Universiteit Twente
faculteit der informatica

De Virtuele Verhalenverteller:

voorstel voor het gebruik van een *upper-ontology* en een nieuwe architectuur.

door R.S.Kooijman

in het kader van een INF vrij-project
aan Universiteit Twente
onder begeleiding van dr. M. Theune
Juli 2004

Inleiding

Aan de Universiteit Twente zijn op het moment van dit schrijven twee studenten afgestudeerd op een project dat begeleid wordt door Mariët Theune: de Virtuele Verhalenverteller. De algemene doelstelling van de Virtuele Verhalenverteller zou geformuleerd kunnen worden als: het genereren en presenteren van een verhaal. Kenmerken van het systeem zijn: het is een multi-agent systeem, verhalen worden gegenereerd op basis van verhaalkarakters die gerepresenteerd worden door middel van agents, en sturing en opbouw van een plot onder begeleiding van een regisseur die gerepresenteerd wordt door een andere agent, presentatie van het gegenereerde verhaal gebeurt door middel van gesproken tekst en een visuele representatie van de verteller.

Dit verslag zal globaal twee onderdelen van de Virtuele Verhalenverteller behandelen. Het eerste onderwerp zal gericht zijn op een verbetering van de flexibiliteit van het systeem, en dan met name een voorstel voor een nieuwe architectuur. Dan wordt onderzocht in hoeverre de bestaande ontologie van het systeem, de storyworldontology, vervangen kan worden door een ontologie met een meer formele en gestandaardiseerde basis. De algemene doelstelling is om een aantal voorstellen voor verbetering te doen voor de tot nu toe beschikbare versies van de Virtuele Verhalenverteller.

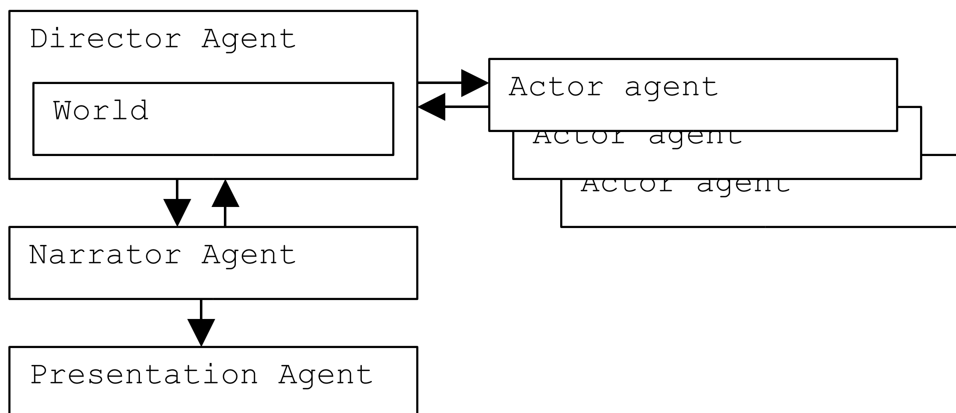
Inhoudsopgave

1.Nieuwe architectuur voor het MAS....4
1.1Verdeling van verantwoordelijkheden....5
1.2Communicatiemodel, StoryAgentOntology....6
1.2.1Informatie-uitwisseling ten behoeve het uiteindelijke verhaal....6
1.2.2Organisatorische informatie-uitwisseling....7
1.2.3De StoryAgentOntology....7
1.3Veranderingen ten aanzien van de flexibiliteit....9
1.3.1Problemen met betrekking tot de StoryWorldOntology....9
1.3.2Oorzaak sterke koppeling tussen ontologie en rest van het systeem....9
1.3.3Scheiding op basis van ontologieën....10
2.SUMO als basis voor de verhalenwereld....12
2.1Waarom?....12
2.2Waarom SUMO?....12
2.3Hoe?....13
2.4SUMO naar Jess vertalen....14
2.4.1De vertaler....16
2.5Samenstellen van een 'upper ontology' als deelverzameling van SUMO....16
2.5.1Selectie van modules uit SUMO....17
2.5.2Selectie bij parsing....18
2.5.3Selectie na parsing....18
2.6Samenstellen van een 'application specific ontology'....19
2.7Voorbeeld uit het verhaal van Sander Faas....19
2.7.1Plops motivatie....19
2.8Honger als gewaarwording....21
2.8.1ValuedAttribute....21
2.8.2Perception....22
2.9Redenatie van de ActorAgent....23
2.9.1Goal selection....24
2.9.2Backward-chaining....25
2.9.3Action selection....25
2.9.4Module layout....25
3.Prototype....27
3.1SUMO naar Jess vertaler....27
3.1.1parsing....27
3.1.2transfer....27
3.1.3generation....27
3.2Gebruik van SUMO en de nieuwe architectuur....27
3.2.1Functionaliteit....27
3.2.2Problemen bij implementatie....28
3.2.3Wat er wel en niet werkt....29
4.Conclussies....30
5.Referenties....31

1.Nieuwe architectuur voor het MAS

De Virtuele Verhalenverteller is een multi-agent systeem, dat wil zeggen een computerprogramma dat bestaat uit autonoom opererende programma onderdelen, die in samenwerking met elkaar een verhaal samenstellen. De verschillende agents hebben elk een rol toebedeeld gekregen. De afhankelijkheid tussen de verschillende rollen in het systeem kan worden gemodelleerd door een architectuur waarin de afhankelijkheden, verantwoordelijkheden, eventuele communicatie en dergelijke zijn beschreven.

Volgens [RENS2004] is de huidige architectuur voor het multi-agent systeem van de Virtuele Verhalenverteller niet ideaal. Allereerst wordt hier de huidige architectuur en de voorgestelde architectuur geïntroduceerd, waarna er dieper in wordt gegaan op de motivatie voor de voorgestelde architectuur.



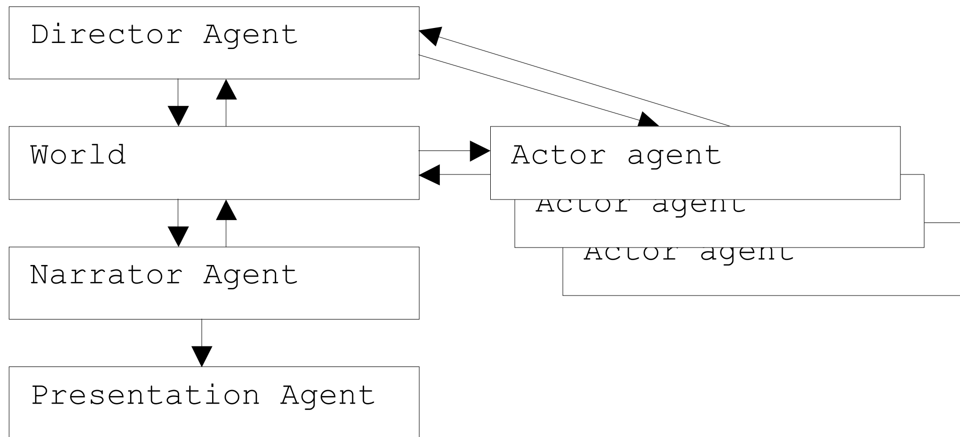
Figuur 1: De huidige architectuur van het MAS

In deze architectuur communiceren de acteurs (Actor agent) met de regisseur (Director Agent) over bepaalde acties die de acteurs in het verhaal kunnen uitvoeren. De verteller (Narrator Agent) wordt door de regisseur op de hoogte gehouden van de wereld van het verhaal (World). De Presentation Agent presenteert het verhaal aan de gebruiker.

De volgende beperkingen van de hieronder getoonde architectuur kunnen worden opgesomd:

- De director kan invloed uitoefenen op wat er verteld wordt, doordat de director verantwoordelijk is voor het doorsturen van acties die zich in de verhalenwereld af hebben gespeeld.
- De narrator heeft niet de mogelijkheid om autonoom de wereld te observeren, dit gaat altijd via de director.
- Het bijhouden van een verhalenwereld past niet direct bij de rol van regisseur die in eerste instantie aan de Director Agent is toegewezen.
- Er moet altijd een director aanwezig zijn.

In [RENS2004] wordt een architectuur voorgesteld voor het MAS van de Virtuele Verhalenverteller, waarin de verantwoordelijkheden van de verschillende agents duidelijker zijn onderscheiden. Het belangrijkste verschil zit in de loskoppeling van het model van de wereld van de director als aparte agent.



Figuur 2: De voorgestelde architectuur van het MAS

Twee gevolgen voor het ontwerp van het systeem door deze verandering van de architectuur willen we hier noemen:

- De verantwoordelijkheden van de verschillende agents zijn enigszins veranderd.
- Hierdoor zal ook de communicatie tussen de agents anders zijn dan nu het geval; wat moet er met welke agent gecommuniceerd worden?

In dit hoofdstuk wordt verder ingegaan op deze twee aspecten. Daarnaast wordt ingegaan op andere aanpassingen aan het systeem die de flexibiliteit moeten verhogen.

1.1Verdeling van verantwoordelijkheden

Voordat de nieuwe architectuur geïmplementeerd kan worden, is het misschien handig om nog even te kijken naar de motivatie en globale invulling van een dergelijke structuur.

Volgens de Gaia analyse- en ontwerpmethodologie is het bij een MAS belangrijk dat voor elke agent duidelijk is vastgelegd welke rollen deze agent op zich moet nemen [WOOL2002]. Volgens deze methode kunnen de verschillende agents mogelijk verschillende rollen vervullen in het systeem, waarbij een rol gedefinieerd wordt door: verantwoordelijkheden, permissies, activiteiten en protocollen. Hieronder staat een lijst met de verschillende soorten agents die voor kunnen komen in het MAS van de Virtuele Verhalenverteller, met hun rollen en verantwoordelijkheden.

- **DirectorAgent**
  De regisseur is verantwoordelijk voor een van de eisen die in [FAAS2002] worden opgelegd aan het plot: het behouden van een goede structuur [THEU2004]. De globale structuur wordt door de regisseur beheerst door: toestaan van acties, en introduceren van nieuwe elementen in de wereld, het wijzigen/toevoegen van doelen voor agents (*goals*), en het selecteren van episoden. Verder is deze regisseur verantwoordelijk voor het opstarten van de JADE agents. De DirectorAgent is in principe de agent waarmee de gebruiker in eerste instantie mee zal interacteren.

- **WorldAgent**
  De WorldAgent zorgt initieel voor de beschrijving van de verhalenwereld. In de Virtuele Verhalenverteller is deze beschrijving in de vorm van predikaten in de redeneertaal Jess. Deze agent is een ‘alwetende agent’ die als een soort centrale

gegevensbank kennis over de wereld van het verhaal moet bijhouden. Deze kennis kan worden opgestuurd in de vorm van *percepts* (indrukken, gewaarwordingen) naar overige agents. Omdat de karakters een (virtuele) werkelijkheid met elkaar moeten delen is het handig om dit centraal bij te houden.

- **ActorAgent**
  Vertegenwoordigt een karakter in het verhaal. De karakters zijn verantwoordelijk voor een andere eis die in [FAAS2002] worden opgelegd aan het plot: consistentie [THEU2004]. De ActorAgent vraagt toestemming aan de DirectorAgent om acties uit te voeren. Deze agent krijgt van de WorldAgent indrukken (*percepts*) opgestuurd die dit karakter op dit moment ervaart, op basis waarvan een actie kan worden gekozen, en van de DirectorAgent doelen (*goals)* toegewezen, op basis waarvan een actie kan worden gekozen. Een uit te voeren actie moet worden doorgegeven aan de WorldAgent.

