , PRIORITY_SECTIONS_DEF ),
        RULE )

rel[ loc , str ] SYNTAX_RULE =
        getContainedL2R2( rangeR( SECTIONS , SYNTAX_SECTIONS_DEF union
        VARIABLES_SECTIONS_DEF ), RULE )
```

```
rel[ loc , str ] RHS_PRIOR =
        getContainedL1R2( rangeR( SECTIONS , PRIORITY_SECTIONS_DEF ),
                                          RHS_RULE )

rel[ loc , str ] CONTEXT_FREE_SYNTAX_SECTIONS =
        rangeR( SECTIONS , CONTEXT_FREE_SYNTAX_SECTIONS_DEF )


rel[ loc , str ] EXPORTRED_PARAMETER_AREA =
        getNotContainedL2R2( HIDDEN_SECTIONS union EXPORT_SECTIONS ,
        DEF_PARAMETER union DEF_RENAMING )

%%BEGIN extract import relations and visible areas per module

rel[ loc , str ] IMPORTS_VISIBLE = getNotContainedL2R2(HIDDEN_SECTIONS, IMPORTS)
rel[ str , str ] IMPORT_REL = getContainedR1R2(MODULES, IMPORTS)
rel[ str , str ] IMPORT_REL_VISIBLE = getContainedR1R2(MODULES, IMPORTS_VISIBLE)
rel[ str , str ] CLOSURE_IMPORT_REL = IMPORT_REL+
rel[ str , str ] CLOSURE_IMPORT_REL_VISIBLE = IMPORT_REL_VISIBLE+


rel[ str , str ] MODULE_REACHABLE =
        IMPORT_REL
        union
        { < A , Y > | < str A , str X > : IMPORT_REL,
        < str R , str Y > : CLOSURE_IMPORT_REL_VISIBLE, R == X}

rel[ loc , str ] MODULE_VISIBLE_AREAS =
        { < A , X > | < str X , str Y > : MODULE_REACHABLE,
        < loc P , str Q > : MODULES,
        < loc A , str Z > : EXPORT_SECTIONS union EXPORTRED_PARAMETER_AREA ,
        Y == Q, A <= P  }
        union
        MODULES

rel[ loc , str ] VARIABLES_SECTIONS = rangeR( SECTIONS , VARIABLES_SECTIONS_DEF )

%%END extract import relations and visible areas per module

%%BEGIN SETs extracting

set[ str ]       TOP_MODULES =
        top ( CLOSURE_IMPORT_REL )
set[ str ]       SET_MODULES =
        range(MODULES)
rel[ loc , str ] LEXICAL_RULE =
        getContainedL2R2( LEXICAL_SYNTAX_SECTIONS , SYNTAX_RULE )
rel[ loc , str ] SYNTAX_ATTRIBUTES =
        rangeR( ATTRIBUTES , ATTRIBUTE_DEF )
rel[ loc , str ] EXPORTED_SORT =
        getContainedL2R2( EXPORT_SECTIONS , DEF_SORT )

%%END SETs extracting

%%BEGIN SDF Check functions

rel[ loc , str ] DOUBLE_DECLARED_SORTS =
        { < X , Y > | str W : SET_MODULES, < loc X , str Y > :
        duplicatedRHS(getVisible( MODULE_VISIBLE_AREAS, DEF_SORT \ DEF_ALIAS ,
        {W}))}

rel[ loc , str ] DOUBLE_DECLARED_START_SYMBOLS =
        { < X , Y > | str W : SET_MODULES, < loc X , str Y > :
        duplicatedRHS(getVisible( MODULE_VISIBLE_AREAS, DEF_START_SYMBOLS,
        {W}))}

rel[ loc , str ] UNDEFINED_SORTS =
        relDifference( DEF_SORT \ BUILTIN_SORTS_DEF, getContainedL2R2(
        rangeR( SECTIONS , SYNTAX_SECTIONS_DEF ), RHS_SORTS_RULE ) )

rel[ loc , str ] UNUSED_SORTS =
        relDifference( EXPORTED_SORT , USED_SORT )
```

```
rel[ loc , str ] UNDECLARED_SORTS =
        { < X , Y > | str W : SET_MODULES, < loc X , str Y > :
        relDifference( getVisible( MODULE_VISIBLE_AREAS, RHS_SORTS_RULE, {W}),
        getVisible( MODULE_VISIBLE_AREAS, DEF_SORT union DEF_PARAMETER
        union DEF_RENAMING, {W}) union BUILTIN_SORTS_DEF )}
        union
        { < X , Y > |  str W  : SET_MODULES, < loc X , str Y > :
        relDifference( getVisible( MODULES, USED_SORT union
        DEF_START_SYMBOLS, {W}), getVisible( MODULE_VISIBLE_AREAS, DEF_SORT
        union DEF_PARAMETER union DEF_RENAMING, {W} ) union BUILTIN_SORTS_DEF)}

