Faculteit Ingenieurswetenschappen

Dynamische compositie van medische Web services in OWL-S

door

Anna HRISTOSKOVA & Dieter MOEYERSOON

Promotoren:
Prof. Dr. Ir. Filip DE TURCK & Prof. Dr. Johan DECRUYENAERE
Scriptiebegeleiders:
Ir. Sofie VAN HOECKE & Kristof TAVEIRNE & Stijn VERSTICHEL

Scriptie ingediend tot het behalen van de academische graad van
Burgerlijk ingenieur in de ingenieurswetenschappen: computerwetenschappen

Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. Paul LAGASSE

Vakgroep Inwendige ziekten
Voorzitter: Prof. Dr. Martine DE VOS

Academiejaar 2007–2008

Voorwoord

Vandaag leven we in een maatschappij waar mensen constant omgeven zijn door technologie en computers, in de ruimste zin van het woord. We hebben ze aanvaard als onderdeel van het dagelijkse leven, van iets schijnbaar eenvoudigs als een MP3-speler over elektronisch bankieren tot satellietnavigatie.
Helaas lopen bepaalde sectoren wat achter op deze technologische evoluties. De ziekenhuissector is er één van, meer bepaald de afdeling Intensieve Zorgen. Software agents die patiënten constant monitoren en tijdig alarm slaan bij anomalieën, kunnen zo levens redden. Wij hopen met deze thesis het onderzoek naar eHealth applicaties een duwtje in de rug te geven.

Dankwoord

We willen hier ook graag van de gelegenheid gebruik maken om een aantal mensen te bedanken. In de eerste plaats denken we hierbij aan onze begeleiders, Sofie Van Hoecke en Stijn Verstichel. Zij stonden het voorbije jaar steeds klaar om onze vragen te beantwoorden, met ons in discussie te treden en ons in de juiste richting te sturen voor verdere ontwikkelingen. Daarnaast hadden ze nog waardevolle tips in verband met het schrijven van de extended abstract en deze scriptie. Ook alle anderen die ons geholpen hebben bij het nalezen van onze teksten willen we hier bedanken.
Tot slot nog een woord van dank voor al diegenen die interesse toonden in ons werk, ondanks het feit dat het vernoemen van de thesistitel menig wenkbrauw deed fronsen. Deze steun was een bron van motivatie om het steeds nóg beter te doen.

Anna Hristoskova en Dieter Moeyersoon, mei 2008

Toelating tot bruikleen

Anna Hristoskova en Dieter Moeyersoon, mei 2008

Dynamische compositie van medische Web services in OWL-S

door

Anna HRISTOSKOVA & Dieter MOEYERSOON

Scriptie ingediend tot het behalen van de academische graad van
Burgerlijk ingenieur in de ingenieurswetenschappen: computerwetenschappen

Academiejaar 2007–2008

Promotoren:
Prof. Dr. Ir. Filip DE TURCK & Prof. Dr. Johan DECRUYENAERE

Scriptiebegeleiders:
Ir. Sofie VAN HOECKE, Kristof TAVEIRNE & Stijn VERSTICHEL

Faculteit Ingenieurswetenschappen
Universiteit Gent

Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. Paul LAGASSE

Vakgroep Inwendige ziekten
Voorzitter: Prof. Dr. Martine DE VOS

Samenvatting

Deze scriptie bouwt verder op een applicatie voor semi-automatische compositie van Web services die verrijkt zijn met een semantische beschrijving in OWL-S. Deze applicatie werd verder uitgebreid om volledig dynamische compositie te ondersteunen. Daarvoor werden algoritmen ontwikkeld voor het opbouwen van de boomstructuur om de compositie te verwezenlijken. Meer bepaald gaat het hier om algoritmen die lokaal of globaal een QoS-parameter optimaliseren. Vervolgens werden deze algoritmen uitgebreid getest op performantie en schaalbaarheid.

Als use case werd gekozen voor de compositie van medische software services, die op IZ worden ingezet voor het verwerken van monitordata van patiënten.

Trefwoorden

dynamische compositie, semantiek, OWL-S, QoS algoritmen, software agents, eHealth.

Dynamic composition of medical Web services in OWL-S

Anna Hristoskova, Dieter Moeyersoon

Supervisor(s): Prof. Dr. Ir. Filip DE TURCK, Prof. Dr. Johan DECRUYENAERE

Abstract—**Since the Intensive Care Unit is an extremely data-intensive environment, information technology can support the physicians through the composition of medical software services for processing and monitoring patient's data.**
This article presents the use of OWL-S to construct a dynamic composer for Web services. Based on a semantic description of the Web services, the composer enables a service to be executed by creating a composition of Web services that provide the needed inputs. The composition is achieved using various algorithms that can satisfy certain QoS constraints and requirements. Finally performance and scalability results will be presented as well.

Keywords—**dynamic composition, semantic web, OWL-S, QoS algorithms, software agents, eHealth**

I. Introduction

THIS article describes the development of a medical application that can be used in the intensive care unit to speed up the development of new software modules. An existing semi-automatic composer for Web services will be transformed to a dynamic system. First an overview of the semantic markup language OWL-S is presented in section II. Next the development process of the dynamic application is detailed in section III. Finally performance and scalability measurements is analyzed in section IV.

II. OWL-S

OWL-S[1] is a W3C standard for enriching Web services with a computer-interpretable semantic description. This facilitates the automation of Web service tasks including automated Web service discovery, execution, interoperation, composition and execution monitoring.

An OWL-S description of a Web service consists of three parts:

- A *Service Profile* provides the necessary inputs, outputs and other parameters using semantics.
- A *Service Model* describes the service's interface.
- A *Service Grounding* defines the interaction with the WSDL description for invoking the Web service.

Apart from the support for atomic processes, OWL-S also enables the use of composite processes due to the existence of flow controls.

A. OWL-S Tools

While searching for an OWL-S toolkit, some interesting challenges arose. Most of the tools were already collecting dust and none of them had a stable working version. Additionally, there were some interoperability problems regarding the supported OWL-S version.

Despite the aforementioned problems, a toolkit was assembled that could accomplish the required goal.

- The *Web Service Composer*[2], seen in Figure 1, is developed for semi-automatic composition and was the starting point of the application.
- The *Protégé editor*[3] was needed for defining the medical ontology used for the inputs and outputs of the Web services.
- The *OWL-S API*[4] in combination with the Composer allows to invoke the OWL-S Grounding.

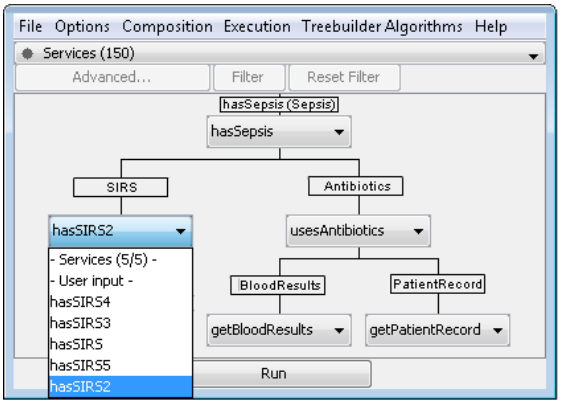


Fig. 1. Web Service Composer

III. Design of a medical application

The Web Service Composer, mentioned in the previous section, was adapted to support dynamic composition. An overview of the architecture can be seen in Figure 2. In the following paragraphs the development process will be outlined.

A. Ontology

A medical ontology was created for defining the inputs and outputs semantically. The Web services were augmented with an OWL-S description that can be generated automatically using the Web Service Composer and were assigned QoS parameters (time and cost). An XSL Transformation was added manually to translate between the medical ontology and the XSD datatypes used in the WSDL.

B. Dynamic composition

The first logical step was making the automatic composition possible. The original Web Service Composer works with semi-

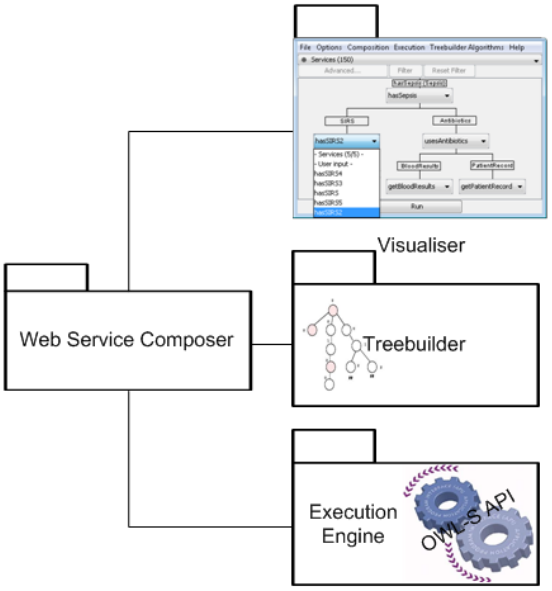


Fig. 2. Architecture of the medical application

automatic composition and uses backward chaining to select the preferred input services for the composition. Using this algorithm, it is possible to automatically create the composition tree without any human intervention. Afterwards, the user can still manually tune the tree as desired.

When the automatic composition was up and running, the Composer was taken to the next level: runtime composition. There is no use of a perfect composition tree if some services become unavailable when invoking the composite service. In order to create the dynamic composition, some changes were implemented in the OWL-S API. In case of a service failure the enhanced Composer automatically defines a new composition replacing the failed node.

C. Parallel execution

The former developers of the Composer weren't taking advantage of the full capabilities of OWL-S. The creation of a composite service from the composition tree was based on putting the different services in depth-first sequence and invoking them one by one. By replacing the *sequence* construct with a *split-join* construct from OWL-S, independent Web services (e.g. children of the same parent node) can be executed in parallel.

D. QoS algorithms

From the start the composition process was in need of a tree-building algorithm. The Composer could make suggestions for the matching input services but the question remained which one to choose. For this purpose a few algorithms were developed.

- *Local algorithms*: contrary to the global algorithms above, the local ones trade optimal weight for performance. A decision is made locally at the current node without considering the whole tree. The decision can be based on: first, random, cheapest or fastest service.
- *Minimal weight optimal*: a backtracking algorithm that composes a minimal weight tree (cheapest or fastest).
- *Minimal weight not optimal*: an algorithm that first computes all the possible trees and then selects the minimal weight tree. Here the user can choose between a serial (minimal tree) or a parallel (minimal path) composition.
- *Tune time vs. cost*: based on the previous algorithm, it first creates a fastest path tree. Then it tunes non-critical paths by searching for cheaper and slower trees while not exceeding the execution time of the slowest path.

IV. Performance and scalability

In this section the different performance and scalability results are presented for the MODS (Multiple Organ Disfunction Syndrome) use case with 6 levels, 35 nodes and 10 semantically equivalent services per node.

Two tests were executed for the different algorithms. One for measuring their execution time and one for the weight of the obtained composition. The optimal backtracking algorithm performed significantly better than the other global ones. The local algorithms were naturally much faster but couldn't guarantee an optimal result.
The tuning algorithm was able to decrease the cost of the composition while maintaining the original time weight.