- **NarratorAgent**
  Genereert zinnen op basis van entiteiten die in de verhalenwereld aanwezig zijn en acties die ActorAgents hebben uitgevoerd. De NarratorAgent interpreteert in feite de wereld en houdt de toehoorder op de hoogte van belangrijke veranderingen.

- **PresentationAgent**
  Deze agent presenteert het verhaal door middel van onder andere *text-to-speech* van de tekst zoals die door de NarratorAgent wordt gegenereerd.

1.2Communicatiemodel, StoryAgentOntology

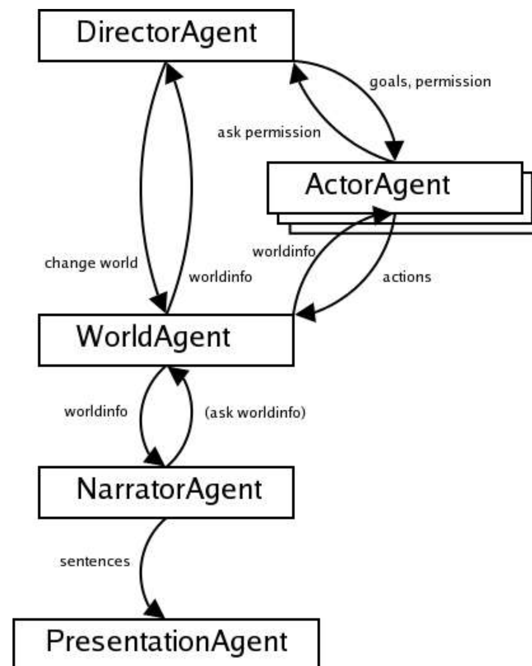
In dit multi-agent systeem vormt communicatie tussen de agents een belangrijk aspect, omdat de interactie tussen acteurs en verhalenwereld, acteurs en regisseur, en acteurs onderling, de uitwerking van het plot verzorgt. Daarnaast moeten de agents in het MAS ook het een en ander weten over elkaar. Eerst wordt behandeld *wat* er gecommuniceerd moet worden en vervolgens *hoe*.

1.2.1Informatie-uitwisseling ten behoeve het uiteindelijke verhaal

Aan de hand van de verdeling van verantwoordelijkheden zoals beschreven in paragraaf 1.1 kunnen we bepalen welke soorten van informatie moeten worden uitgewisseld tussen de verschillende agents. Hieronder een lijst met mogelijke soorten van informatie en hun vorm.

- De toestand van de verhalenwereld in de vorm van predikaten die voldoen aan de *StoryWorldOntology (zie 1.2.3).*
- Toestemming voor het uitvoeren van acties. Parameters: acteur en actie.
- Het opleggen van doelen aan acteurs door de regisseur. Parameters: acteur en doel.
- Het uitvoeren van een actie door een acteur. Parameters: acteur en actie.
- Het doorsturen van zinnen van de NarratorAgent naar de PresentationAgent. De zinnen zijn van een natuurlijke taal.

In figuur 3 is te zien welke informatie er tussen welke agents uitgewisseld moet worden.



Figuur 3: Informatie-uitwisseling over het verhaal tussen de agents

1.2.2Organisatorische informatie-uitwisseling

Omdat agents in dit multi-agent systeem elk hun eigen geheugen bezitten, en in principe autonoom kunnen beslissen over te nemen acties, moeten deze agents *weten* van elkaar dat ze er zijn. Zo moet bijvoorbeeld ActorAgent "Plop" op de hoogte gebracht worden van het feit dat zijn regisseur DirectorAgent "director" is en zijn wereld WorldAgent "StoryWorld". Dit soort informatie is dus puur van organisatorische aard, en voegt niet direct iets toe aan de generatie van een verhaal.

De volgende tabel laat zien welke informatie beschikbaar moet zijn voor welke agent:

agent	*knowledge about*
DirectorAgent	WorldAgent, ActorAgent
ActorAgent	DirectorAgent, WorldAgent
NarratorAgent	WorldAgent

Tabel 1: Kennis over andere agents

De WorldAgent moet van alle agents op de hoogte worden gehouden.

1.2.3De StoryAgentOntology

Om agents met elkaar te kunnen laten communiceren moet er een bepaalde overeenstemming zijn over het medium, de syntaxis, maar ook de semantiek. De Virtuele Verhalenverteller is geïmplementeerd in het agentraamwerk JADE [JADEWEB]. Zoals reeds uitgelegd in [RENS2004] levert JADE een raamwerk voor

communicatie, en hoeven we ons slechts bezig te houden met het specificeren van een ontologie.

In de Virtuele Verhalenverteller wordt gebruik gemaakt van twee ontologieën:

- de StoryAgentOntology: voor het uitwisselen van boodschappen,
- de StoryWorldOntology: voor het beschrijven van de wereld en acties.

Dit verslag richt zich in hoofdstuk 2 op de *StoryWorldOntology*. Als het gaat om het versturen van informatie over de toestand van de wereld, wordt, zoals in paragraaf 1.2.1 reeds genoemd, binnen de *StoryAgentOntology* gebruik gemaakt van zinnen uit de *StoryWorldOntology*. Bijvoorbeeld de WorldAgent stuurt de ActorAgent informatie over de wereld toe als een instantie van een *World Info* predikaat van de *StoryAgentOntology*. Dit *World Info* predikaat zal echter een *slot* (plek voor gegevens) hebben die informatie bevat gesteld in de *StoryWorldOntology*.

Het plaatje hiernaast geeft de hiërarchie van de *StoryAgentOntology* weer zoals die in de Protégé-2000[1] tool getoond wordt. In het werk van Sander Rensen en Sander Faas is voor het ontwerp van deze ontologie ook gebruik gemaakt van Protégé2000.

Figuur 4: structuur StoryAgentOntology

De keuze voor concepten en predikaten in deze ontologie is gebaseerd op figuur 3 en tabel 1. Verder forceert JADE de opdeling van een ontologie in de categorieën *Concept* en *Predicate.* Conform de tutorial voor het maken van JADE ontologieën [GIOV2002], beschrijven *Predicates* ware feiten over de wereld, en worden vooral gebruikt als *content* van INFORM berichten, *AgentActions* zijn bedoelt om agentspecifieke acties te beschrijven. *AgentActions* hebben vooral betekenis als communicatieve actie, dat wil zeggen dat ze beschrijven *wat* deze agents *doen*, binnen een bepaalde context van bijvoorbeeld de rol die deze agents vervullen.

De *StoryAgentOntology* die hier beschreven staat verschilt in een aantal opzichten van die van Sander Rensen. Deze verschillen zijn voor een groot deel een kwestie van naamgeving en verschillen in implementatie.

1 http://protege.stanford.edu

1.3Veranderingen ten aanzien van de flexibiliteit

Zowel in de Virtuele Verhalenverteller van Sander Faas [FAAS2002] als in die van Sander Rensen [RENS2004] is de mate van flexibiliteit niet hoog. In het algemeen heeft dit tot gevolg dat het lastig is in deze versies om uitbreidingen van het systeem te maken, en met name uitbreidingen voor een complexer verhaal met meerdere karakters.

1.3.1Problemen met betrekking tot de StoryWorldOntology

Bij de implementatie van Sander Rensen is de koppeling tussen de gebruikte ontologie voor de beschrijving van de wereld en de rest van het systeem erg hoog, dat wil zeggen veel Jess-scripts en Java-classes zijn *direct* afhankelijk van de *StoryWorldOntology*. Bij een inventarisatie van de bron-code is gebleken dat een groot aantal onderdelen van het systeem *direct* afhankelijk zijn van de *StoryWorldOntology*, dat wil zeggen dat deze onderdelen bij een geringe aanpassing aan de ontologie ook aangepast moeten worden, als er van deze nieuwe concepten in de ontologie gebruik gemaakt dient te worden.

Een aantal afhankelijke Java-classes:

- *vs.nlg.Lexicon*
- *vs.nlg.NLGenerator*
- *vs.narrator.Narrator*
- *vs.actor.BeliefsModule*
- *vs.actor.ActionSelectionModule*
- *vs.actor.EventAppraiser*
- *vs.actor.EmotionalModel*
- *vs.director.WorldInformation*
- *vs.director.PermissionModule*

Verder zijn alle Jess-scripts afhankelijk van zowel *StoryAgentOntology* als de *StoryWorldOntology*.

Om even een voorbeeld te geven: stel de wereld bevat niet alleen een kasteel, maar ook een boerderij, en de *StoryWorldOntology* moet hiermee uitgebreid worden. Dan is het uitbreiden van de ontologie met behulp van Protégé2000en de BeanGenerator[2] geen probleem. Echter de ActionSelectionModule moet ook aangepast worden zodat karakters ook de mogelijkheid hebben om in de boerderij te verschuilen. Daarnaast moeten de onderdelen voor de generatie van natuurlijke taal ook uitgebreid worden.

1.3.2Oorzaak sterke koppeling tussen ontologie en rest van het systeem

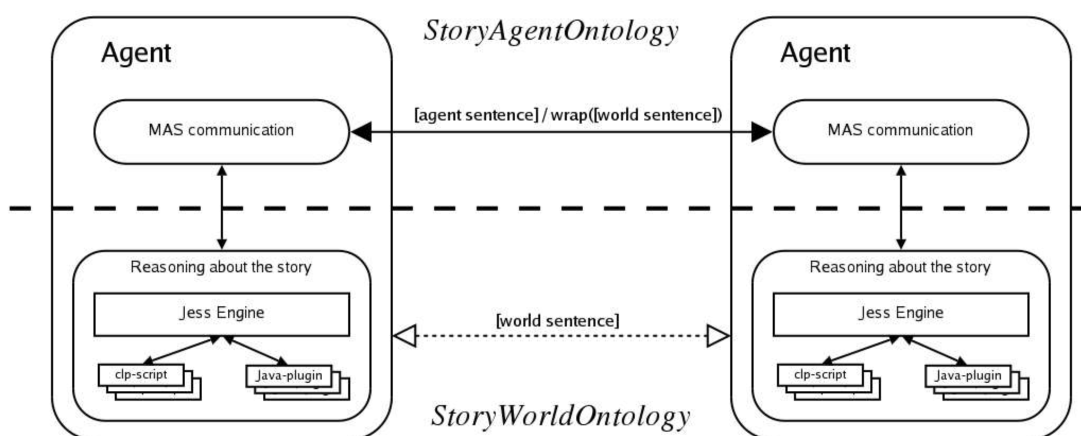
De StoryWorldOntology en de daar bijbehorende *beans* zijn bijna overal in het systeem letterlijk terug te vinden. Er is voor gekozen de agents Virtuele Verhalenverteller twee verschillende ontologieën mee te geven om zo onderscheid te kunnen maken tussen begrippen in de verhalenwereld, waar *karakters* interacteren met de wereld en elkaar, en communicatie benodigd voor het verdelen van de taken over meerdere *software agents*. Idealiter zou de afhandeling van acties/taken binnen deze twee verschillende 'werelden' door verschillende systeemonderdelen moeten worden afgehandeld. Zoals eerder in dit hoofdstuk al opgemerkt blijft er een verbinding bestaan tussen deze twee werelden

2 http://gaper.swi.psy.uva.nl/beangenerator/content/main.php

doordat zinnen in de *StoryAgentOntology* zinnen van de *StoryWorldOntology* kunnen bevatten. Dit hoeft geen probleem te zijn als *StoryWorldOntology*-zinnen door de *StoryAgentOntology* zonder verdere inhoudelijke analyse kunnen worden gebruikt. Dit is voor het grootste gedeelte ook het geval in Sander Rensens implementatie. Echter wordt er bij deze implementatie gedeeltelijk gebruik gemaakt van Jess-scripts, die globaal onafhankelijk kunnen opereren van het agent systeem, en voor een ander gedeelte wordt er gebruik gemaakt van Java classes die direct zijn gekoppeld aan én een JADE agent, én de *StoryWorldOntology*.