rel[ loc , str ] START_SYMBOL_NOT_IN_RHS =
        { < X , Y > | str W : SET_MODULES, < loc X , str Y > :
        relDifference( getVisible( MODULE_VISIBLE_AREAS, DEF_START_SYMBOLS, {W}),
        getVisible( MODULE_VISIBLE_AREAS, RHS_RULE, {W}) )}

rel[ loc , str ] USED_IN_PRIORITIES_BUT_UNDEFINIED =
        { < X , Y > | str W : SET_MODULES, < loc X , str Y > :
        relDifference( getVisible( MODULES, PRIOR_RULE, {W}),
        getVisible( MODULE_VISIBLE_AREAS, SYNTAX_RULE, {W}) )}

rel[ loc , str ] INCONSISTENT_RHS_IN_PRIOR =
        duplicatedLHS( RHS_PRIOR )

rel[ loc , str ] MODULE_NOT_AVAILABLE =
        relDifference( IMPORTS , MODULES )

rel[ loc , str ] CHAR_IN_RULE =
        rangeR( getNotContainedL2R2( VARIABLES_SECTIONS , USED_SORT ), {"CHAR"} )


rel[ loc , str ] MULTI_SORT_RHS_LEX_FUN =
        relDifference( getContainedL2R2( LEXICAL_RULE ,
        rangeR( RHS_RULE  , range( RHS_RULE  ) \ {"LAYOUT"} ) ) ,
        getContainedL2R2( LEXICAL_RULE , RHS_SORTS_RULE ) )

rel[ loc, str ] DOUBLE_USED_LABEL =
        { < V , W > |  < loc V , str W > : RULE ,
        < loc X , str Y > : duplicatedRHS(getContainedL2R2( {< V , W >},
        LABELS )) }

rel[ loc , str ] ILLEGAL_ATTRIBUTE =
        getNotContainedL2R2( BRACKET_PRODUCTION ,
                SYNTAX_ATTRIBUTES[-,"bracket"] x {"bracket"} ) union
        getNotContainedL2R2( BINARY_PRODUCTION ,
                SYNTAX_ATTRIBUTES[-,"left"] x {"left"} ) union
        getNotContainedL2R2( BINARY_PRODUCTION ,
                SYNTAX_ATTRIBUTES[-,"right"] x {"right"} ) union
        getNotContainedL2R2( BINARY_PRODUCTION ,
                SYNTAX_ATTRIBUTES[-,"assoc"] x {"assoc"} ) union
        getNotContainedL2R2( BINARY_PRODUCTION ,
                SYNTAX_ATTRIBUTES[-,"non-assoc"] x {"non-assoc"} )

rel[ loc , str ] ALIASED_SYMBOL_DOUBLE_DECLARED =
        { < X , Y > | str W : SET_MODULES, < loc X , str Y > :
        duplicatedRHS(getVisible( MODULE_VISIBLE_AREAS, DEF_ALIAS, {W}))}

rel[ loc , str ] LITERAL_IN_RHS =
        RHS_LITERAL_RULE

rel[ loc , str ] NO_STARTSSYMBOL_DEFINED =
        { < V , X > | < loc V , str X > : MODULES,  str M  : TOP_MODULES,
        int I : { #getVisible( MODULE_VISIBLE_AREAS, DEF_START_SYMBOLS, {M}) },
        I == 0, M == X }

%%END SDF Check functions

%%BEGIN check yields

rel[ str , rel[ loc , str ] ] WARNINGS_SDF = {
<"undeclared sort: " , UNDECLARED_SORTS >,
<"double declared sort: " , DOUBLE_DECLARED_SORTS >,
<"double declared start-symbol: " , DOUBLE_DECLARED_START_SYMBOLS >,
<"illegal attribute: " , ILLEGAL_ATTRIBUTE >,
```

```
<"used in priorities but undefined: " , USED_IN_PRIORITIES_BUT_UNDEFINIED >,
<"inconsistent right-hand side in priorities: " , INCONSISTENT_RHS_IN_PRIOR >,
<"char in rule: " , CHAR_IN_RULE >,
<"aliased symbol already declared: " , ALIASED_SYMBOL_DOUBLE_DECLARED >,

<"unused sort definitions: " ,UNUSED_SORTS >}

rel[ str , rel[ loc , str ] ] ERRORS_SDF = {
<"module not available: " , MODULE_NOT_AVAILABLE >,
<"start-symbol not defined in any right hand: " , START_SYMBOL_NOT_IN_RHS >,
<"literal in right-hand side not allowed in: " , LITERAL_IN_RHS >,
<"only sort allowed in right-hand-side of lexical-function in: " ,
        MULTI_SORT_RHS_LEX_FUN >,
<"double used label in: " , DOUBLE_USED_LABEL >,

<"used symbol but not defined in any right-hand side: " , UNDEFINED_SORTS >,
<"syntax definition contains no start symbols for module in definition: " ,
        NO_STARTSSYMBOL_DEFINED >}

%%END check yields
```