To evaluate the scalability of the application, the execution time was measured with 5 vs. 10 services per node. As expected, for the global algorithms it grows exponentially. Here the potential of the optimal algorithm becomes apparent. However, the performance of this algorithm largely depends on the distribution of the weights of the services and the opportunities for pruning the tree.

Next serial vs. parallel execution of the composition was compared. The results confirmed the assertion that serial execution time is proportional to the total number of nodes, while parallel execution time is proportional to the depth of the tree.

The last measurement concerned the recovery of a failed service. The possibility to reuse previous results was implemented. When working with equivalent services this can produce a huge time/cost profit.

V. Conclusions

The flexibility of OWL-S made it possible to easily adapt the Web Service Composer for a fully fledged dynamic composition.

Acknowledgments

The authors would like to acknowledge Sofie Van Hoecke and Stijn Verstichel for their support.

References

[1] DAML Services Home Page, http://www.daml.org/services/owl-s/
[2] Web Service Composer, MindSwap, Evren Sirin, http://www.mindswap.org/ evren/composer/
[3] Protégé, Stanford University, http://protege.stanford.edu/
[4] Mindswap OWL-S Java API, Mindswap, Evren Sirin, http://www.mindswap.org/2004/owl-s/api/

Inhoudsopgave

Tabel met afkortingen en symbolen

API:	Application Programming Interface
BPEL:	Business Process Execution Language
DAML:	Darpa Agent Markup Language
DAML+OIL:	Darpa Agent Markup Language + Ontology Inference Layer
DAML-S:	Darpa Agent Markup Language for Web Services
ebXML:	Electronic Business XML
MODS:	Multiple Organ Dysfunction Syndrome
IZ:	Intensieve Zorgen
OWL:	Web Ontology Language
OWL-S:	Web Ontology Language for Web Services
QoS:	Quality of Service
RDF:	Resource Description Framework
SIRS:	Systemic inflammatory response syndrome
UDDI:	Universal Description, Discovery, and Integration
WSDL:	Web Service Description Language
XML:	Extensible Markup Language
XSL:	Extensible Stylesheet Language
XSLT:	Extensible Stylesheet Language Transformations

Hoofdstuk 1

Inleiding

Dynamiek en efficiëntie zijn begrippen van de toekomst. Het World Wide Web ondergaat een evolutie van een statische omgeving naar een dynamische wereld. Daarin zullen software agents een centrale rol spelen. Deze zelfstandige entiteiten zullen, via minimale interactie met de gebruiker, zijn repetitieve en vervelende klusjes opknappen. Zo kunnen ze bijvoorbeeld enkel op basis van een reisbestemming een volledige reis plannen, gaande van een taxi naar de luchthaven over het boeken van een hotel tot het afhandelen van de betaling.

We merken dat deze manier van werken in bepaalde businessmodellen stilaan binnensijpelt. Ook in de medische sector kunnen toepassingen gevonden worden. Op een kritieke afdeling zoals intensieve zorgen moet elke patiënt individueel 24/7 gemonitord worden. Levens kunnen gered worden indien de technologie zo aangewend wordt dat zij de rekenintensieve taken op zich neemt en de vitale analyses aan het medisch personeel overlaat. Indien artsen geen tijd verliezen met het analyseren van hopen papier, grafieken, metingen,... zullen ze optimaler kunnen werken.

Onze scriptie speelt daar op in via een applicatie die dit hele gebeuren tot bij de artsen brengt. Deze applicatie zal de compositie verzorgen van medische Web services om een bepaalde berekening uit te voeren. Zo kunnen artsen op een intuïtieve manier de toestand van hun patiënten opvolgen. Daarvoor zullen de mogelijkheden van de semantische taal OWL-S onderzocht worden om dynamische compositie van Web services te realiseren.

De scriptie is als volgt opgedeeld.

Het *eerste hoofdstuk* is deze inleiding.

Hoofdstuk 2 geeft een inleiding tot de verschillende onderdelen van het semantisch Web. De tekortkomingen van de huidige infrastructuur voor Web services worden eerst blootgelegd. Vervolgens zullen de semantische markup taal OWL en diens uitbreiding voor Web services OWL-S besproken worden.

In *Hoofdstuk 3* zal een selectie van de nodige tools gekozen worden.

Vervolgens zal in *Hoofdstuk 4* onderzocht worden wat de verschillende mogelijkheden zijn voor dynamische compositie. Een aantal bestaande applicaties zullen uitgebreid besproken worden.

De beschrijving van de medische applicatie zal besproken worden in *Hoofdstuk 5*. Eerst zullen de mogelijkheden en tekortkomingen van de gekozen toolkit aan het licht gebracht worden. Daarna komen de uitbreidingen die geïmplementeerd zullen worden aan bod.

Hoofdstuk 6 overloopt stap voor stap het ontwikkelproces van de applicatie. Een semi-automatische composer zal uitgebreid worden via automatische compositie en recovery tot een dynamische composer. Daarnaast zullen een aantal QoS-algoritmes voor het opbouwen van de boomstructuur, nodig voor de automatische compositie, besproken worden.

Als laatste stap van het ontwikkelproces, zal de ontworpen applicatie uitgebreid getest worden op prestatie en schaalbaarheid in *Hoofdstuk 7*. Naast de uitvoeringstijd van de algoritmen, wordt ook de kwaliteit van de bekomen oplossingsbomen bekeken. Ook de prestatie van de recovery zal onder de loep genomen worden.

Onze ervaringen in acht nemend, zal in *Hoofdstuk 8* OWL-S vergeleken worden met een andere opkomende technologie voor de compositie van Web services, BPEL. De voor- en nadelen van beide technologieën komen aan bod.

Tot slot ronden we in *Hoofdstuk 9* af met conclusies en voorstellen voor toekomstige uitbreidingen.

Hoofdstuk 2

OWL-S: Web Ontologie voor het beschrijven van Web Services

Zoals vermeld in de Inleiding (Hoofdstuk 1) willen we ons hier concentreren op het gebruik van de ontologie gebaseerde taal OWL-S voor het beschrijven van Web services. We zullen in dit hoofdstuk dieper ingaan op een aantal belangrijke kenmerken ervan. We zullen eerst een meer algemene uitleg verschaffen omtrent het Semantisch Web en OWL. Daarna zullen we ons volledig concentreren op OWL-S. Hopelijk slagen we erin om de lezer duidelijk te maken waarom deze opkomende technologie het World Wide Web zal kunnen verrijken.

2.1 Introductie: Web services en het Semantisch Web

Web services zijn software *componenten* die een specifieke taak uitvoeren. Ze worden aangesproken door middel van XML berichten gebaseerd op de WSDL interface.
Web services kennen de laatste tijd een groeiende populariteit, die ze te danken hebben aan de flexibiliteit die ze bieden. Ze zijn herbruikbaar en lenen zich eenvoudig tot het bouwen van componentgebaseerde systemen. De huidige infrastructuur voor Web services heeft echter één grote tekortkoming: in de WSDL interface wordt enkel de syntax gespecificeerd, maar niet de semantiek[1] van de operaties. Zoals in [1] beschreven staat, zou het Semantisch Web meer toegang moeten verlenen, niet alleen voor inhoud, maar ook voor services op het Web. Daarnaast is men geïnteresseerd in het bereiken van Web resources ,niet alleen met behulp van kern-

[1] betekenis

woorden, maar ook op basis van de inhoud. Gebruikers en *software agents* moeten in staat zijn tot het ontdekken, oproepen, samenstellen en monitoren van Web resources die een specifieke dienst aanbieden of bepaalde kenmerken hebben. Dit alles zou liefst op zo *automatisch* mogelijke manier moeten gebeuren.

De ontwikkeling van de semantische markup talen zoals *OWL* en zijn voorganger *DAML+OIL* opent onmiddellijk beloftevolle perspectieven voor de groei van het Semantisch Web. Wat hebben ze precies meer te bieden dan de stabiele bestaande architecturen?

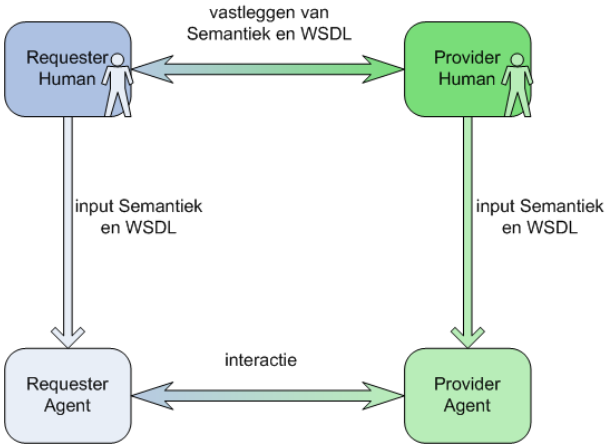
2.2 OWL

OWL is een semantische markup taal die de creatie van *ontologieën*[2] mogelijk maakt. Een OWL ontologie kan gebruikt worden om betekenis te geven aan gebruikte termen[3] en de relaties tussen hen[4]. OWL kan vergeleken worden met markup talen zoals XML, RDF en RDF schema's, maar verschilt daarvan door het verschaffen van semantiek.

OWL kan gebruikt worden voor het beschrijven van Web pagina's. We zijn meer bepaald geïnteresseerd in Web pagina's die een bepaalde service aanbieden. Met *service* bedoelen we Web pagina's die niet puur statische informatie aanbieden, maar in staat zijn om bepaalde dynamische operaties uit te voeren. Gebruikers kunnen zonder veel problemen werken met zulke pagina's en op het eerste zicht hebben we er niets nieuws aan toegevoegd. De lezer zou zich echter moeten herinneren dat we op een zo automatisch mogelijke manier met behulp van het Semantisch Web services willen gebruiken. Daarom introduceren we hier de notie van software agents. *Software agents*, zie Figuur 2.1, moeten in staat zijn om zelf het Web te onderzoeken zonder tussenkomst van de gebruikers. Daarvoor is er echter nood aan computer-interpreteerbare beschrijvingen van de verschillende services die aangeboden worden. Het Semantisch Web en meer bepaald talen zoals OWL zijn het middel voor het beschrijven en delen van service descripties. Op die manier kunnen Web pagina's ook nog verwerkt worden, naast hun puur statische voorstelling.

OWL bestaat uit drie dialecten met toenemende expressiviteit, die afhankelijk van de gebrui-

[2] Een ontologie is de wetenschap voor het beschrijven van de entiteiten en de relaties tussen hen.
[3] classes
[4] properties



Figuur 2.1: Aanmaken van software agents en de interactie ertussen.

kers en de toepassing gebruikt kunnen worden. Deze zijn *OWL Lite*, *OWL DL* en *OWL Full*. De meest gebruikte daarvan is OWL DL, aangezien OWL Full voor het moment niet ondersteund wordt door de bestaande reasoning software.

Voor verdere informatie omtrent de OWL taal verwijzen we de lezer naar een zestal documenten die elk dieper ingaan op een specifiek onderdeel.

OWL Overview[2]: introductie tot OWL.