Zo algemeen mogelijk gesteld zou de flexibiliteit van het systeem toenemen door zo min mogelijk koppeling te bewerkstelligen tussen het agentraamwerk met de onderlinge agent-communicatie en de zaken die zich afspelen op het niveau van de verhalenwereld.

1.3.3Scheiding op basis van ontologieën



Figuur 5: Onderscheid agent- en worldontology

Het diagram hierboven laat een deel van een mogelijke interne architectuur voor een agent in de Virtuele Verhalenverteller zien. De dikke stippellijn geeft de scheiding aan die in een agent aanwezig zou moeten zijn tussen het gedeelte dat verantwoordelijk is voor het redeneren over de verhalenwereld (*storyworld*), en een gedeelte dat verantwoordelijk is voor de communicatie tussen de verschillende agents. De gevulde pijl geeft aan dat er berichten worden verstuurd tussen agents, en dat dit afgehandeld wordt door het communicatie-gedeelte. Alle zinnen die worden overgestuurd zijn van de StoryAgentOntology. Binnen deze zinnen kunnen StoryWorldOntology zinnen *gewrapt* zitten. De gestippelde pijl geeft aan dat er tussen de storyworld-lagen (onderste laag in het plaatje) van de verschillende agents feitelijk (maar indirect) overdracht plaats vindt van *world sentences* (zinnen over de verhalenwereld). Het redeneren wordt in de Virtuele Verhalenverteller gedaan met behulp van Jess[3]. Jess kan scripts uitvoeren en eventueel zouden er Java klassen kunnen worden aangeroepen.

De verwachting is dat de meeste veranderingen in het systeem, voor toekomstige uitbreidingen, zullen plaatsvinden op de storyworld-laag, omdat de mogelijkheid om verhaalgrammatica's, karakters, doelen et cetera te veranderen inherent is aan de eis van het systeem om willekeurige (nieuwe) verhalen te kunnen genereren. Het is daarom handig om onderdelen van het systeem uit de storyworld-laag zoveel mogelijk door scripts uit te laten voeren, zodat voor kleine aanpassingen aan verhaalelementen niet het hele systeem opnieuw hoeft te worden gecompileerd. Ook kan gedacht worden aan

3 http://herzberg.ca.sandia.gov/jess/

Java-plugin modules, wat door de mogelijkheid van Java om dynamisch nieuwe klassen te laden ook opnieuw compileren vermijdt.

interface

Om hierboven genoemde mogelijkheden te kunnen realiseren moet er een *interface* worden gespecificeerd. Als voorbeeld van zo'n interface de volgende ActorAgent-functies die door de storyagent-laag (bovenste gedeelte) worden aangeroepen en kunnen worden geïmplementeerd in Jess (onderste gedeelte):

- *set-percept (?percept)* : ontvangt, voor de acteur beschikbare, informatie over de wereld. Het argument *?percept* is een string die een zin bevat in *StoryWorldOntology*.
- *set-goal (?actor ?action ?arg)* : ontvangt een nieuw doel voor het karakter om na te streven. Het argument *?action* is een voor de *StoryWorldOntology* betekenisvolle term die een uit te voeren actie aanduidt, met eventueel een extra argument *?arg*.
- *got-permission (?actor ?action ?arg)* : geeft deze acteur door dat de actie *?action* met argument *?arg* mag worden uitgevoerd.

De storyworld-laag kan gebruik maken van de reeds aanwezige Jess-userfunction *send-message ?performative ?destination ?object ?ontology.*

Waarbij voor *?performative* het meest INFORM of REQUEST zal worden gebruikt, om aan te geven dat het respectievelijk gaat om een bericht die bedoelt is voor informatie overdracht en een bericht dat bedoelt is om informatie te vergaren.

Bij *?destination* kan het AID (*Agent IDentification*) van de ontvangende agent worden ingevuld.

Voor *?ontology* wordt er altijd de StoryAgentOntology gebruikt, het gaat immers om berichten gesteld in deze ontologie.

Voor *?object* wordt een instantie van een subklasse van *Predicate* of *AgentAction* ingevuld uit de *StoryAgentOntology* (zie paragraaf 1.2.3), dit vormt de inhoud van het bericht.

samengevat

Deze paragraaf nog even samengevat; de vooronderstelling van de hier beschreven scheiding op basis van ontologieën is dat dit de flexibiliteit van het systeem wordt verhoogd door:

- de mogelijkheid om de *StoryWorldOntology* te vervangen of uit te breiden zonder gevolgen voor agent communicatie die gesteld is in de *StoryAgentOntology*,
- duidelijker beeld van verantwoordelijkheden van verschillende onderdelen binnen een agent door de lage koppeling die tussen deze onderdelen bestaat.

2.SUMO als basis voor de verhalenwereld

Dit hoofdstuk gaat over de mogelijkheden om de Suggested Upper Merged Ontology (SUMO) als basis voor een Jess knowledgebase en rulebase in het multiagent systeem van de Virtuele Verhalenverteller te gebruiken.

De agents zoals die gebruikt worden in de Virtuele Verhalenverteller, maken beslissingen om *acties* te ondernemen op basis van een kennisbank (*knowledge base*). Een *knowledge-based* agent (*KB-agent*) kan met behulp van gegeven kennis van de wereld (omgeving) *redeneren* om nieuwe feiten over de wereld te weten te komen en op basis daarvan een juiste actie te kiezen. De kennisbank van een *KB-agent* bevat in eerste instantie vaak al enige kennis die als basis voor kennis over de wereld kan dienen, hier wordt naar gerefereerd als initiële kennis, achtergrond kennis/informatie, of *background knowledge* [RUS2003].

2.1Waarom?

Door de agents van de Virtuele Verhalenverteller bepaalde *background knowledge* te geven hopen we dat het verhaal realistischer zal overkomen. Een agent zonder *background knowledge* weet nog helemaal niets over zijn omgeving, en zal dus om te kunnen redeneren over zijn omgeving helemaal 'from scratch' een model van zijn omgeving moeten construeren op basis van onderzoek. Voor de toepassing van zo'n agent in de Virtuele Verhalenverteller is dit niet handig, aangezien er van karakters in het verhaal wordt verwacht dat ze initiële kennis hebben die er voor het karakter toe doet. Om een voorbeeld te geven: de prinses in de verhalen van Sander Rensen *weet* bijvoorbeeld *van tevoren* dat een zwaard een wapen is, en dat een wapen gebruikt kan worden om een ander karakter te slaan of te doden.

De verwachting is dat kennis over de wereld zoals mensen die interpreteren bijdraagt aan een meer menselijke uitleg (vertelling) van het verhaal, en geeft de acteurs meer menselijke redenatie mogelijkheden. In de Virtuele Verhalenverteller is ervoor gekozen om deze kennis in de vorm van een ontologie aan de agents die de karakters moeten vertegenwoordigen mee te geven.

2.2Waarom SUMO?

Volgens [USERTEK] kan ontologie worden gedefinieerd als:

> *a systematic formalization of concepts, definitions, relationships, and rules that captures the semantic content of a domain in a machine-readable format.*

Daarnaast worden een in dit document een aantal redenen genoemd om in computer applicaties *upper-ontologies* te gebruiken, waarbij de toevoeging *upper* op de bovenste laag slaat waar deze ontologie zich manifesteert, dat wil zeggen dat een *upper-ontology* als basis voor onderliggende ontologieën gebruikt kan worden, wat ook blijkt uit het volgende uit [USERTEK]. Een *upper-ontology*:

- geeft een basis voor meer specialistische ontologieën;
- kan als raamwerk dienen voor integratie van domein-geörienteerde ontologieën, en/of;
- als hulp bij het vertalen van ontologieën die over hetzelfde domein gaan, maar een ander vocabulaire hanteren.

Voor de Virtuele Verhalenverteller is vooral het eerste punt belangrijk. Daarnaast zijn er nog een aantal redenen te noemen voor de Virtuele Verhalenverteller om een *upper-ontology* te gebruiken:

1. concepten die in een dergelijke ontologie worden gedefinieerd zijn algemeen van aard, en hebben daardoor een sterke binding met een menselijke 'voorstelling van zaken' (nadere toelichting hieronder),
2. als er overeenstemming is over de te gebruiken *upper-ontology*, kan er een domein-specifieke ontologie (domein van verhalenverteller, of sprookjes) worden ontworpen die deze *upper-ontology* als basis gebruikt, en die zelf weer als basis kan dienen voor verschillende soorten verhalen.

Het eerste punt kan nog nader worden toegelicht. De karakter-gebaseerde benadering voor het genereren van verhalen heeft als voordeel dat de geloofwaardigheid van de karakters in het verhaal hoog is. Dit werkt alleen als het karakter *background-knowledge* bezit die geloofwaardig overkomt, en als er in het verhaal alleen realistische acties zijn toegestaan. Dus de *background-knowledge* van de agents moet algemene kennis over de wereld bevatten, dat wil zeggen dat de kennis van de agents tot op zekere hoogte overeen moet komen met de kennis over de echte wereld zoals wij mensen die ook hebben. Bijvoorbeeld weten mensen in het algemeen dat een appel fruit is, dat je dat kan eten, dat fruit van een plant komt, et cetera.

SUMO

Suggested Upper Merged Ontology [SUMO] is een ontologie die wordt ontwikkeld door de IEEE Standard Upper Ontology Working Group [SUOWG]. In de rest van dit hoofdstuk zal de aandacht uitgaan naar SUMO, en op welke manier SUMO eventueel gebruikt kan worden als *upper-ontology* in de Virtuele Vehalenverteller. De keuze voor SUMO is gemotiveerd door de volgende punten:

- IEEE standaard (vrij beschikbaar),
- gedefinieerd in een tevens vrij beschikbare variant van KIF genaamd SUO-KIF [SUOKIF],
- algemeen, maar niet te uitgebreid.
- Er bestaat een mapping van WordNet naar SUMO. Voor de implementatie van de NarratorAgent die de generatie van natuurlijke taal uitvoert kan dit handig zijn.

2.3Hoe?

De agents van de Virtuele Verhalenverteller maken gebruik van Jess om een planning te maken en eventuele andere redeneer acties uit te voeren. Onder andere in het werk van Sander Faas [FAAS2002] wordt besproken hoe de keuze voor Jess gemotiveerd kan worden, maar hier staan nog een drietal punten die vooral van belang zijn gebleken bij de implementatie van het prototype (zie hoofdstuk 3).

- mogelijkheid tot gebruik van zowel forward-chaining als backward-chaining,
- mogelijkheid tot definiëren van *userfunctions* om zo Jess naar behoefte uit te breiden, en
- implementatie in Java, met mogelijkheid om Java methoden en klassen aan te roepen en Java-objecten te manipuleren.

SUMO en Jess

SUMO bevat feiten en axioma's (of regels), de feiten geven zo minimaal mogelijk weer

welke *relaties* er tussen verschillende *entiteiten* gelden, de axioma's kunnen agents de mogelijkheid geven om nieuwe feiten af te leiden uit reeds afgeleide of reeds aanwezige feiten. Daarnaast geven de axioma's de uiteindelijke semantiek weer van concepten.