OWL Guide[3]: gedetailleerd voorbeeld van het gebruik van OWL.

OWL Reference[4]: beschrijving van alle modelleringsprimitieven van OWL.

OWL Semantics and Abstract Syntax[5]: volledige formele beschrijving van OWL.

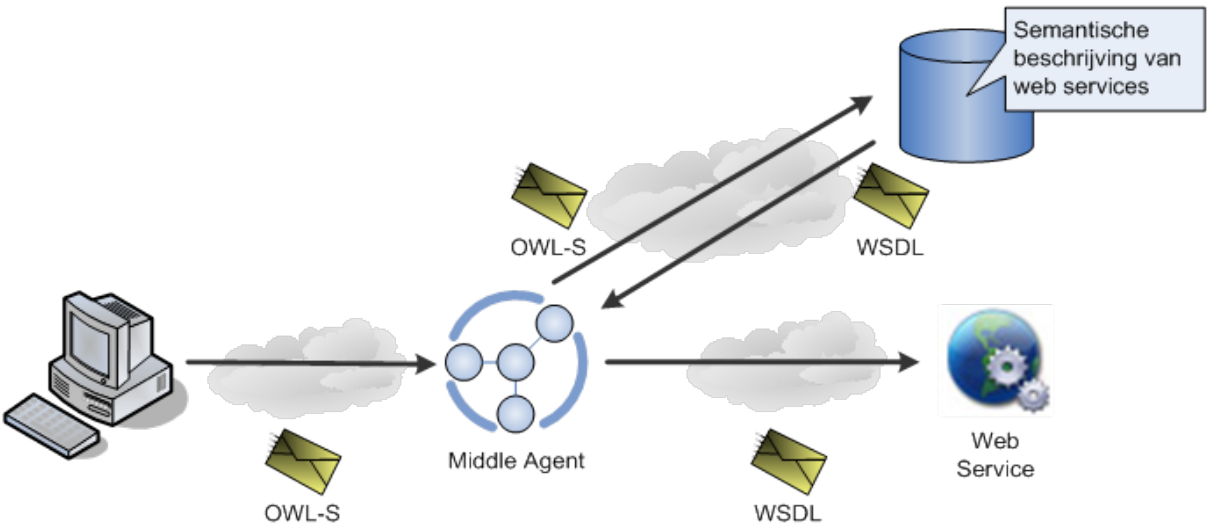
OWL Web Ontology Language Test Cases[6]: een aantal test cases.

OWL Use Cases and Requirements[7]: gebruikersmogelijkheden en vereisten van OWL.

2.3 OWL-S

OWL-S is een ontologie die een uitbreiding biedt voor OWL voor het beschrijven van *Semantische Web services*. Ze is ontworpen door het DARPA DAML programma en is een vervanger voor de oude DAML-S ontologie[8]. Terwijl OWL enkel gebruikt wordt voor het definiëren van entiteiten en de relaties ertussen, is OWL-S in staat om een volledige service te beschrijven. Dit zal gebruikers en software agents in staat stellen om op een automatische manier Web resources

te ontdekken, op te roepen, samen te stellen en te monitoren op basis van welbepaalde kenmerken. Figuur 2.2 toont de basisarchitectuur voor het oproepen van een Web service met behulp van OWL-S. De client moet over een OWL-S beschrijving beschikken van de specifieke Web service die nodig is. Een voorbeeld van een OWL-S beschrijving kan in Appendix A gevonden worden. De OWL-S beschrijving kan van een bestaande Web service zijn of kan onafhankelijk van een Web service opgesteld zijn. Vervolgens wordt naar een geschikte Web service gezocht in een repository. Men kan eventueel meerdere services samenstellen om de gevraagde taak uit te voeren. Als er een juiste service of service compositie gevonden wordt, kan men met behulp van de WSDL beschrijving deze oproepen.



Figuur 2.2: Gebruik van OWL-S bij het oproepen van Web services

2.3.1 Doel

OWL-S ondersteunt twee categorieën van Web services, nl. atomaire en samengestelde of complexe. *Atomaire services* worden gebruikt in simpele gevallen wanneer één enkel programma, sensor of toestel opgeroepen moet worden om een bepaalde operatie uit te voeren en eventueel een resultaat terug te geven. Men kan niet spreken van uitgebreide interactie tussen de gebruiker en de service. *Complexe services* daarentegen zijn samengesteld uit meerdere primitieve services. Hierbij zullen meerdere interacties of conversaties nodig zijn tussen de gebruiker en die services.

Daarnaast zou OWL-S de volgende functionaliteit moeten aanbieden:

- **Automatische Web service publicatie**: met de ontwikkeling van het Semantisch Web zal het aantal beschikbare services op het Web drastisch groeien. Elk ervan zal een specifieke functionaliteit aanbieden. OWL-S zal als hulpmiddel kunnen gebruikt worden door software agents voor het ontdekken van de Web service die een specifieke taak kan vervullen en die voldoet aan een aantal kwaliteitsvoorwaarden, zonder dat er daarbij nood is aan de tussenkomst van de gebruiker.

- **Automatische Web service invocatie**: software agents zijn entiteiten die in staat zijn om bepaalde Web services op te roepen op basis van hun WSDL beschrijving. Dankzij OWL-S zijn deze agents in staat om op een automatische manier de beschrijving van inputs en outputs van een service te lezen en deze op basis daarvan op te roepen.

- **Automatische Web service compositie en samenwerking**: naast het puur ontdekken en oproepen van services zou men meer complexe taken kunnen voeren. Deze zullen een gecoördineerde invocatie en compositie van meerdere Web services inhouden.

2.3.2 Opbouw van de Service Ontologie

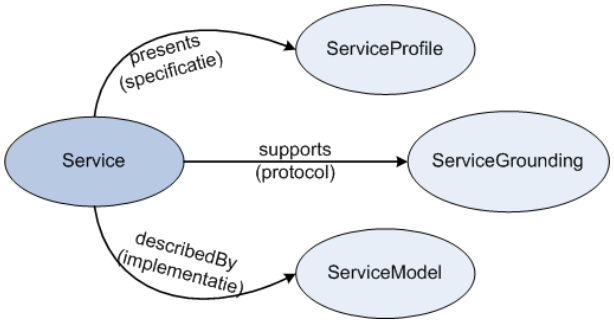
Om dit allemaal mogelijk te maken zal OWL-S de beschrijving van een Web service structureren in drie types van ontologieën (zie Figuur 2.3). Elk zal verantwoordelijk zijn voor een deel van de servicebeschrijving en beantwoordt een specifieke vraag. Deze zijn:

Service Profile: Wat heeft de service te bieden aan mogelijke gebruikers? Het profiel wordt gebruikt voor het publiceren van de service. Elke instantie van de klasse Service biedt[5] een Service Profile aan.

Service Model: Hoe wordt hij gebruikt? Een instantie van de klasse Service wordt omschreven[6] door een Service Model.

Service Grounding: Hoe interageert men ermee? De Grounding specificeert de nodige details omtrent het transport protocol. Elke instantie van de klasse Service ondersteunt[7] een Service Grounding.

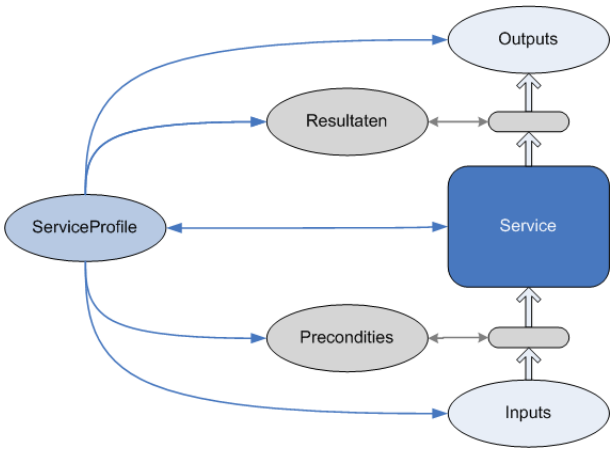
[5] presents
[6] describedBy
[7] supports



Figuur 2.3: Opbouw van een service ontologie

Meer algemeen verstrekt het Service Profile de nodige informatie voor een software agent om de service te vinden. Het Service Model en de Service Grounding beschrijven de manier waarop de agent gebruik kan maken van de functionaliteit van de service. Er zijn een paar beperkingen omtrent deze ontologieën, nl. een service kan door maximum één Service Model beschreven worden en een Service Grounding wordt gekoppeld aan slechts één service. Wij zullen nu elke ontologie meer in detail bespreken.

Service Profile



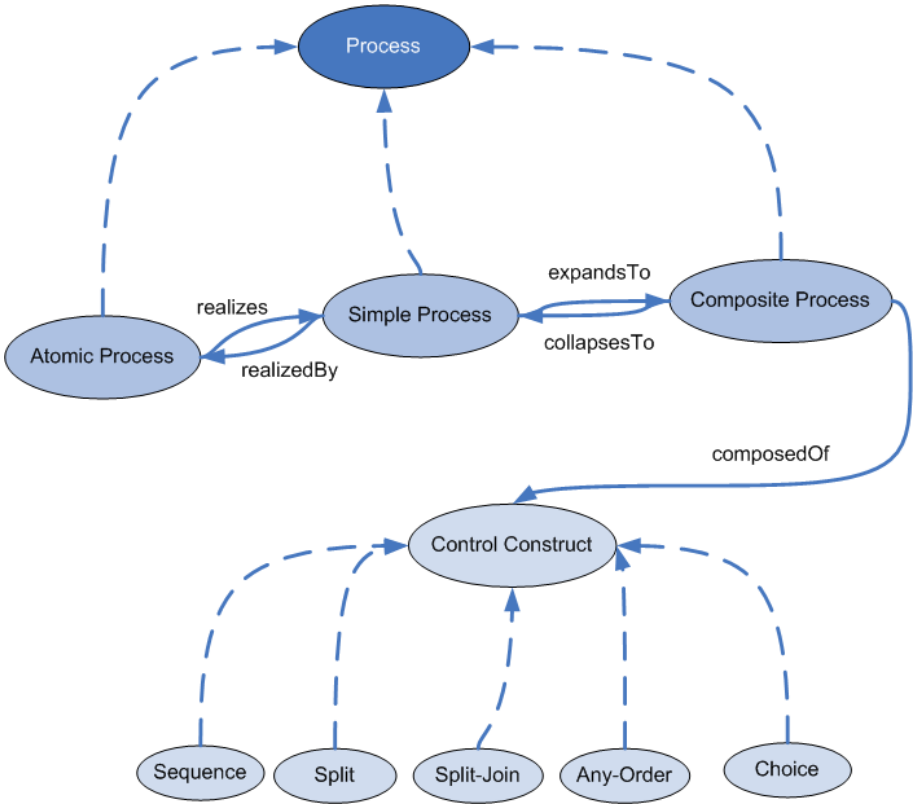
Figuur 2.4: Voorstelling van het Service Profile

Het Service Profile legt vast wat de service kan, op een manier die verwerkt kan worden door een agent. Zo kan de agent bepalen of de service voldoet aan zijn noden. Zoals op Figuur 2.4 te zien is, presenteert het profiel de nodige inputs en de gegenereerde outputs, de vereisten waaraan men moet voldoen om de service met succes te kunnen gebruiken en de verwachte resultaten bij de uitvoering van de service. Daarnaast kunnen eventuele beperkingen, de kwa-

liteit van de service en bepaalde parameters gespecificeerd worden.

Service Model

Het Service Model omschrijft hoe men de service kan gebruiken door een beschrijving van de semantische inhoud van de aanvragen, de condities waaronder bepaalde resultaten zullen verkregen worden en indien nodig (bij complexe processen) de verschillende stappen die men moet uitvoeren om een resultaat te bekomen. Met andere woorden, het laat weten hoe men een service kan oproepen en wat het resultaat zal zijn van die oproep. Deze beschrijving kan zeer nuttig gebruikt worden bij het uitvoeren van meer complexere taken, zoals compositie en coördinatie van meerdere services om bepaalde bewerkingen uit te voeren.



Figuur 2.5: Opbouw van het Service Model

Zoals aangegeven op Figuur 2.5 ondersteunt een Service Model drie types van processen, nl. atomaire, simpele en complexe.

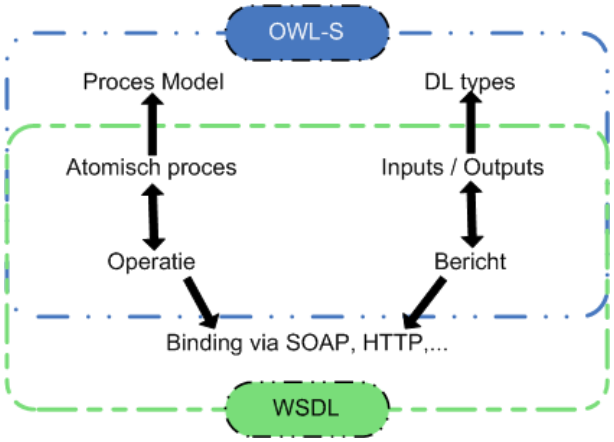
Atomaire processen stellen één enkele interactie voor. Ze hebben geen subprocessen en worden uitgevoerd in één enkele stap.

Complexe processen corresponderen met interacties bestaande uit meerdere stappen, waarbij verschillende services opgeroepen kunnen worden. In tegenstelling tot atomaire processen zijn ze wel afbreekbaar in meerdere subprocessen. Voor de compositie van subprocessen gebruikt men constructoren met elk zijn specifieke functionaliteit.

Simpele processen zijn slechts een abstractie voor het weergeven van meerdere views van hetzelfde proces. Net als atomaire processen omvatten ze één executiestap. Ze kunnen op die manier een atomair proces of een gesimplificeerd beeld van een complex proces voorstellen.

Service Grounding

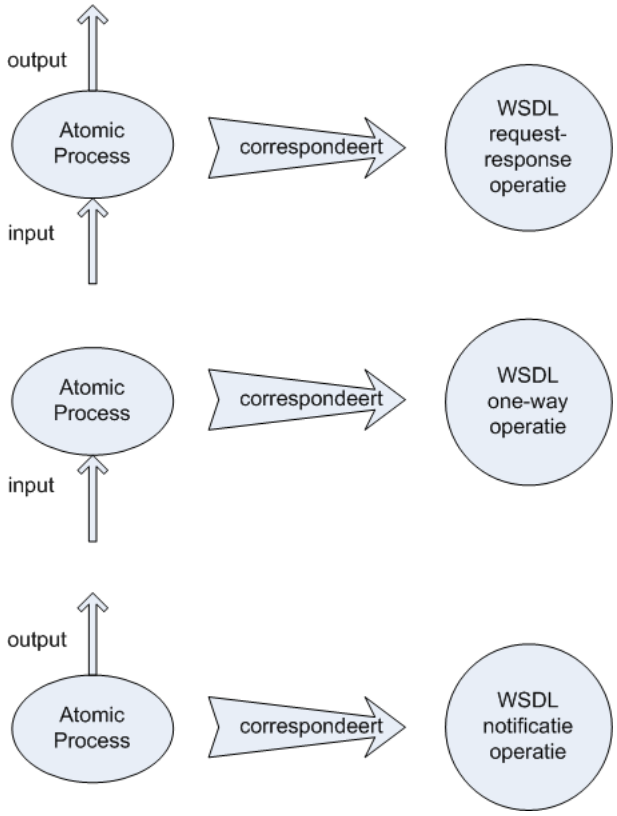
De Service Grounding is een omschrijving van hoe een agent de service kan oproepen. De grounding definieert onder andere een communicatieprotocol, berichtformaat, adressering en serialisatietechnieken voor inputs en outputs. De grounding kan aanzien worden als de concrete realisatie van de abstracte definities in het profiel en het model.



Figuur 2.6: Mapping tussen OWL-S en WSDL

Aangezien er al genoeg goede technieken bestaan voor het omschrijven van Web services, maakt de grounding gebruik van de bestaande WSDL architectuur. De OWL-S grounding en de WSDL kunnen dan als complementair beschouwd worden, wat duidelijk blijkt uit Figuur 2.6.

Terwijl de OWL-S grounding de koppeling met het Service Model en de definitie van de verschillende datatypes voorstelt, specificeert de WSDL het gebruikte transportprotocol. De overeenkomsten tussen de twee liggen in het overlappingsgebied. Een OWL-S atomair proces stemt overeen met een WSDL operatie (Figuur 2.7). Daarnaast is de definitie van de OWL-S inputs



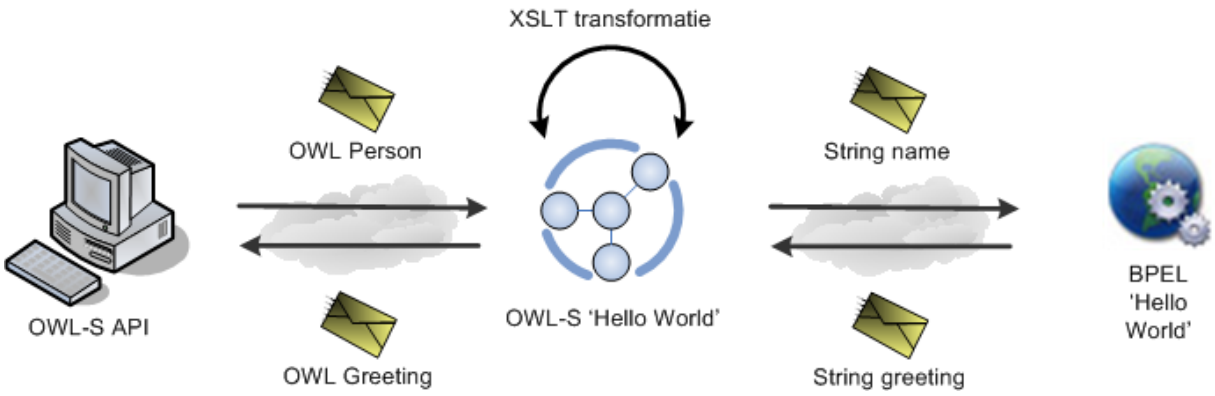
Figuur 2.7: Vergelijking atomaire processen en WSDL descripties

en outputs gelijk aan een WSDL bericht[8].

2.4 OWL-S Hello World Case

We hebben een simpele Hello World applicatie uitgewerkt om beter vertrouwd te geraken met OWL-S en de tools waarmee we verder zullen werken (Figuur 2.8). Er werd een BPEL Hello World service gemaakt. Op basis van de WSDL beschrijving ervan hebben we met behulp van de Protégé Editor (Appendix B.1.1) een OWL-S beschrijving gegenereerd voor de service. Tevens hebben we een ontologie 'Person' en 'Greeting' aangemaakt, te gebruiken als input en ouput voor de semantische service. Een laatste aanpassing was een XSLT transformatie om de data uit de ontologie te kunnen extraheren en door te geven naar de service. Het geheel werd uitgevoerd door de OWL-S API van Mindswap (Appendix B.2.1).

[8]Men is ook in staat om XSL transfomaties uit te voeren zodat de inputs en de outputs niet volledig hoeven overeen te komen met een specifiek WSDL bericht. Dit heeft vooral als voordeel dat de OWL-S en de WSDL specificatie niet op hetzelfde moment moeten gemaakt worden. Op die manier kunnen bestaande systemen ook uitgebreid



Figuur 2.8: 'Hello World' case

worden met semantiek.

Hoofdstuk 3

OWL-S Tools

In dit hoofdstuk gaan we op zoek naar tools die ons het werken met OWL-S kunnen vergemakkelijken. We verzamelen eerst zoveel mogelijk tools (zie appendix B); de websites van DAML[9], SemWebCentral[10] en het overzicht in presentatie [11] vormen hiervoor een goed startpunt. Vervolgens spelen we wat met elk ervan om de sterke en zwakke punten aan het licht te brengen om uiteindelijk een selectie te maken van diegene die we zullen gebruiken.

3.1 Selectie tools

3.1.1 Tegengekomen problemen

OWL-S is duidelijk nog een nieuwigheid in de World Wide Web wereld en nog volop in ontwikkeling. Door onze experimenten met alle mogelijke OWL-S tools hebben we een zeer goed beeld gekregen van een aantal terugkomende problemen. Deze zijn te wijten aan de status van ontwikkeling van OWL-S zelf of de specifieke tool.

- De meeste tools beschikken niet over een stabiel werkende versie aangezien de auteurs ervan nog in de leerfase zitten van OWL-S. Een deel ervan is zelfs niet actief meer, waardoor de tools in de vergetelheid dreigen te verdwijnen.
- De verschillende OWL-S versies zijn niet volledig compatibel. Daarbij komt ook nog het feit dat de bestaande tools slechts een specifieke versie ondersteunen en daardoor ook niet compatibel zijn met mekaar.

Bvb. De OWL-S API ondersteunt de laatste stabiele versie (1.1) van OWL-S terwijl de stabiel werkende versie van de Protégé editor de nieuwste versie (1.2) van OWL-S ondersteunt. Hierdoor moesten we een oude versie van Protégé en de bijhorende OWL-S plugin gebruiken die echter nog veel bugs en compatibiliteitsproblemen vertoonde.

- Vaak zijn er ook compatibiliteitsproblemen tussen het hoofdprogramma en de plugin die OWL-S mogelijk maakt.
  Bvb. Aangezien we met een wat oudere versie moesten werken van de editors, waren er heel wat compatibiliteitsproblemen tussen. Het belangrijkste was dat de Protégé OWL-S editor plugin niet in staat was correcte XSL transformaties aan te maken.

3.1.2 Gebruikte tools voor de scriptie

Door alle tegengekomen problemen hebben we besloten om het simpel te houden en uiteindelijk hebben we ervoor gekozen om het bij drie tools te houden.