Ook Jess kan feiten (*facts*) en regels (*rules*) bevatten, het verschil tussen de taal waarin SUMO is gedefinieerd (SUO-KIF) en Jess zit in het feit dat Jess een functionele taal is, waarin bepaalde data-manipulaties (berekeningen) kunnen worden uitgevoerd, en SUO-KIF een declaratieve taal is waar niet programma's mee kunnen worden uitgevoerd, maar slechts relaties tussen concepten kunnen worden beschreven op een abstract niveau (eerste orde logica).

Om de structuur en kennis die SUMO definieert te gebruiken in de Virtuele Verhalenverteller moet er een dus *vertaling* worden uitgevoerd van SUMO naar een bruikbare Jess kennisbank.

2.4SUMO naar Jess vertalen

Zoals in de vorige paragraaf al genoemd, zijn SUO-KIF en Jess verschillend van elkaar als we kijken naar de functie waarvoor deze taal is ontwikkeld, descriptief (vrije interpretatie) tegen operationeel (vaste interpretatie). Toch zijn er behoorlijk wat overeenkomsten tussen Jess en SUO-KIF te noemen:

- de algemene syntaxis is nagenoeg gelijk, in beide talen hebben zinnen een structuur waarin met gewone haken wordt aangegeven welke gedeelten bij elkaar horen. Onder andere de tokens =>, *exists*, *and* en *or* zijn hetzelfde in beide talen. Gewone variabelen worden in beide talen vooraf gegaan door een *?*. Strings, nummers en woorden worden bijna op dezelfde manier gerepresenteerd.
- de representatie van een waar feit is gelijk aan elkaar, bijvoorbeeld de SUO-KIF zin
  (subclass Dwarf Human)
  wordt in het werkgeheugen van Jess op dezelfde manier gerepresenteerd (met uitzondering van module-prefix).
- axioma's in SUO-KIF kunnen in een redelijk aantal gevallen naar regels (*defrule*s) in Jess vertaald worden.

Er zijn ook verschillen tussen beide talen, maar net als bij de overeenkomsten, zal hier slechts een algemeen overzicht genoemd worden, toegespitst op de toepassing binnen de Virtuele Verhalenverteller:

- SUO-KIF kent wel functies, maar deze zijn abstract gedefinieerd, en hoeven niet persé concreet te worden gemaakt. In Jess moeten functies altijd gedefinieerd zijn.
- Jess maakt onderscheid tussen werkgeheugen (gerepresenteerd door een Rete netwerk, zie [FRIE2003]), en regels en functies et cetera, die dit geheugen na een opdracht van de gebruiker kunnen manipuleren. In SUO-KIF is er alleen maar 'geheugen' en zijn axioma's dus ook onderdeel van het geheugen. Dit betekent dat bijvoorbeeld de SUO-KIF zin

```
(=>
        (parent ?CHILD ?PARENT)
        (or
                (mother ?CHILD ?PARENT)
                (father ?CHILD ?PARENT)))
```

Niet gerepresenteerd kan worden door de volgende *defrule* in Jess

```
(derule parentlogic
        (parent ?CHILD ?PARENT)
        =>
        (assert (or
                (mother ?CHILD ?PARENT)
                (father ?CHILD ?PARENT))))
```

Waarbij vermeldt dat de functie *assert* nieuwe feiten in het werkgeheugen van Jess zet. In het geval er ergens een kind van een ouder is, *weet* Jess niet of deze ouder dan moeder is of vader. Een dergelijk feit (moeder of vader) kan niet in het werkgeheugen van Jess worden opgeslagen, alleen ware (zekere) feiten kunnen worden opgeslagen. Zinnen als argument van een *assert* kunnen dan ook geen connectieven als *and* of *or* bevatten, evenals *exists*, of *forall*. Dit kan ook niet simpel opgelost worden.

- Gerelateerd hieraan is het probleem dat in Jess een keuze moet worden gemaakt tussen het verwijderen van bepaalde feiten of het toevoegen ervan indien de feiten niet overeenkomen met de regels. Bijvoorbeeld de SUO-KIF zin uit de SUMO

```
(=>
        (immediateInstance ?ENTITY ?CLASS)
        (not (exists (?SUBCLASS)
                (and
                        (subclass ?SUBCLASS ?CLASS)
                        (instance ?ENTITY ?SUBCLASS)))))
```

Zou vertaald kunnen worden zodat indien het existsgedeelte aan rechterkant van deze regel tòch geldt (dus er bestaat dan wel een dergelijk subklasse en instantie), dat dan de linkerkant (dat als feit aanwezig moet zijn in het geheugen voor de toepassing van deze regel) ingetrokken moet worden. In Jess wordt dat dan

```
(defrule iminst
        ?im <- (immediateInstance ?ENTITY ?CLASS)
        (exists
                (and
                        (subclass ?SUBCLASS ?CLASS)
                        (instance ?ENTITY ?SUBCLASS)))))
        =>
        (retract ?im))
```

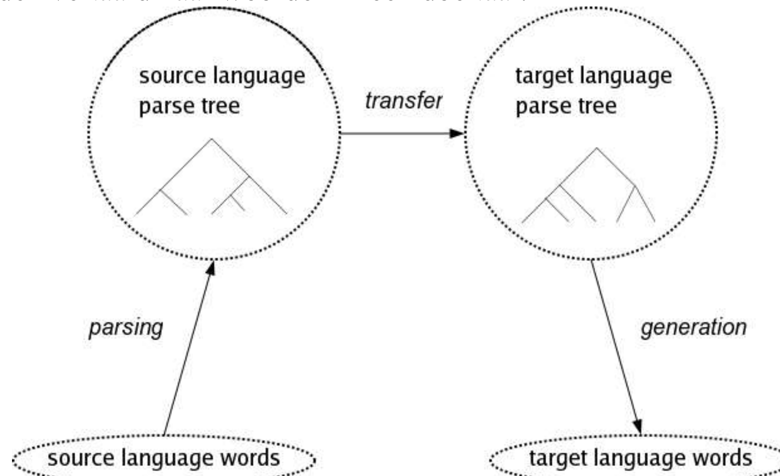
Deze *defrule* is er slechts om de integriteit te behouden van de kennisbank, maar voegt niets toe aan de kennisbank. Als een agent deze regel wil gebruiken, wordt er verondersteld dat het voor kan komen dat de relatie immediateInstance **wel** geldt voor bepaalde entiteiten, maar dat dit **niet** conform de regels is. Ofwel; er wordt vanuit gegaan dat onware feiten in het systeem kunnen ontstaan, die vervolgens weer verwijderd dienen te worden. Dit lijkt vrij zinloos, en is waarschijnlijk ook niet de bedoeling van de oorspronkelijke SUMO regel.
Een andere manier om deze SUMO regel te vertalen naar Jess zou kunnen zijn dat de immediatieInstancerelatie, die geldt tussen de twee entiteiten, afgeleid kan worden uit het niet bestaan van èn de gegeven subklasserelatie, èn de instantierelatie. Het probleem hierbij is echter dat er oneindig veel van deze relaties niet bestaan. Dit kan dus niet op deze manier in Jess worden gezet vanwege problemen met het binden van variabelen in een existszin. Meer informatie hierover is te vinden in de Jess Handleiding [FRIE2003].

- Het binden van variabelen werkt iets anders.

Vanwege de grote overeenkomst in syntaxis tussen Jess en SUO-KIF lijkt een vertaling vrij simpel, en hoeft er slechts rekening te worden gehouden met uitzonderingen. In de rest van de paragraaf wordt kort ingegaan op het vertaalproces.

2.4.1De vertaler

In het diagram hieronder is aangegeven hoe de vertaling van woorden in een brontaal kunnen worden vertaald naar woorden in een doeltaal.



Figuur 6: Vertaalproces (overgenomen uit [JURA2000])

Er worden drie processen getoond die hier kort worden toegelicht.

parsing

Een zin kan worden uitgelezen en via een grammatica kan er een zogenaamde *parse tree* worden opgebouwd. De grammatica geeft de structuur aan waarin de woorden met elkaar verbonden zijn. Voor het bouwen van zo'n *parser* kunnen *parser-generators* worden gebruikt. Voor het *parsen* van SUO-KIF zinnen is bij het prototype uitgegaan van de grammatica zoals gegeven in bijlage A.

transfer

Omdat globaal gezien de syntaxis van SUO-KIF en Jess goed overeenkomen is deze stap grotendeels ontdaan van het omzetten van SUO-KIF *tokens* naar Jess *tokens*. De transfer bestaat hier vooral uit het omzetten van de structuur van de *parse tree*. Dit kan bijvoorbeeld gedaan worden met structuurschema's (*subtrees* van de complete *parse tree*) die een SUO-KIF structuur afbeelden (*mappen*) op een Jess structuur.

generation

De opgebouwde Jess *parse tree* wordt in deze stap omgezet naar daadwerkelijke zinnen. De uitvoer na deze stap zou geschikt moeten zijn om als script in Jess ingeladen te kunnen worden.

2.5Samenstellen van een '*upper ontology*' als deelverzameling van SUMO.

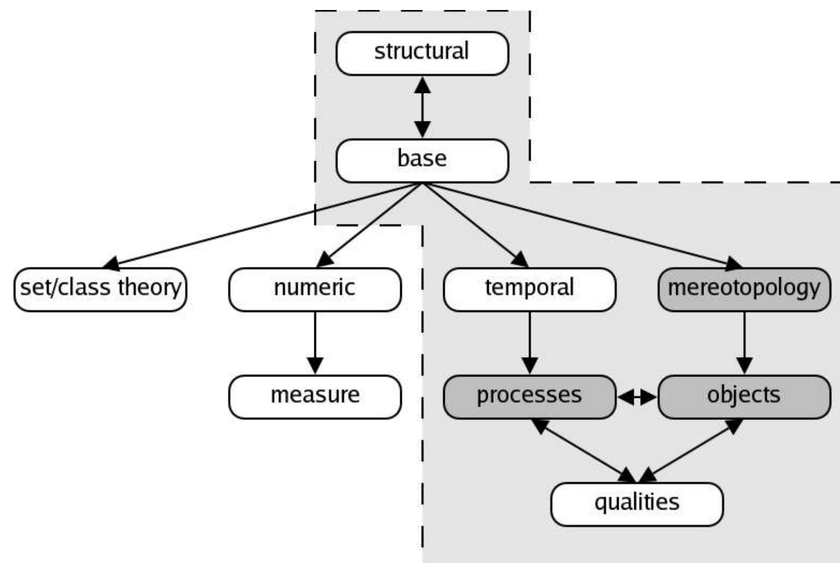
Bij de implementatie van een prototype dat gebruik maakt van een vertaling van SUMO naar een Jess script, is gebleken dat indien er *forward-chaining* toegepast wordt en er

teveel SUMO regels zijn vertaald, er geheugenproblemen optreden. Deze paragraaf gaat over de inperking van de totale SUMO, naar een voldoende deelverzameling.

Om de *overhead* bij het gebruik van SUMO in de Virtuele Verhalenverteller zo veel mogelijk in te perken, vindt er een selectie plaats op verschillende niveaus. De algemene richtlijn voor de selectie van bepaalde delen van SUMO is dat er alleen delen worden weggelaten die niet nodig zijn, en dat het deel wat overblijft voldoende is om als basis te dienen voor de generatie van verhalen.

2.5.1 Selectie van modules uit SUMO

SUMO is opgebouwd uit modules. De modules hebben bepaalde afhankelijkheden van elkaar die in het onderstaande figuur staan aangegeven.



Figuur 7: SUMO module hiërarchie.