- **Web Service Composer**: We hebben gekozen om te starten vanaf de bestaande versie van de Web Service Composer en deze verder uit te breiden voor onze doeleinden. De source code ervan is vrij beschikbaar, maar al jaren niet aangeraakt door de auteur ervan. Bijgevolg waren er van in het begin enkele correcties nodig om deze werkend te krijgen. Het voornaamste voordeel van de Composer is dat deze zeer simpel is en het idee erachter is zeer goed uitgedacht. Bijkomend is de Composer in staat om zelf OWL-S te genereren op basis van een bestaande WSDL descriptie. Het enige wat we nog manueel moeten toevoegen zijn de XSL transformaties.
- **Protégé editor**: Aangezien de Composer in staat is om OWL-S te genereren, hebben we Protégé enkel nodig om eenvoudige ontologieën te maken.
- **Java OWL-S API**: De Composer maakt gebruik van de OWL-S API om de Web services op te roepen. We hebben de source code ervan ook gedownload, zodat we die ook konden uitbreiden.

Meer dan deze tools hadden we niet nodig om een mooie en nuttige applicatie zonder veel problemen uit te breiden. Een goede raad van ons is dan ook: *keep it simple!*

Hoofdstuk 4

Dynamische Compositie van Web Services

4.1 Software Agents

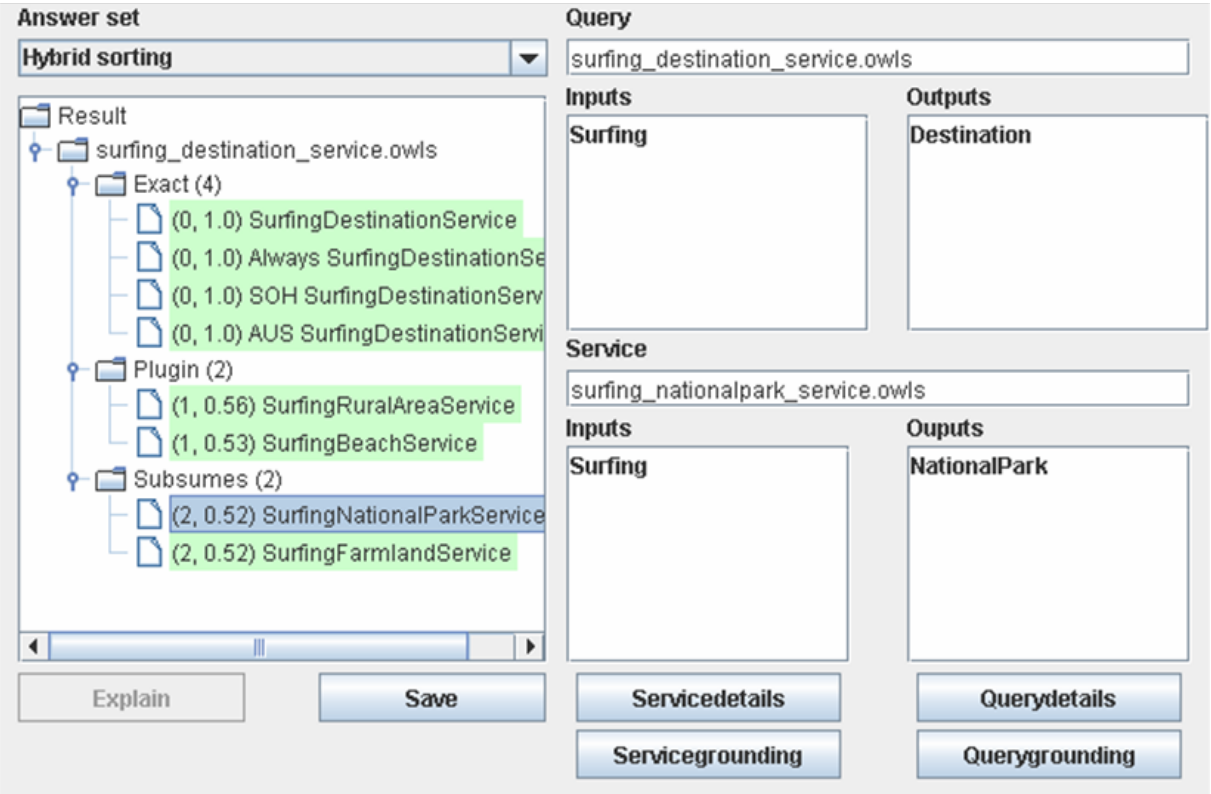
In Hoofdstuk 2 (OWL-S) hebben we even het concept van *software agents* als autonome entiteiten aangeraakt. Deze agents moeten in staat zijn bepaalde taken zelf of in samenwerking met andere agents uit te voeren zonder tussenkomst van de clients. Op weg naar een meer autonoom Web dat in staat is alle eisen van de gebruikers uit te voeren, is deze zelfstandige samenwerking van software agents van groot belang. Daarom zijn we geïnteresseerd in entiteiten die deze samenwerking verzorgen.

We kunnen verschillende gradaties van compositie tussen de agents onderscheiden. Op het laagste niveau vinden we de *Matchers* die twee software agents vergelijken om na te gaan in hoeverre de outputs van de ene als input kunnen fungeren voor de andere. Op het niveau daarboven hebben we de *Composers*. Deze kunnen op *semi-automatische* of *automatische* wijze een volledige compositie van agents samenstellen en uitvoeren. Op het hoogste niveau bevinden zich de *Middle Agents*. Deze software componenten kunnen naast een volledige compositie van software agents ook andere taken verzorgen, zoals transformatie van vragen en antwoorden, kwaliteit van de services, enz. We zullen de verschillende mogelijkheden bespreken samen met een aantal voorbeelden van bestaande systemen.

4.2 Matchers

De Matchers zijn software componenten die zoeken naar passende services. Dit gebeurt op basis van een specifiek profiel dat men nodig heeft. Zo kan men de nodige service vinden om een bepaalde taak uit te voeren of twee software agents aan elkaar te koppelen. Bestaande OWL-S systemen kunnen ook een score geven al naar gelang de mate waarin de services voldoen aan het gevraagde profiel en zo de beste service uitkiezen. In Appendix B (Tools) hebben we twee matchers besproken, nl. de OWL-S Matcher (DAML-S Matcher) en de OWL-S MX Matchmaker.

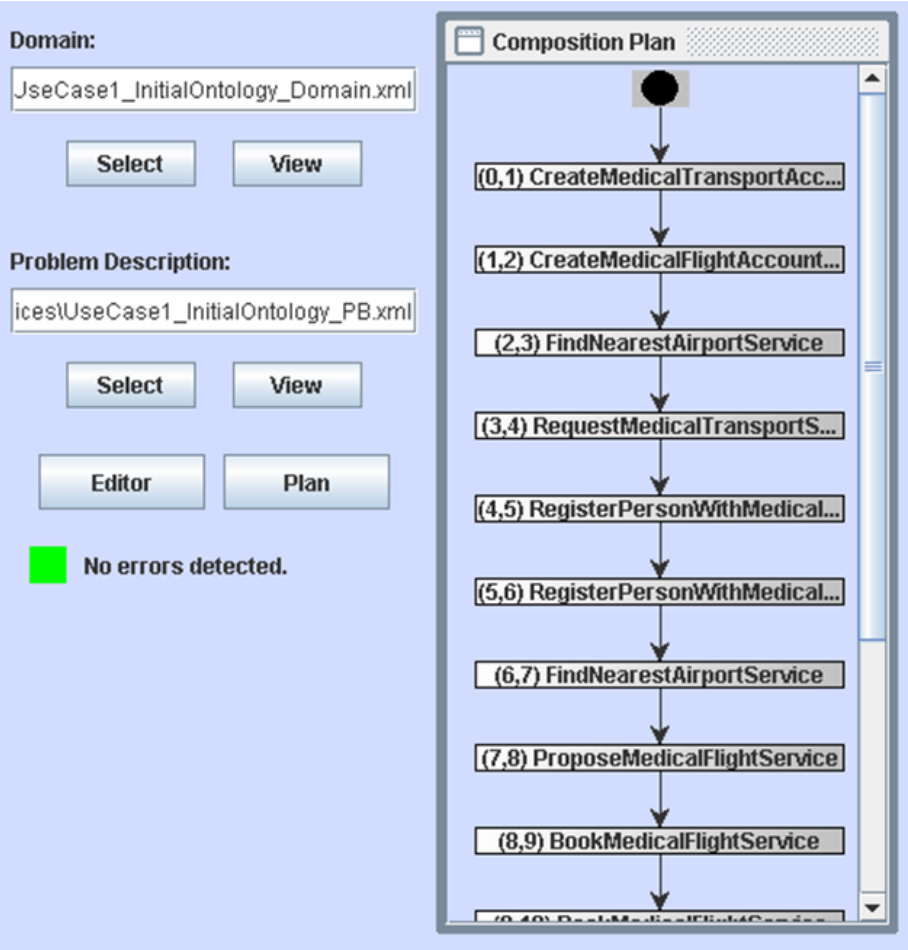
De *OWL-S Matcher* kan twee software agents vergelijken, terwijl de *OWL-S MX Matchmaker* uit een aantal software agents kan kiezen en voor elk een score zal genereren. In Figuur 4.1 is daarvan een voorbeeld te zien. Er zijn vier software agents gevonden die exact voldoen aan het gevraagde profiel (gevraagde inputs en outputs). Daarnaast zijn er twee die als alternatieven kunnen gebruikt worden. Dit houdt in dat hun inputs en/of outputs een ouder-kind relatie hebben met de gevraagde inputs/outputs. Dus in dit voorbeeld is 'NationalPark' een soort 'Destination'. Zo kunnen we onmiddellijk zien waarom het semantisch Web bepaalde voordelen kan bieden bij het samenwerken van meerdere software agents.



Figuur 4.1: Rangschikking passende software agents mbv. OWL-S MX Matchmaker

4.3 Composers

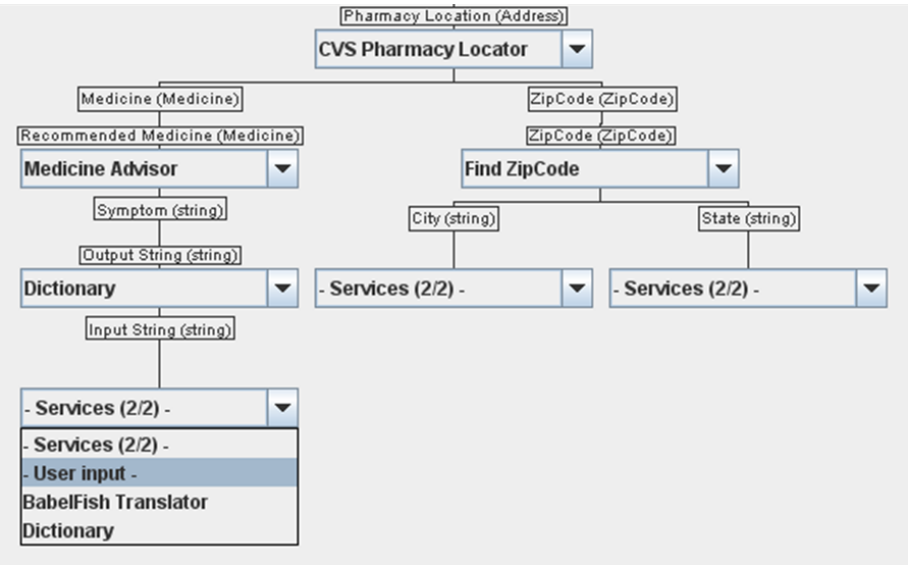
De Composers maken gebruik van *AI planning* om een volledige compositie van software agents te verwezenlijken. Het resultaat van hun compositie is een volledig pad van software agents die een gevraagde taak kunnen uitvoeren. We kunnen twee methodes van compositie onderscheiden, nl. automatische en semi-automatische compositie.



Figuur 4.2: Volledige compositie mbv. OWLS-Xplan

OWLS-Xplan [12] is een tool die automatische compositie kan verzorgen zonder tussenkomst van gebruikers. De werking ervan werd reeds uitgelegd in Sectie B.4.
Om een en ander wat duidelijker te maken, volgt hier een voorbeeldje. Een wintersporter heeft een ongeval gehad en ligt in een ziekenhuis in Oostenrijk, maar moet naar België gerepatrieerd worden. De beschrijving van de initiële toestand zal dan aangeven dat de skiër in een bed in het ziekenhuis in Oostenrijk ligt (met o.m. adres van het ziekenhuis). Het doel zal dan beschrijven

dat de patiënt een bed in een Belgisch ziekenhuis heeft. Xplan zal dan een compositie maken die Web services bevat voor het reserveren van een ziekenhuisbed, een ziekenwagen van en naar de luchthaven, een medische vlucht, etc. Zie Figuur 4.2 voor een screenshot van dit voorbeeld. De *Web Service Composer* [13, 14, 15] is een andere tool die een semi-automatische compositie kan genereren. In Figuur 4.3 is er een voorbeeld van de interface te zien. Bij elke volgende stap wordt er een lijst van passende services bijgehouden die men kan koppelen aan de voorgaande service. De gebruiker kan dan zelf kiezen uit deze lijst en zo de flow bepalen.

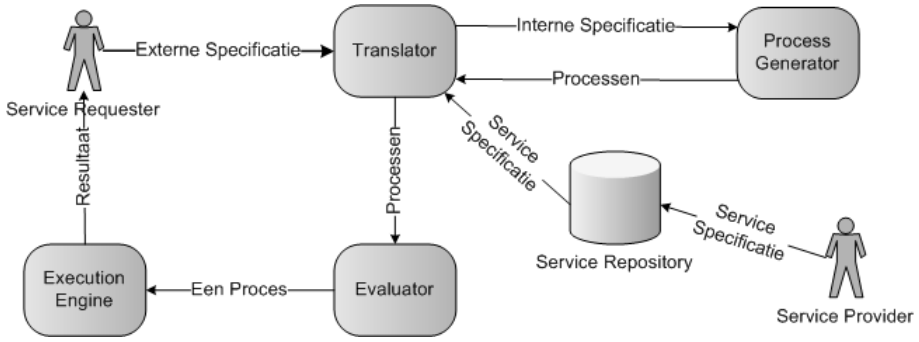


Figuur 4.3: Manuele compositie mbv. Web Service Composer

4.4 Middle Agents

Middle Agents zijn Web services die zich bevinden tussen de gebruiker en de aanbieder van een Web service. Op Figuur 4.4 kan men de werking van een Middle Agent bekijken. De aanbieders zullen zich bij de Middle Agent registreren, zodat deze een overzicht heeft van alle beschikbare Web services. De gebruiker zal zijn vraag lanceren aan de Middle Agent, die dan op zoek gaat naar een geschikte service en deze uitvoert om de vraag te beantwoorden. Deze architectuur biedt voordelen omdat de Middle Agent de gebruikers en aanbieders bijeenbrengt. Zonder dit systeem zouden bvb. alle aanbieders zich moeten registeren bij alle gebruikers, wat uiteraard veel complexer is. Daarnaast biedt de gecentraliseerde aanpak ook mogelijkheden tot extra diensten zoals load balancing (indien er meerdere services zijn met dezelfde functionaliteit, kan de Middle Agent deze alternerend gebruiken bij opeenvolgende aanvragen), vertalingen

tussen verschillende programmeertalen/ontologieën, beveiliging, etc.

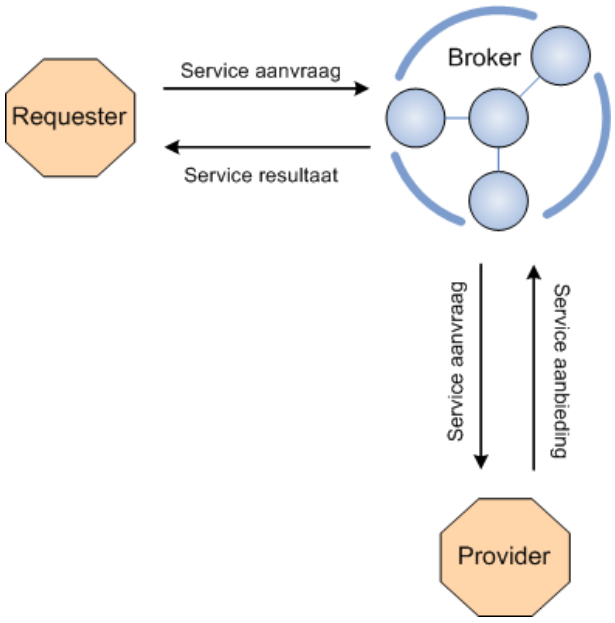


Figuur 4.4: Framework voor een service compositie systeem

Afhankelijk van de functionaliteit die de Middle Agents verder aanbieden kunnen we hen onderverdelen in verschillende klassen. We bekijken er twee van, met name de Brokers en de Matchmakers.

4.4.1 Broker

De Broker heeft de meeste functionaliteit. Wanneer hij een vraag krijgt van een gebruiker, zal hij deze volledig proberen te beantwoorden. De gebruiker hoeft maar te wachten op het eindresultaat zoals te zien op Figuur 4.5. Dit betekent dat er verschillende taken achter de schermen



Figuur 4.5: Broker als Middle Agent

kunnen uitgevoerd worden:

- In het eenvoudigste geval bestaat er een geschikte Web service en volstaat het deze op te roepen en het antwoord door te sturen naar de gebruiker.
- Het is ook mogelijk dat de inputs in verschillende ontologieën beschreven zijn (bv. 'ZIP' en 'Postcode'). De Broker kan dan zorgen voor een correcte omzetting van de ene ontologie naar de andere.
- Moeilijker wordt het wanneer er geen Web service bestaat die direct aan de vraag kan voldoen. In dat geval zal de Broker proberen op basis van de gekende Web services een compositie samen te stellen die wél een antwoord kan geven. Dit kan hij zelf doen of delegeren aan een gespecialiseerde compositie-agent. Als er een compositie gevonden wordt, zal de Broker de services in de juiste volgorde oproepen en zorgen dat de inputs en de outputs op een correcte manier aan elkaar gekoppeld worden. Het eindresultaat wordt dan naar de gebruiker gestuurd, die niets hoeft te weten van heel het achterliggend proces.

Eigenschappen

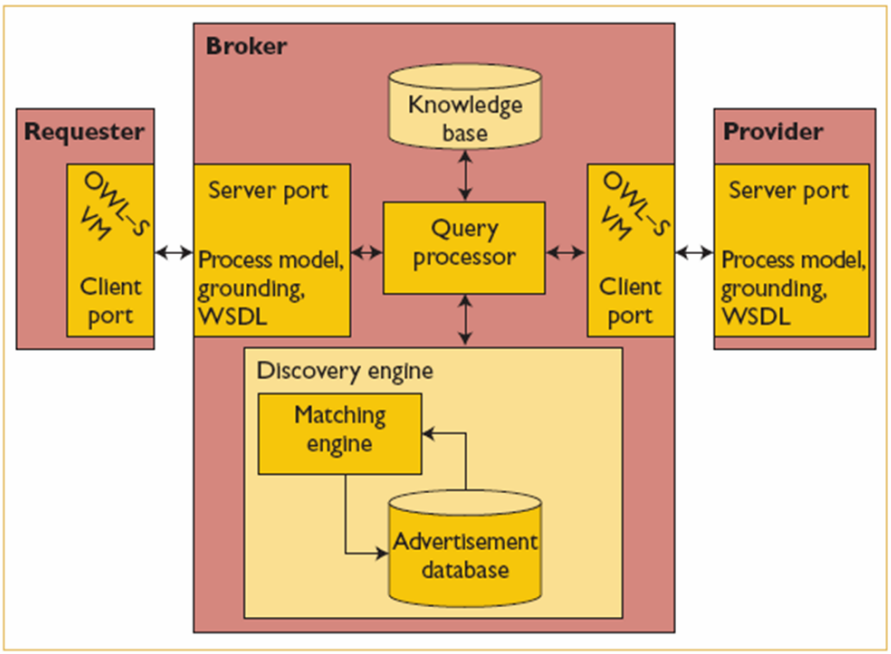
Een volledige Broker zal beschikken over de volgende eigenschappen:

- Aankondigingen in lokale knowledge base
- Discovery van service-aanbieders
- Gecentraliseerd: single point of failure
- Load Balancing
- Vertaalservices
- Vertrouwde tussenpersoon
- Anonimiteit

Voorbeelden van Brokers

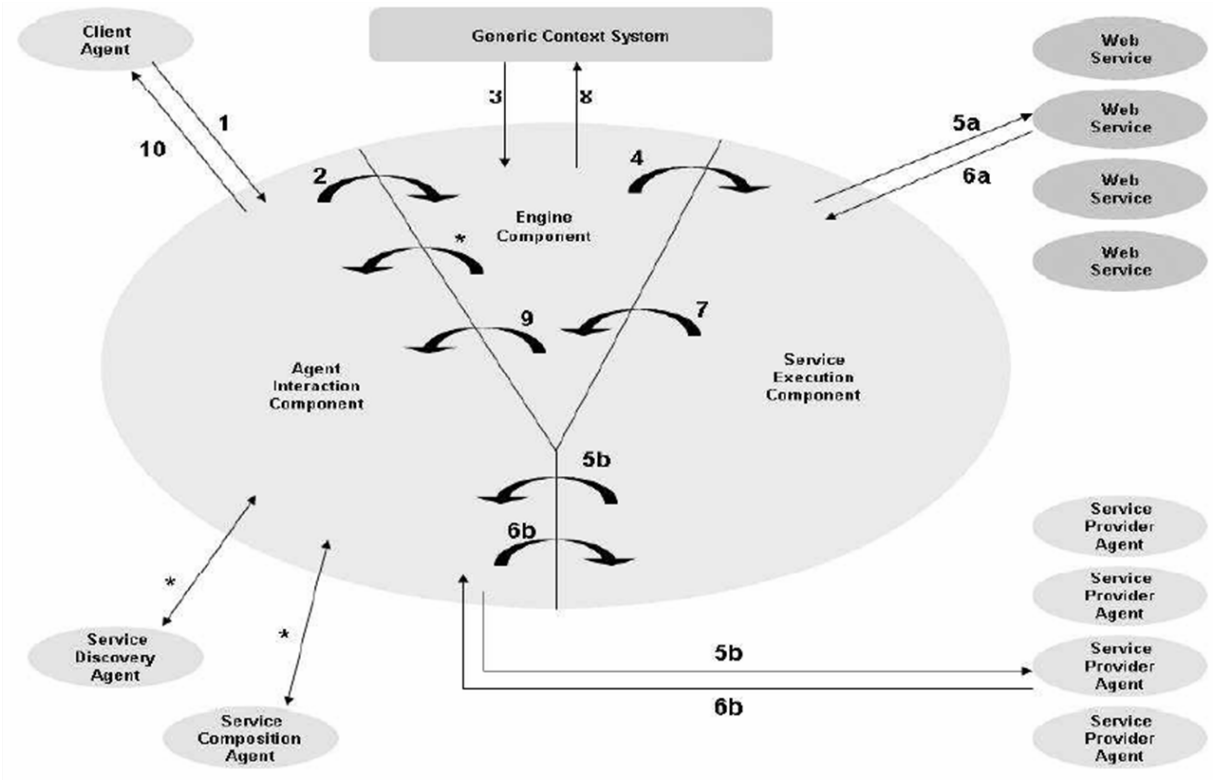
We hebben volgende voorbeelden van Brokers gevonden:

- De *OWL-S Broker*[16, 17], Figuur 4.6, is tot nu toe de enige officiële Broker. Hij is uitgebreid beschreven in de literatuur, maar een implementatie ervan is niet beschikbaar.