De pijlen geven een afhankelijkheid aan van de ene module van de andere module. Het gestippeld omlijnde gedeelte is de selectie die is gemaakt, waarbij uitgegaan is van de donker gevulde modules.

De ***mereotopology*** module is gebaseerd op een formele theorie die de relatie tussen delen en gehelen beschrijft. Volgens de makers van SUMO is deze module vooral gebaseerd op Barry Smiths theorie[4] over dit onderwerp. Voor de Virtuele Verhalenverteller is deze module nodig gebleken vanwege de relatie part die aangeeft dat een bepaald object onderdeel is van een ander object. Bijvoorbeeld in het verhaal van Sander Faas zou de appel in de kamer in het huis zich kunnen bevinden. Hier moet in de kennisbank het feit dat de kamer onderdeel is van het huis aanwezig zijn, om te kunnen beredeneren dat Plop naar het huis moet lopen om de appel te kunnen pakken.

De ***processes*** module beschrijft processen zoals *Walking*.

De ***objects*** module beschrijft objecten en met name subklasserelaties tussen objecten. Zo is bijvoorbeeld *Apple* een subklasse van *FruitOrVegetable* wat weer een subklasse is van *Food*, waardoor Plop weet (in combinatie met de juiste regels) dat hij een appel kan eten om zijn honger te stillen.

De overige modules die in de figuur binnen de stippellijn zich bevinden zijn afhankelijk van *mereotopology*, *processes* en *objects*, waardoor deze modules ook meegenomen

4 http://ontology.buffalo.edu/smith/articles/mereotopology.htm

dienen te worden. De modules die er buiten vallen zijn meer van theoretische aard en waarschijnlijk nooit nodig voor de Verhalenverteller[5].

2.5.2 Selectie bij parsing

Bepaalde feiten en axioma's in SUMO staan het toe om een relatie aan te geven tussen een entiteit (wortel van de SUMO structuur) en een predikaat, dit is ook toegelaten volgens de specificatie van de SUO-KIF grammatica. De mate waarin deze geneste predikaten worden toegepast in SUMO beperkt zich vooral tot de relaties *holdsDuring*, *hasPurpose* en *hasPurposeForAgent*. Jess heeft niet een directe mogelijkheid om deze constructie te ondersteunen, en er is voor gekozen om de regels waarin deze constructie wordt gebruikt handmatig uit te commentariëren. De geïmplementeerde *parser* is dus strikter dan de gespecificeerde grammatica.

2.5.3 Selectie na parsing

Nadat de bovengenoemde selecties zijn uitgevoerd is er een bestand in (iets striktere) SUO-KIF die door de *parser* kan worden geaccepteerd en een *parse tree* kan vormen. Over deze *parse tree* kunnen nog filters gehaald worden die sommige delen van te voren alvast omvormen, of zoals ook hier toegepast bepaalde delen weggooien. Voor het gebruik van SUMO in de Virtuele Verhalenverteller zijn twee filters gebruikt; de *LiteralFilter* en de *RetractFilter* die verderop worden uitgelegd.

Daarnaast is er voor sommige regels geen *schemamapping* gedefinieerd, dat wil zeggen dat voor sommige regels simpelweg niet een vertaling is gedefinieerd, bijvoorbeeld omdat er een *or* aan de rechterkant van een axioma voorkomt (zie ook voorbeeld in de inleiding van 2.4).

LiteralFilter

Dit filter gooit alle *nodes* in de *tree* weg die entiteiten die **niet** in een speciaal bestand zijn genoemd. Dit speciaal bestand bevat een verzameling van SUMO termen (*literals*) die handmatig is samengesteld, en waarin alleen termen staan die er voor de Verhalenverteller toe doen.

Met behulp van een van de online KIF-browsers[6] is het mogelijk om termen te selecteren door *bottom-up* de SUMO structuur door te lopen (*browsen*), en telkens de termen die *direct* met de vorige term te maken hebben te selecteren. Met direct wordt bedoelt dat de vorige term afhankelijk is voor het bestaan van de huidige term, dat wil zeggen dat de huidige term de link vormt tussen de vorige term en de wortel. Deze selectie begint dus met op stellen van een lijst van *bodemtermen* (die van belang zijn in verhalen) waar vanuit de boom *bottom-up* kan worden doorgelopen.

RetractFilter

Dit filter voegt indien nodig een zogenaamd *fact-id* (zie Jess Handleiding) toe aan een feit dat moet worden teruggetrokken uit het geheugen door de *retract* functie. Deze functie accepteert alleen *fact-ids* en geen *patterns*.

5 Meer informatie over de modules van SUMO is te vinden op [SUMO] en in het SUMO document.
6 http://virtual.cvut.cz/kifb/en/ of http://berkelium.teknowledge.com:8080/sigma

2.6Samenstellen van een '*application specific ontology*'

Omdat SUMO bedoelt is als *upper-ontology* die als basis gebruikt kan worden voor een ontologie in een specifieke applicatie, moet er, indien nodig, een applicatie specifieke ontologie opgesteld worden. Een applicatie specifieke ontologie voegt extra concepten en regels toe aan de ontologie die nog niet in SUMO zijn gedefinieerd, maar wel nodig zijn in de applicatie. Om van de bestaande regels en concepten in SUMO gebruik te maken, is het handig om nieuwe concepten bijvoorbeeld een *instance* of *subclass* te laten zijn wat correspondeert met respectievelijk een nieuw individu of een nieuwe klasse [USERTEK].

Voor de Virtuele Verhalenverteller is een applicatie specifieke ontologie gemaakt die gebaseerd is op SUMO.

In de rest van dit hoofdstuk zal aan de hand van een voorbeeld de verdere uitwerking van het gebruik van SUMO in Jess worden getoond. Als voorbeeld wordt het verhaal van Sander Faas [FAAS2002] genomen, en gekeken in hoeverre een gelijksoortig verhaal gemaakt kan worden. De agents van het systeem worden uitgerust met initiële kennis over de wereld in de vorm van SUMO als *upper-ontology* in combinatie met een applicatie specifieke ontologie.

De volgende concepten met hun relatie tot bestaande SUMO concepten zijn aanwezig in de applicatie specifieke ontologie (in dit geval de *StoryWorldOntology*):

(subclass Dwarf Human)
(subclass Hungry Perception)
(subclass HungerAttribute BiologicalAttribute)
(subclass HungerAttribute PerceptualAttribute)
(subclass HungerAttribute ValuedAttribute)
(instance attributeValue BinaryPredicate)

Met behulp van de WordNet[7] functie in de KIFbrowser kunnen nieuwe klassen, individuen, attributen en relaties worden geplaatst binnen de SUMO. In dit voorbeeld is de relatie *attributeValue* toegevoegd als een soort uitbreiding op de bestaande notie van *attribute* in SUMO. Meer uitleg over *attributeValue* en *HungerAttribute* volgt in paragraaf 2.8.

2.7Voorbeeld uit het verhaal van Sander Faas

In deze paragraaf wordt ingegaan op een concreet voorbeeld om te laten zien hoe er in Jess gebruik kan worden gemaakt van SUMO. Sander Faas [FAAS2002] heeft voor de eerste versie van de Virtuele Verhalenverteller als basis voor een verhaal kabouter Plop genomen die hongerig is en voedsel tot zich moet nemen om deze honger te stillen. Deze basis zal hier ook gebruikt worden, omdat de complexiteit niet erg hoog is en daardoor goed bekeken kan worden hoe een proces van **redeneren** en **actie ondernemen** zich kan voltrekken.

2.7.1Plops motivatie

Voordat Plop iets in het verhaal gaat doen kan er een reden op worden gegeven *waarom* hij iets gaat doen. Voor de toehoorder van het verhaal is het ondernemen van actie door een van de karakters in het verhaal *gegrond* als daar bijvoorbeeld een rationele of emotionele reden achter zit. Voor het verhaal van Plop zou kunnen gelden dat de reden

7 http://www.cogsci.princeton.edu/~wn/

om op zoek te gaan naar eten gegeven wordt door het feit dat Plop honger heeft. Er wordt hier mee een volgorde vastgelegd;

1. **Plop heeft honger, en daardoor gaat Plop eten.**

De volgorde hoeft niet eenduidig te zijn. Vanuit de toehoorder zou namelijk ook beredeneerd kunnen worden dat Plop honger heeft juist omdat hij op zoek is naar eten. Dus:

2. **Plop gaat eten, dus Plop zal honger hebben.**

Ook zou Plop zich *bewust* kunnen worden van honger nadat hij merkt dat hij op zoek is naar eten.

3. **Plop gaat op zoek naar eten, dan realiseert Plop zich dat hij honger moet hebben.**

Op deze manier zijn vast nog meer interpretaties van het verband tussen de gewaarwording 'honger' en de actie 'eten zoeken' mogelijk. Bij de interpretatie van een verhaal wordt dus min of meer een volgorde van acties en gewaarwordingen verwacht wat uitgelegd moeten worden.

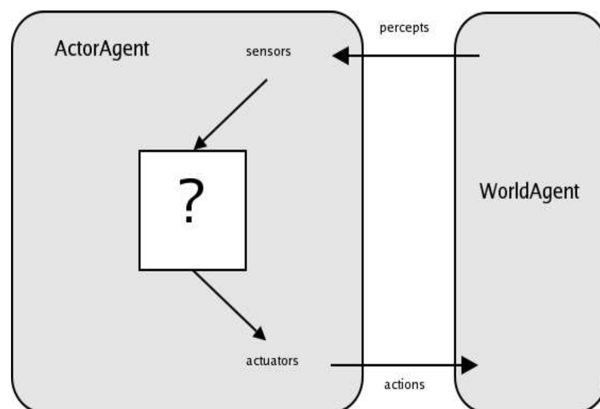
Het uitleggen van het verband tussen een gewaarwording en een actie is bij de Virtuele Verhalenverteller in principe een verantwoordelijkheid van de NarratorAgent, omdat deze de interpretatie verzorgt van wat er in de verhalenwereld zich afspeelt. In dit project ligt de nadruk echter niet op de NarratorAgent, en daarnaast zal het initiëren van een actie of een gewaarwording toch op een of andere manier gemodelleerd moeten worden in de ActorAgent of de WorldAgent.

sense, think, act

In agentsystemen wordt vaak uit gegaan van het volgende patroon, dat min of meer overeenkomt met hierboven genoemde volgorde bij punt 1:

sense → think → act.

Vanuit een ActorAgent gezien vindt er dus een actie plaats op basis van een eerder ontvangen gewaarwording (*perception*). In [RUS2003] wordt dit idee als volgt in een plaatje weergegeven, waarbij voor Agent ActorAgent en voor Environment WorldAgent is ingevuld:



Figuur 8: gewaarworden en actie nemen (aangepast overgenomen uit [RUS2003]

Voor de Virtuele Verhalenverteller betekent dit dat de WorldAgent de ActorAgent op de hoogte moet stellen van gewaarwordingen, omdat de WorldAgent weet wat de ActorAgent kan zien, ruiken et cetera. Bij honger ligt dit iets ingewikkelder, omdat dit *intern* aan het lichaam van de ActorAgent is. De vraag is dus, van wie krijgt de ActorAgent door dat hij hongerig is? Moet de WorldAgent zich ook gaan bezighouden met processen die zich in het lichaam van de ActorAgent afspelen, of moet de ActorAgent gemodelleerd worden met een lichaam, waarbij het lichaam van zo'n agent

dan sommige van de gewaarwordingen voor zijn rekening neemt? Welke signalen worden passief ontvangen op de *sensors*, en slechts gefilterd, en welke informatie moet actief worden verzameld?