Figuur 4.6: OWL-S Broker

- *SEA Broker*[18, 19] + *SCA, SDA, SGA*, Figuur 4.7, is een volledige entiteit die we wel degelijk werkend hebben mogen aanschouwen. De kern van de architectuur is de Software Execution Agent (SEA) bestaande uit drie componenten die de uitvoering van een bepaalde taak verzorgen. De Service Composition Agent (SCA) kan een volledig automatische compositie van services samenstellen om een taak uit te voeren waarvoor geen afzonderlijke Web service bestaat. De Service Discovery Agent (SDA) zal op zoek gaan naar meer Web services die voldoen aan een specifiek profiel. Ten slotte is er de Service Context Agent of Generic Context System (SGA) die bijkomende informatie verzamelt en bijhoudt (kwaliteit van de service, beschikbaarheid,...) over gekende services.



Figuur 4.7: SEA

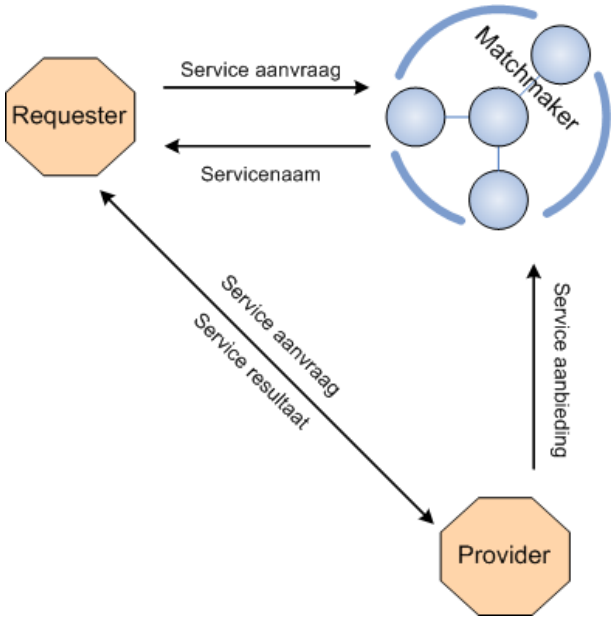
4.4.2 Matchmaker

De Matchmaker is veel beperkter in functionaliteit dan de Broker. Zoals op Figuur 4.8 te zien is, zal deze op een vraag van een gebruiker enkel antwoorden met de locatie van een geschikte Web service. Het oproepen en uitvoeren van de service wordt aan de client overgelaten. Dit heeft als voordeel dat de Matchmaker veel minder belast wordt dan een Broker, geen single point of failure vormt en ook eenvoudiger in elkaar zit. De Matchmaker kan bijvoorbeeld gebruikt worden als onderdeel in een systeem voor semi-automatische compositie.

Eigenschappen

Een volledige Matchmaker zal beschikken over de volgende eigenschappen:

- Aankondigingen in lokale knowledge base
- Discovery van service-aanbieders
- Fouttolerantie voor gedistribueerde systemen

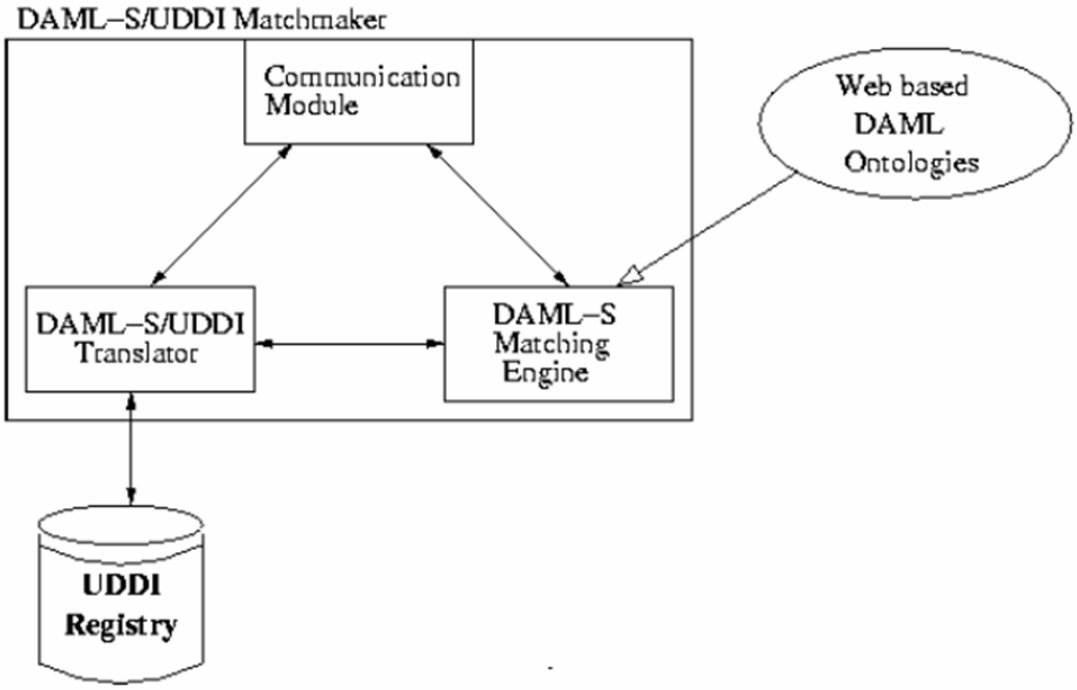


Figuur 4.8: Matchmaker als Middle Agent

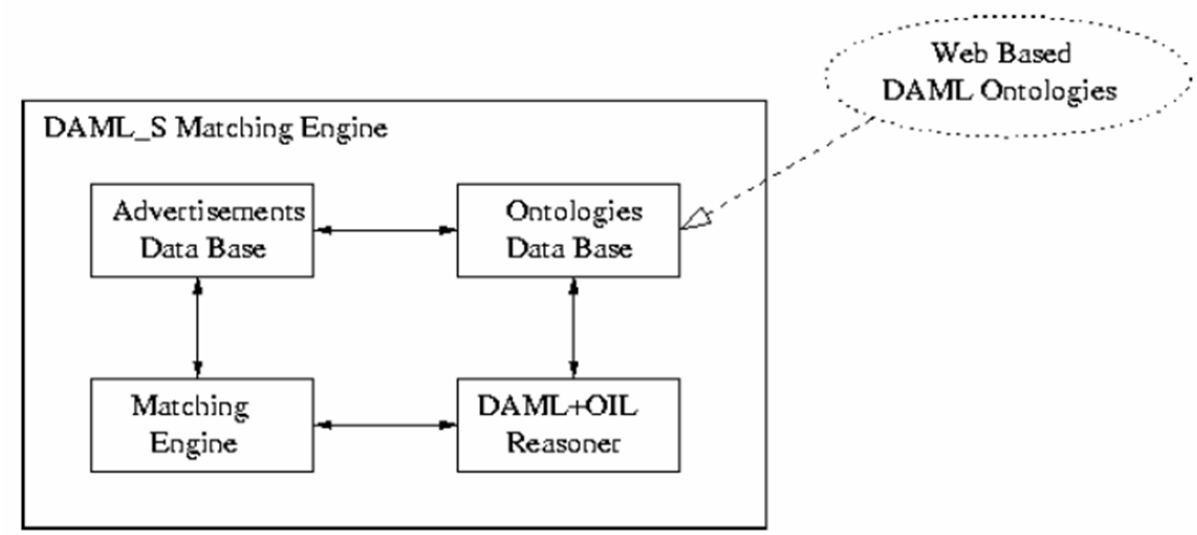
- Privacy
- Semi-automatische compositie

Voorbeelden van Matchmakers

We konden slechts één Matchmaker vinden, de *OWL-S Matchmaker* [20] (Figuur 4.9). Deze beschikt over een Matching Engine (Figuur 4.10) om een service te kunnen vinden overeenstemmend met een gevraagd profiel. De services zelf moeten geregistreerd worden in een UDDI registry en er kan ook nog gebruik gemaakt worden van de Matchmaker Client om aanvragen te formuleren.



Figuur 4.9: DAML-S/UDDI Matchmaker



Figuur 4.10: DAML-S Matching Engine

Hoofdstuk 5

Medische Web Service Composities

5.1 Doel: Applicatie voor artsen

Voor onze thesis ontwerpen we een applicatie die de artsen toelaat om zelf snel nieuwe services te bouwen. Dit gebeurt aan de hand van de beschikbare gegevens over de patiënt en de services die reeds aanwezig zijn in het systeem. De arts bepaalt de workflow die de service moet volgen, hierbij geholpen door de applicatie die de nodige bouwblokken suggereert. Dat is mogelijk doordat elke service vergezeld is van een semantische beschrijving, waardoor een reasoner *automatisch* kan bepalen welke agents aan elkaar kunnen gekoppeld worden. Het spreekt voor zich dat deze applicatie zo eenvoudig mogelijk moet zijn in het gebruik. In een later stadium zal ze gebruikt worden als basis voor een programma dat volledig *dynamisch* composities kan maken en uitvoeren.

5.2 Oplossing: Web Service Composer (Mindswap)

Om niet volledig vanaf nul te moeten beginnen, vertrekken we van een bestaand programma: de Web Service Composer van Mindswap, besproken in Sectie 4.3. Deze applicatie biedt een basisfunctionaliteit en is tevens flexibel genoeg om uit te breiden. Een ander pluspunt is de beschikbaarheid van de code. In deze sectie leggen we kort de werking ervan uit, vervolgens leggen we de tekortkomingen bloot die we trachten te verhelpen met onze uitbreidingen.