Voor deze versie van de Verhalenverteller is gekozen om signalen (*perceptions*) die van het lichaam van de ActorAgent zelf komen, ook door de WorldAgent te laten genereren en versturen. In feite modelleert de WorldAgent dan ook het 'lichaam' van de ActorAgent, en de ActorAgent dus slechts het 'brein'.

2.8Honger als gewaarwording

Het eerste wat moet gebeuren om Plop over te laten gaan tot de actie eten, is de gewaarwording honger die Plop moet ervaren. In de KIFbrowser kan met de WordNet functie opgezocht worden waar honger in SUMO geplaatst kan worden.

Honger zelf blijkt nog niet als concept aanwezig te zijn binnen SUMO, maar de WordNet*mappings* hebben wel SUMO termen genomen die er het dichtst bij zitten. De meest passende resultaten bij de zoek term *hunger* zijn:

- hunger, hungriness
  a physiological need for food
  SUMO term(s): kind of BiologicalAttribute
- hunger
  feel the need to eat
  SUMO term(s): kind of Perception

De term *hungry* levert:

- hungry
  feeling hunger; feeling a need or desire to eat food; "a world full of hungry people"
  SUMO term(s): kind of PerceptualAttribute

Hieronder volgt een tabel met de betekenissen van de SUMO termen zoals die in de *documentation* relatie zijn vastgelegd [SUMO].

SUMO term	***documentation string***
BiologicalAttribute	Attributes that apply specifically to instances of Organism.
Attribute	Qualities which we cannot or choose not to reify into subclasses of Object.
Perception	Sensing some aspect of the material world. Note that the agent of this sensing is assumed to be an Animal.
PerceptualAttribute	Any Attribute whose presence is detected by an act of Perception.

2.8.1ValuedAttribute

Om honger te modelleren wordt hier dus gebruikt gemaakt van een attribuut dat *intern* aan een *Organism* kan worden toegekend. Hiervoor voegen we aan de applicatiespecifieke ontologie de term *HungerAttribute* toe als subklasse van *BiologicalAttribute* en *PerceptualAttribute*.

Specifiek voor de Virtuele Verhalenverteller is een concept bedacht die lijkt op *Attribute*, met als extra dat er een waarde kan worden toegekend aan dit attribuut: *ValuedAttribute*. Dit geeft de mogelijkheid om aan te geven *hoeveel* honger Plop heeft door middel van de relatie *attributeValue*. De daadwerkelijke gewaarwording van honger (*Hungry*) vindt nu plaats op het moment dat het *HungerAttribute* een *attributeValue* krijgt boven een bepaalde waarde. De volgende Jess regel is een voorbeeld van hoe dit in zijn werk zou kunnen gaan:

```
(defrule agent-hungry-perception
        (instance ?agent Agent)
        (instance ?agent-attribute HungerAttribute)
        (attribute ?agent-attribute ?agent)
        (instance ?agent-perception Hungry)
        (attributeValue ?agent-attribute ?val)
        (test (> ?val 5))
        =>
        (assert (agent ?agent-perception ?agent))
)
```

Text 1: agent hungry perception

Globaal betekent deze regel dat als er een agent is met een hongerattribuut die een waarde heeft boven de 5 dan wordt deze agent *agent* van een (instantie) van de gewaarwording honger, waarbij de *agent* relatie aangeeft dat de gewaarwording gekoppeld is aan de agent *?agent*.

Andere voorbeelden van *ValuedAttribute* zijn *strength*, *speed* en *health*. In SUMO zijn deze termen ook gedefinieerd, maar de nadruk in SUMO ligt meer op de semantiek van entiteiten (*Entity*) zoals voor mensen te begrijpen, en minder op hoe een karakter in een verhaal realistisch gemodelleerd zou kunnen worden.

2.8.2Perception

Uit figuur 8 kan worden afgeleid dat een ActorAgent dus instanties van percepties (*percepts)* toegestuurd krijgt door de WorldAgent. De *ervaring* van een perceptie bestaat uit de volgende ware predikaten.

```
(instance ?percept Perception)
(agent ?percept ?agent)
```

Waar *?agent* een ActorAgent aangeeft en *?percept* de instantie van *Perception*. Voor Plop om gewaar te worden van honger (om dat te onderscheiden van een willekeurige gewaarwording) moet er ergens een relatie bestaan tussen de gewaarwording en het attribuut dat voor deze gewaarwording verantwoordelijk is. In voorbeeld [XX] is dit niet aangegeven, wat in principe betekent dat de ene agent de gewaarwording van de andere agent kan ontvangen. Om dit probleem op te lossen kan de relatie *apmap,* wat staat voor *attributeperception mapping*, worden ingevoerd. In het voorbeeld staat ook niet de situatie zoals in figuur 8 uitgebeeld goed aangegeven, en daarnaast gaat het heel specifiek over honger. Het volgende voorbeeld van een Jess regel die door de WorldAgent kan worden uitgevoerd:

- is algemener en geldt voor willekeurige perception-attribuut paren,
- geeft het versturen van een perceptie weer, en

- lost de koppeling tussen perceptie en attribuut op door middel van de *apmap* relatie.

```
(defrule send-perception
        (subclass ?attr ValuedAttribute)
        (instance ?inst ?attr)
        (attribute ?inst ?actor)
        (instance ?actor CognitiveAgent)
        (attributeValue ?val ?inst)
        (test (> ?val 5))
        (instance ?percept Perception)
        (apmap ?inst ?percept)
        (immediateInstance ?percept ?pclass)
        =>
        (send-actor-percept ?actor (str-cat "(instance " ?percept " " ?pclass ")"))
        (send-actor-percept ?actor (str-cat "(agent " ?percept " " ?actor ")"))
)
```

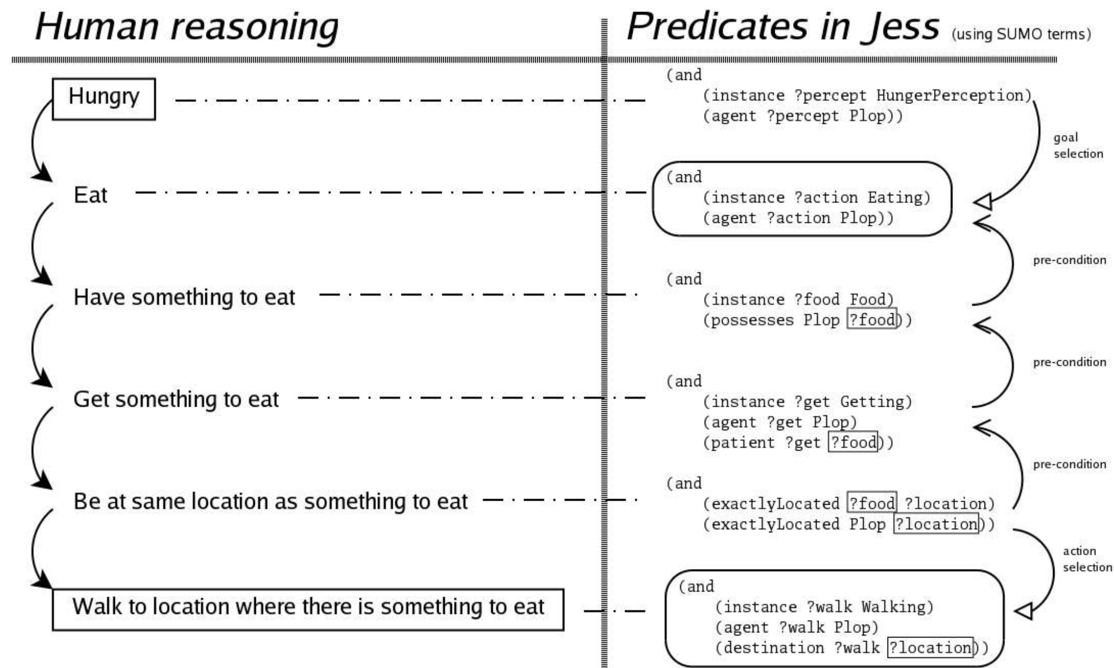
Text 2: WorldAgent send-perception rule

De SUMO relatie *immediateInstance* wordt gebruikt in dit voorbeeld om ervoor te zorgen dat de ActorAgent de meest specifieke klasse van deze *Perception* krijgt opgestuurd. Voor het voorbeeld van Plop is dit dus *Hungry*.

2.9Redenatie van de ActorAgent

Nu uitgelegd is hoe een ActorAgent bepaalde gewaarwordingen kan ervaren, blijft de vraag over hoe een gewaarwording kan leiden tot een actie. Het *think* gedeelte van *sense, think, act* en de vraagteken in figuur 8 moeten nu worden ingevuld. In de versie van de Verhalenverteller van Sander Faas, is hiervoor de *backward-chaining*, ook wel *goal-seeking*, functionaliteit van Jess gebruikt.

De figuur hieronder bestaat uit een gedeelte menselijke redenatie en een gedeelte predikaten in Jess. Het linker gedeelte is overgenomen uit [RENS2004]. De figuur is een poging om een verband aan te geven tussen menselijke redenatie en predikaten in Jess, plus een manier om deze predikaten in Jess door middel van *backward-chaining* als oplossing voor het vraagteken uit figuur 8 te nemen. De rest van de paragraaf zal worden besteed aan de uitleg en uitwerking van dit figuur.



Figuur 9: Honger krijgen en bedenken om ergens eten te halen d.m.v. backward-chaining

De linkerkant van figuur 9 laat zien welke redenatie stappen er kunnen worden gemaakt om van de honger situatie af te leiden dat er naar een lokatie gelopen moet worden waar er iets te eten is. De rechterkant bevat predikaten die waar moeten zijn om de corresponderende redenatiestap mee aan te duiden.

Deze Jess predikaten kunnen niet zo direct in Jess gezet worden, maar kunnen worden gebruikt in regels die de redenatie moeten modelleren. Alle variabelen (aangegeven met een ?prefix) met dezelfde naam vertegenwoordigen hetzelfde object. De vierkant omlijnde variabelen zijn bijzonder in de zin dat deze variabelen hypothetische instanties van objecten aangeven. Dat wil zeggen dat *?food* vanwege de restricties die door de predikaten worden opgelegd betekent '*something to eat*'. En *?location* vanwege de predikaten betekent '*location where there is something to eat*', zonder dat Plop *weet* dat deze objecten echt bestaan.

Het bovenste rond omlijnde predikaat vertegenwoordigt een doel dat Plop moet nastreven om zijn honger te stillen. De onderste vertegenwoordigt een actie die ondernomen kan worden om het op één na onderste predikaat te vervullen.

2.9.1Goal selection

Een ActorAgent kan verschillende percepties hebben, en elke perceptie kan op verschillende manieren er toe leiden dat een bepaald doel nagestreefd dient te worden. Daarnaast oppert Sander Rensen in [RENS2004] dat de DirectorAgent zogenaamde episodische doelen kan opleggen aan de ActorAgents, om zo sturing te geven aan het verhaal. Ook emoties kunnen in [RENS2004] invloed hebben op de selectie van een doel.

Er is dus een bepaald proces in de ActorAgents aanwezig dat moet bepalen welk doel er moet worden nagestreefd. Met *goal-selection*pijl in figuur 9 wordt dit proces van selecteren van het doel eten aangegeven. Vervolgens kan de agent beredeneren welke

acties hij moet ondernemen om dit doel te bereiken.

2.9.2Backward-chaining

Nadat een doel is vastgesteld kan dus een plan opgesteld worden om dit doel te bereiken. In dit voorbeeld is het uitgangspunt dat Plop nog geen voedsel in bezit heeft, en dat hij zich niet bevind op een locatie waar voedsel is. Het plan wordt opgesteld met *backward-chaining.*

Om de actie eten te kunnen uitvoeren moet Plop eerst eten in zijn bezit hebben. Het in het bezit hebben van eten is een pre-conditie voor het uitvoeren van de actie eten. De pre-conditie pijlen in de figuur geven dus aan dat eerst het onderliggende predikaat vervuld moet zijn (waar gemaakt moet worden) alvorens het daarboven liggende predikaat kan worden waar gemaakt. Deze *chain* vormt in feite een soort plan van actie dat uitgevoerd dient te worden om het gegeven doel te bereiken.

Bij een nadere bestudering van figuur 8 valt op dat er twee verschillende soorten van predikaten in de *backward-chain* staan:

1. predikaten die een actie aanduiden, en
2. predikaten die een relatie tussen objecten aanduiden.

Zowel *Eating*, *Getting* als *Walking* zijn allen in SUMO subklassen van *Process*. Ze duiden aan dat deze acties dienen plaats te vinden en de pre-condities daarvoor worden gesteld in predikaten die een relatie tussen objecten aanduiden. Bij de implementatie van het prototype worden deze twee soorten aangeduid met respectievelijk *acteffect* en *releffect*, om aan te geven dat het gaat om een actie die uitgevoerd dient te worden of een relatie die moet gelden, om het doel te bereiken.

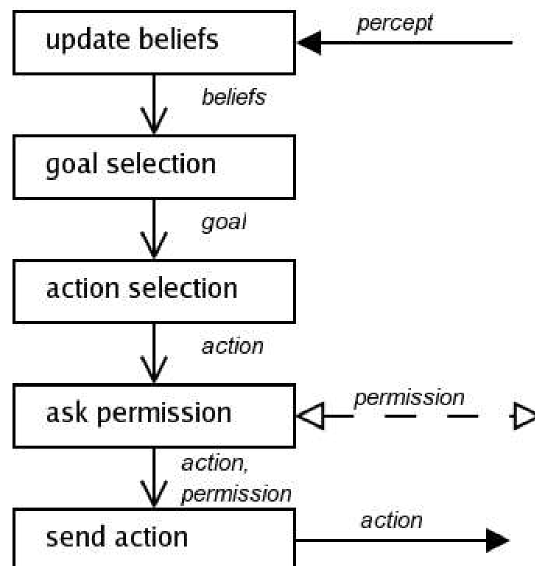
Uiteindelijk eindigt de *chain* met twee *releffects* die aangeven dat Plop zich moet bevinden om dezelfde (hypothetische) locatie als waar het (hypothetische) eten zich bevind. In principe kan elk *releffect* waarvan de pre-condities zijn vervuld leiden tot het uitvoeren een actie.

2.9.3Action selection

Om een bepaald doel te bereiken kan een plan soms meerdere acties suggereren om uit te voeren. Bijvoorbeeld kan het zo zijn dat Plop weet dat er zich op een concrete locatie een concrete instantie van voedsel bevind, maar de optie dat er op een onbekende locatie zich een onbekende instantie van voedsel bevindt, waar Plop misschien wel eerder langskomt, moet open gehouden worden. Tussen de door een opgesteld plan opgeleverde acties moet dus ook nog een keuze worden gemaakt. Dit is wat aangeduid wordt in figuur 9 met de pijl *actionselection.*

2.9.4Module layout

Het hierboven beschreven proces van selectie van doel en actie, tezamen met het feit dat voor het uitvoeren van een actie eerst toestemming aan de DirectorAgent gevraagd moet worden, geeft aanleiding tot het hieronder staande figuur, dat in figuur 8 op de vraagteken zou kunnen worden ingevuld.



Figuur 10: ActorAgent modules

De *update beliefs* module verzorgt de ontvangst van percepts en het bijhouden van een eigen model van de verhalenwereld. Jess biedt de mogelijkheid om het uitvoeren van regels op te splitsen in *modules*, de hierboven getoonde modules zou het uitgangspunt kunnen zijn bij het gebruik van modules voor de ActorAgent.

3.Prototype

3.1SUMO naar Jess vertaler

De SUMO naar Jess vertaler staat in principe los van de Virtuele Verhalenverteller, maar kan wel gebruikt worden voor het samenstellen van een *upper-ontology* op basis van SUO-KIF ontologieën. Zoals in 2.4.1 uitgelegd, bestaat de vertaler uit drie onderdelen: *parsing*, *transfer* en *generation*. Hieronder wordt kort besproken hoe deze onderdelen zijn geïmplementeerd. In bijlage B is uitgelegd hoe het programma gebruikt kan worden.

3.1.1parsing

Het *parsing* gedeelte is in eerste instantie gegenereerd met behulp van JavaCC[8]. Op basis van een grammatica is een parser gegenereerd. Aan de *parser* kan een SUO-KIF bestand worden meegegeven. Vervolgens wordt er een boom opgebouwd die de grammaticale structuur van het bestand vertegenwoordigt. De structuur bestaat uit *nodes* van een bepaald type.

3.1.2transfer

Voordat de transfer plaatsvindt, worden er nog twee filters over de boom uitgevoerd, zoals uitgelegd in 2.5.3. De uiteindelijke transfer vindt plaats met behulp van een bestand dat *mappings* definieert van SUO-KIF zinnen naar Jess zinnen. Dit bestand is niet afgekomen, en het is nog erg veel werk, om voor elke zin-structuur een mapping te definiëren.

3.1.3generation

Het *generation* gedeelte gebeurt via de *SentenceComposer* interface die door *JessSentenceComposer* wordt geïmplementeerd. Op basis van het (Jess) type van de node kan een gedeelte van een zin worden gegenereerd. Al deze stukken aan elkaar geplakt vormt een Jess zin. Alle Jess zinnen worden in een bestand opgeslagen, dat als script kan dienen.

3.2Gebruik van SUMO en de nieuwe architectuur

Voor de demonstratie van de nieuwe architectuur en het gebruik van SUMO is er een nieuwe versie van de Virtuele Verhalenverteller gemaakt. Deze versie is puur bedoelt als prototype en mist belangrijke functionaliteit, die wel in voorgaande versies aanwezig was.

3.2.1Functionaliteit

De NarratorAgent en de PresentationAgent zijn niet in deze versie op genomen. Alle veranderingen die in de verhalenwereld plaats vinden zullen dus slechts achterhaald kunnen worden door de WorldAgent te raadplegen. De volgende agents zijn wel geïmplementeerd met de gegeven functionaliteit:

- **DirectorAgent**:
  kan een WorldAgent starten,
  handelt het initialiseren van de ActorAgents af, en
  geeft (altijd) toestemming voor acties van de ActorAgents.

8 https://javacc.dev.java.net/

- **WorldAgent**:
  maakt gebruik van SUMO en een applicatiespecifieke ontologie,
  stuurt *percepts* naar de ActorAgents wanneer dat nodig is, en
  handelt het uitvoeren van acties door de ActorAgents af.
- **ActorAgent**:
  is opgebouwd uit de modules: *actormain*, actorbeliefs, *goalselection*, *actionselection*, *permission* en voert deze modules in de juiste volgorde uit,
  de *actionselection* module maakt gebruik van *backward-chaining* om een plan van actie op te stellen met waar nodig hypothetische instanties (verder niet uitgelegd),
  selecteert een random actie uit een lijst van mogelijke acties.

3.2.2Problemen bij implementatie

Het gebruik van Jess als redeneerengine leverde nogal wat problemen op. Het blijkt dat Jess vrij stug werkt met name op de hieronder genoemde punten.

backward-chaining

De *backward-chaining* functionaliteit in Jess is vrij beperkt en leverde onvoorziene problemen op.

- Het gebruik ervan is niet transparant voor de gebruiker. Er is geen manier om achteraf te achterhalen op welke manier Jess de *backward-chain* heeft opgebouwd. Daarnaast kan niet worden aangegeven per regel of er gebruik moet worden gemaakt van backward-chaining, maar dit vind op het niveau van relaties plaats.
- Er is een specifieke volgorde waarin zogenaamde *need-facts* (zie [FRIE2003]) worden afgeleid, waardoor ingewikkelde constructies moeten worden bedacht om meerdere verschillende *facts* als pre-condities te gebruiken.
- Ondanks het idee van *backwards-chaining* dat feiten pas hoeven worden afgeleid op het moment dat ze nodig zijn, gaat Jess toch allerlei feiten (*need-facts*) afleiden voordat het echt nodig is. Dit geeft problemen bij grote hoeveelheden regels en feiten (zoals in het geval van de SUMO), omdat er als nog teveel moet worden gedaan.

selecteren op relatie

Jess kent niet de mogelijkheid om een feit te selecteren op zijn relatie argument. Dus het volgende *pattern* wordt niet door de Jess-parser geaccepteerd, terwijl dit wel handig zou zijn, om bijvoorbeeld te weten te komen welke feiten voor Plop van belang zijn.

```
(?rel Plop ?arg)
```

Volgens de Jess website [JESSWEB] zal dit wellicht in toekomstige versies van Jess wel mogelijk worden.

modules

De ondersteuning van modules is beperkt. Het is bijvoorbeeld niet mogelijk om alle feiten in een module te verwijderen, of alleen één module te *resetten*. Dit zou zeer goed van pas komen om bijvoorbeeld bepaalde oude informatie te verwijderen.

queries

Het opvragen van informatie uit het werkgeheugen is mogelijk met behulp van *defquery*. De mogelijkheden binnen een defquery zijn echter zeer beperkt. Het enige wat een *run-query* kan teruggeven is een lijst met bestaande feiten die aan de restricties voldoen. Het is dus niet mogelijk om velden (*slots*) uit feiten te combineren en dit als resultaat terug te geven.

3.2.3Wat er wel en niet werkt

Door de hierboven genoemde problemen is het prototype zeer beperkt, en benadert het voorbeeld van kabouter Plop uit [FAAS2002] wel enigszins, maar niet volledig. Daarnaast is de gebruikersinterface niet uitgewerkt en moet er dus veel met de hand worden ingevoerd.

Het gebruik van SUMO door de WorldAgent wordt wel redelijk goed benut, de regels die geschreven zijn voor het opsturen van *percepts* naar de ActorAgents, zijn redelijk algemeen van aard. Het voordeel hieraan is dat het uitbreiden van de wereld geen toevoegingen aan deze regels tot gevolg heeft.

Met name het onderwerp van het uitvoeren van acties door de ActorAgents is nog niet goed uitgedacht, en ook niet in dit verslag behandeld. Hierdoor wordt het uiteindelijke doel van Plop (het eten van de appel) niet bereikt.

Wel is de ActorAgent in staat om een plan te maken, ook al is deze in het begin niet op de hoogte van dingen die zich in de wereld bevinden. Daarnaast verplaatst Plop zich van het bos naar de huiskamer in het huis alwaar de appel zich bevind.

Het volgen van de stappen genoemd in de handleiding (zie bijlage C) laat zien dat Plop langzaamaan, door om zich heen te kijken en rond te lopen, meer over de verhalenwereld te weten komt, en hier tot op zekere hoogte ook gebruik van maakt.

4.Conclusies

nieuwe architectuur

Het voorstel voor een nieuwe architectuur is gedeeltelijk geïmplementeerd in het prototype. Wat nog mist is de communicatie tussen de WorldAgent en de NarratorAgent en de PresentationAgent. Het prototype laat zien dat de WorldAgent los van de DirectorAgent kan opereren.

SUMO naar Jess vertaler

Het prototype voor de SUMO naar Jess vertaler was nodig, omdat SUMO handmatig vertalen teveel tijd zou kosten en ook vanwege de omvang de kans op fouten groot zou zijn. Het vertalen van SUMO naar Jess is vanwege de tegengekomen problemen geen goed idee. Wat mij betreft zijn er teveel onoverkomelijkheden, waardoor bepaalde dingen weggelaten moeten worden en de kracht van SUMO verloren gaat.

Het karakter van SUMO leent zich ondanks de syntactische overeenkomsten niet voor een afbeelding op een functionele taal als Jess. Jess biedt veel mogelijkheden voor de integratie van een redeneer systeem in Java, maar is te beperkt als het gaat om het verwerken van abstracte noties als genoemd in 2.4.

SUMO is beschikbaar in Protégé en zou misschien met behulp van de bean-generator gebruikt kunnen worden, mits SUMO volledig is vertaald naar Protégé formaat en de beangenerator dit aan kan.

SUMO is beschikbaar in XML formaat. Hier liggen misschien mogelijkheden indien er een ander redeneersysteem wordt gekozen.

SUMO in het MAS van de Virtuele Verhalenverteller

In het prototype is SUMO alleen gebruikt bij de WorldAgent, omdat de *forward-chaining* methode teveel data genereerde, en daardoor testen erg traag verliep. De SUMO is toch wel redelijk groot, maar snelheids en geheugenproblemen, die bij de implementatie aan het licht kwamen, hebben volgens mij vooral te maken met het gebruik van Jess' *forward-chaining* methode, en de *userinterface* die elke keer alle feiten afbeeld.

Om *backward-chaining* regels van de SUMO regels te maken is niet gelukt vanwege de beperkingen van Jess hiermee, maar zeker ook omdat niet altijd even duidelijk is hoe SUMO regels naar *backward-chaining* regels kunnen worden vertaald. Een ander redeneer systeem is nodig indien SUMO gebruikt moet worden.

5.Referenties

Home pages

[JADEWEB] Java Agent DEvelopment home page, http://jade.cselt.it/

[JESSWEB] Java Expert Shell System home page, http://herzberg.ca.sandia.gov/jess/

[SUMO] SUMO Ontology, http://ontology.teknowledge.com/

[SUOWG] Standard Upper Ontology Working Group home page, http://suo.ieee.org/

Overige documenten

[FAAS2002] S.Faas, Virtual Storyteller: an approach to computational storytelling, Masters thesis University of Twente, 2002

[FRIE2003] E.Friedman-Hill, Jess in Action, Manning, 2003

[GIOV2002] G. Caire, JADE tutorial applicationdefined content languages and ontologies, http://jade.cselt.it/doc/CLOntoSupport.pdf, TILAB, 2002

[JURA2000] D.Jurafsky, J.H.Martin, Speech and Language Processing, international edition, Prentice Hall, 2000

[RENS2004] S.H.J.Rensen, De virtuele verhalenverteller: Agent-gebaseerde generatie van interessante plots, afstudeeropdracht aan Universiteit Twente, Enschede, 2004

[RUS2003] S.J.Russel, P.Norvig, Artificial Intelligence, a modern approach, second edition, Prentice Hall, 2003

[SUOKIF] Standard Upper Ontology Knowledge Interchange Format, http://suo.ieee.org/SUO/KIF/suo-kif.html

[THEU2004] M.Theune et al, Emotional Characters for Automatic Plot Creation, University of Twente, Enschede, The Netherlands, 2004

[USERTEK] D.Nichols, A.Terry, User's Guide to Teknowledge Ontologies, http://ontology.teknowledge.com/Ontology_User_Guide.doc, 2003

[WOOL2002] M.J.Wooldridge, An introduction to MultiAgent Systems, John Wiley, Chichester, England, 2002

Bijlage A: SUO-KIF Grammatica

Deze bijlage bevat de grammatica voor SUO-KIF zoals gespecificeerd in [SUOKIF] in BNF syntaxis.

```
upper ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
          N | O | P | Q | R | S | T | U | V | W | X | Y | Z

lower ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
          n | o | p | q | r | s | t | u | v | w | x | y | z

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

special ::= ! | $ | % | & | * | + | - | . | / | < | = | > | ? |
          @ | _ | ~ |

white ::= space | tab | return | linefeed | page

initialchar ::= upper | lower

wordchar ::= upper | lower | digit | - | _ | special

character ::= upper | lower | digit | special | white

word ::= initialchar wordchar*

string ::= "character*"

variable ::= ?word | @word

number ::= [-] digit+ [. digit+] [exponent]

exponent ::= e [-] digit+

term ::= variable | word | string | funterm | number | sentence

relword ::= initialchar wordchar*

funword ::= initialchar wordchar*

funterm ::= (funword term+)

sentence ::= word | equation | inequality |
             relsent | logsent | quantsent

equation ::= (= term term)

relsent ::= (relword term+)

logsent ::= (not sentence) |
            (and sentence+) |
            (or sentence+) |
            (=> sentence sentence) |
            (<=> sentence sentence)

quantsent ::= (forall (variable+) sentence) |
              (exists (variable+) sentence)
```

Bijlage B: Gebruik van de SUMO naar Jess vertaler

Het programma heeft een *command-line interface* kan worden door naar vanuit *sumoparse* directory:

runparser.bat <kif-file>

of

java SuoKifParser <kif-file>

te gebruiken. Voor *<kif-file>* moet het SUO-KIF bestand worden ingevuld. Voor de Virtuele Verhalenverteller is een bestand samengesteld genaamd *subSUMO.kif*.

Als het vertalen goed is verlopen verschijnt de melding

SUO-KIF Parser: translation finished without problems.

Naast de opgegeven *<kif-file>* maakt het programma ook nog gebruik van de hieronder genoemde bestanden.

suo_to_jess.trans

Dit bestand bevat de structuurschema's om SUO-KIF *subtrees* te kunnen vertalen naar een Jess *subtrees*, dat wil zeggen het bevat een mapping van SUO-KIF parse tree naar een Jess parse tree, in overeenstemming met figuur 5.

Het formaat elke regel van dit bestand is als volgt:

[SUO-KIF node] = [Jess node].

waarbij [SUO-KIF node] als volgt is opgebouwd:

<[nodetype];[roottype];[parentlist];[position];null;[tree-id]>

waar

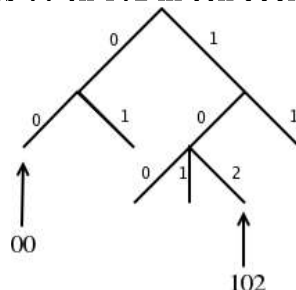
[nodetype] staat voor een integer dat het type van de node aanduidt zoals gedefinieerd in *SuoKifParserTreeConstants*. Het gaat hier om een zogenaamde terminal node, dat wil zeggen een node die een symbool vertegenwoordigt in de zin.

[roottype] een zelfde soort integer die het wortel van deze *subtree* (structuur van de zin) aanduidt.

[parentlist] een door komma's gescheiden lijst van integers die in volgorde van links naar rechts de ouders van deze node aangeven.

[tree-id] is een uniek identificatie voor deze zin. Dit id wordt gebruikt om te bepalen welke regels bij elkaar horen.

[position] geeft de positie van de *terminal-node* in de boom weer. Het volgende plaatje geeft een voorbeeld van posities 00 en 102 in een boom.



De [Jess node] is als volgt opgebouwd:

<[nodetype];[roottype];[tree-nr];[parent-list];[position]>

Waar [tree-nr] staat voor het (unieke) nummer van de op te leveren boom. Het kan namelijk voorkomen dat een SUO-KIF *subtree* meerdere Jess *trees* oplevert.

literals.txt

Dit bestand bevat de SUMO termen die geselecteerd moeten woorden door het *LiteralFilter*. Op elke regel staat een term.

Als uitvoer geeft het programma de volgende bestanden.

subSUMO.clp

Dit is het bestand waar het om gaat. Het bevat regels en feiten die zijn afgeleid uit de *<kif-file>*. Onderaan staat vermeld hoeveel feiten en hoeveel regels het bestand bevat.

failed.log

Hier staan alle structuurschema's van de SUO-KIF parse tree, die niet zijn gevonden in *suo_to_jess.trans*. Dit bestand kan gebruikt worden om het *suo_to_jess.trans* bestand verder uit te breiden. Het formaat de regels in *failed.log* komt overeen met de linker kant van een *suo_to_jess.trans* regel.

Bijlage C: Gebruik prototype van de Virtuele Verhalenverteller

Vanwege het gebrek aan tijd is kan het prototype maar een beperkt deel van wat er allemaal in dit verslag is behandeld laten zien. Aangezien het prototype niet zo robuust is, en nog veel *bugs* bevat, kan alleen door het nauwkeurig doorlopen van de hieronder genoemde stappen iets van de functionaliteit worden getoond.

1. Start een command-line interpreter (DOS-box, terminal)
2. Ga naar de directory vs_ruben en start het programma op met demo.bat.
3. Kies in de menubalk van de DirectorAgent Agents → CreateWorldAgent.
4. Kies in de menubalk van de WorldAgent Knowledge Base → Load Jess File...
5. Open het bestand scripts/storyworld.clp. Het script wordt dan geladen, dit kan even duren. Nadat de melding ready van StoryWorld is gekomen kan de volgende stap worden uitgevoerd.
6. Kies in de menubalk van de DirectorAgent Knowledge Base → Load Jess File...
7. Open het bestand scripts/story_grammar.clp. Nu verschijnt de window van ActorAgent Plop op het scherm.
8. Kies in de menubalk van Plop Knowledge Base → Load Jess File...
9. Open het bestand scripts/actormain.clp. In het facts panel is te zien welke kennis Plop op dit moment heeft. Feiten bevatten een prefix gevolgd door '::', dit geeft de module aan waar dit feit geldt.
10. Voer in het tekstveld bij de WorldAgent het volgende in:
    (attributeValue 7 PlopsHungerAttribute)
    en druk op 'Add Fact'. Hiermee is Plop hongerig genoeg om honger waar te nemen.
11. Druk op 'Run' bij de WorldAgent.
12. In het facts panel bij de ActorAgent Plop zijn nu de volgende feiten te zien:
    (BELIEFS::instance PlopsHungerPerception Hungry)
    (BELIEFS::agent PlopsHungerPerception Plop)
    Dit betekent dat Plop honger ervaart.
13. Druk op 'Run' bij Plop. Druk op 'Run' bij de WorldAgent.
14. Het het factspanel is nu te zien dat Plop nieuwe informatie heeft ontvangen van de WorldAgent.
15. Druk op 'Run' bij Plop. Plop kiest ervoor om naar het huis te lopen, en daar zijn weer nieuwe dingen te zien.
16. Druk op 'Run' bij de WorldAgent. Plop krijgt weer nieuwe informatie en na een nieuwe druk op 'Run' bij Plop zal Plop zich verplaatsen naar de woonkamer (LivingRoom). Na wederom een 'Run' bij de WorldAgent wordt dit weer geëffectueerd, en ziet Plop dat de appel en de peer zich ook in de woonkamer bevinden:
    (BELIEFS::exactlyLocated Apple LivingRoom)
    (BELIEFS::exactlyLocated Pear LivingRoom)

Helaas loopt de demo niet verder dan hier. Een vervelende fout treed er op als de functie *selectacteffect* uit het *actionselection* script wordt uitgevoerd na een 'Run'. Omdat er niet meer tijd was, is het hierbij gelaten.