5.2.1 Werking

Zoals de naam al laat raden, is de Composer gericht op *semi-automatische compositie* van Web services. Bij het opstarten van het programma worden de OWL-S beschrijvingen van de reeds bestaande Web services ingeladen. Als eerste stap is het mogelijk de Web service te selecteren die het uiteindelijke doel is. Vervolgens zal de Composer voor elke input van die Web service nagaan welke andere services een output hebben van hetzelfde type. Uit al de mogelijke services kan de gebruiker dan een keuze maken. Voor de gekozen service wordt dan opnieuw hetzelfde principe toegepast, zodat er uiteindelijk een boomstructuur verkregen wordt van inputs die gekoppeld zijn aan outputs op een lager niveau. In Figuur 4.3 kan men een voorbeeld zien van zo een boomstructuur.
Wanneer de compositie volledig gedefinieerd is, biedt de Composer ook de mogelijkheid om deze uit te voeren. De gebruiker hoeft hiervoor niets meer te doen, alle services worden in de juiste volgorde uitgevoerd en aan elkaar gekoppeld.
Tot slot is het ook mogelijk om de compositie op te slaan, zodat ze later kan hergebruikt worden. Het bewaren gebeurt onder de vorm van een OWL-S beschrijving van de samengestelde Web service.

Samenvattend heeft de Composer de volgende kenmerken:

- Inladen van de beschikbare Web services,
- Filteren op basis van extra eigenschappen van de Web services,
- Semi-automatische compositie met tussenkomst van de gebruiker op basis van achterwaarts redeneren vanuit het doel,
- Opslaan van de compositie,
- Inladen van de opgeslagen compositie,
- Genereren van een OWL-S beschrijving op basis van de WSDL beschrijving van een Web service,
- Uitvoeren van de compositie.

5.2.2 Nadelen

Helaas zijn er ook enkele nadelen aan de Composer. We sommen ze hier kort op:

1. De Composer werkt op basis van semi-automatische compositie waardoor de gebruiker telkens gedwongen wordt om bij elke knoop een service te kiezen. Dit leidt tot grote tijdsverliezen en overhead. In het beste geval zou de Composer in staat moeten zijn om automatisch een volledige boomstructuur te genereren waarbij de gebruiker de mogelijkheid krijgt om aanpassingen aan te brengen.

2. Bij het uitvoeren van een compositie heeft men geen rekening gehouden met het feit dat sommige services onbeschikbaar kunnen blijken. Men zou dan op dynamische wijze alternatieven kunnen zoeken.

3. De seriële oproep van de services leidt tot performantieverlies waardoor het uitvoeren van de compositie langer duurt dan strikt noodzakelijk.

4. Er kunnen gevallen voorkomen waarbij Web services meerdere keren voorkomen in de compositie. De Composer is niet in staat om bekomen tussenresultaten te hergebruiken.

5. De beschikbare Web services worden bij het opstarten van de applicatie ingelezen uit een tekstbestand. Dit is niet handig op het vlak van schaalbaarheid en uitbreidbaarheid; tevens is het niet mogelijk om *at runtime* Web services toe te voegen, te verwijderen of te veranderen.

6. De gebruiker kan niet beschikken over flow of controlestructuren (bv. een 'als-dan' constructie). Het gaat zuiver om het lineair aan elkaar koppelen van outputs en inputs. Dit legt serieuze beperkingen op aan de mogelijke toepassingen en agents die kunnen worden gemaakt.

7. De GUI is vrij Spartaans. Het belang van een intuïtieve en ergonomische gebruikersinterface mag niet onderschat worden, zeker niet voor mensen die geen computeropleiding achter de rug hebben.

5.2.3 Uitbreidingen van de Web Service Composer

Onze uitbreidingen pogen, naast het toevoegen van extra functionaliteit, een oplossing te bieden aan bovenstaande tekortkomingen (tussen haakjes vermelden we de tekortkoming uit vorige sectie waaraan deze uitbreidingen tegemoet komen):

- De Matchmaker breiden we uit om een meer nauwkeurige score te geven aan de **ontologiematch**.
- De applicatie dient als basis voor een volledig **automatische composer**. Die is in staat om, zuiver op basis van gekende gegevens en services, zonder menselijke hulp een compositie te maken die een vooropgesteld doel vervult. De arts is in staat om de bekomen boomstructuur verder te tunen naar zijn eigen wensen. (1)
- Verschillende **algoritmen** zijn beschikbaar om de automatische compositie te verwezenlijken. (1)
- De Composer is uitgebreid om volledig **dynamische compositie** te ondersteunen. Daarbij denken we niet alleen aan de automatische compositie, maar ook aan **runtime compositie** (op basis van de beschikbaarheid van de Web services) waarbij eventueel andere paden of services moeten gevonden worden om een taak uit te voeren. (2)
- Aangezien het om een boomstructuur gaat waarbij meerdere services tegelijkertijd kunnen uitgevoerd worden, is de seriële oproep uitgebreid naar een **parallelle compositie**. Zo kunnen meerdere services tegelijkertijd uitgevoerd worden wat de totale uitvoeringstijd serieus inkort. (3)
- Indien er services zijn die meerdere keren voorkomen in de compositie, kan de Composer bekomen resultaten **hergebruiken** in plaats van de service opnieuw uit te voeren. Dit leidt tot nog meer prestatiewinst, zeker bij een medische applicatie waarbij *reuse* aangemoedigd wordt. (4)
- **Gebruiksvriendelijkheid** is een criterium dat we bij elke stap en elke beslissing in het achterhoofd houden. (7)

Hoofdstuk 6

Ontwikkelproces

In dit hoofdstuk beschrijven we het ontwikkelproces van onze applicatie. De lezer kan zo stapsgewijs onze vooruitgang volgen en nagaan welke ontwerpsbeslissingen we genomen hebben.

6.1 Voorbereidend werk

6.1.1 Medische ontologie

Eerst en vooral hebben we als basis natuurlijk een medische ontologie nodig. Deze beschrijft de in- en uitvoer van de Web services op basis van OWL klassen en zal door de Composer gebruikt worden om de juiste match te maken. Figuur 6.1 toont een voorstelling van de gebruikte ontologie. Alle klassen erven over van de klasse 'Measurement'. Een aantal zijn verder specifieker uitgebreid.
Als editor hiervoor wordt Protégé gebruikt.

Figuur 6.1: Medische OWL ontologie

6.1.2 Dummy services

Om onze uitbreidingen aan de Composer op een gecontroleerde manier te kunnen testen, hebben we nood aan een uitgebreide verzameling Web services. Appendix C bevat meer uitleg bij de gebruikte medische use case. In het vervolg van de scriptie zullen wij deze aanduiden als MODS[21]. Om onze aandacht te kunnen vestigen op de ontwikkeling van de Composer, hebben we deze eenvoudig gehouden: ze bevatten geen implementatie van het gewenste gedrag maar geven een vaste waarde terug. Vanuit het standpunt van de compositie zijn deze Web services immers toch blackboxes.
De Web services zelf ontwikkelen we in Netbeans 6. Hierbij wordt gebruik gemaakt van JAX-WS. Helaas zit hier een vervelende bug in: binnen eenzelfde package mag een methodenaam namelijk maar één keer voorkomen. Eens je dit weet en er rekening mee houdt, kan je echter snel nieuwe services ontwerpen.
De OWL-S beschrijving van de service laten we door de Composer automatisch genereren op basis van de WSDL. Aan dit skelet hoeven we slechts twee dingen aan te passen. Enerzijds voegen we een XSL transformatie toe die instaat voor de omzetten van in- en uitvoer van de Web service naar de correcte ontologie. Anderzijds importeren we de medische ontologie, dit is nodig voor het correct kunnen herkennen van subklassen en dergelijke.

6.1.3 Tegengekomen problemen

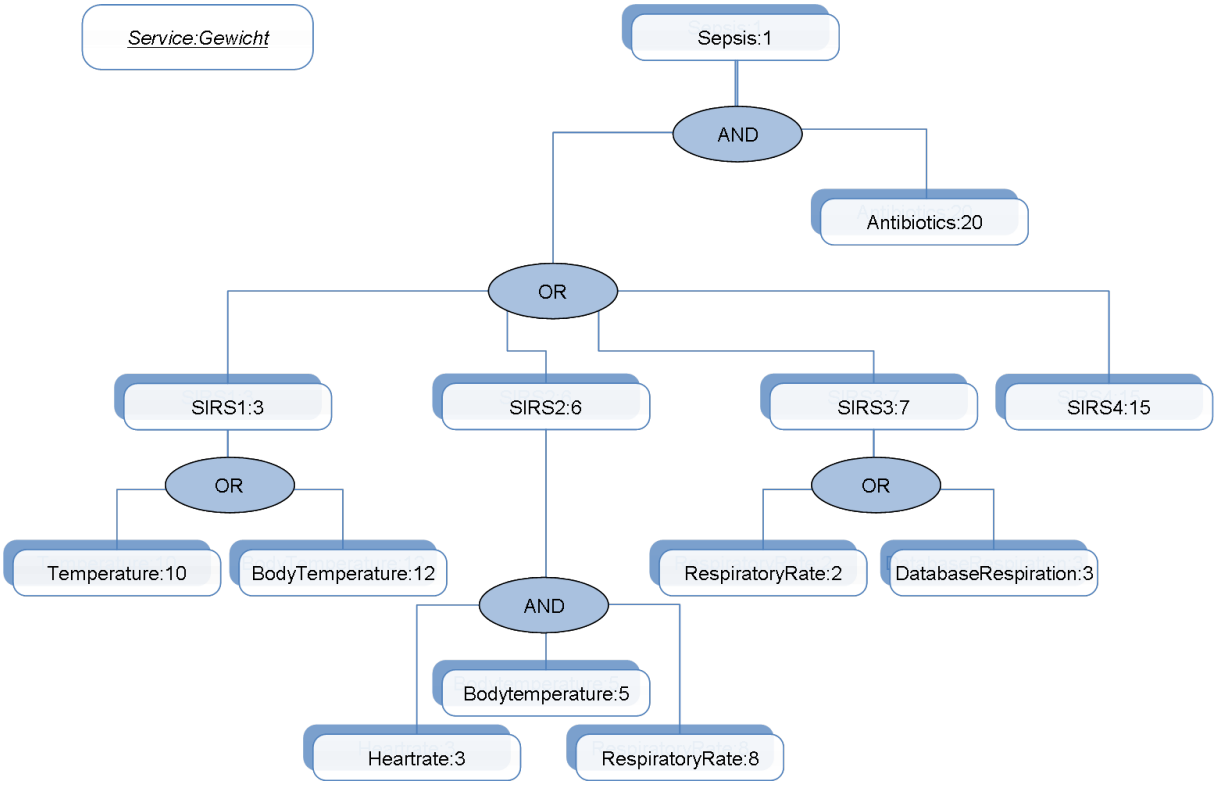
Bij het testen van onze dummies ontdekken we dat de Composer niet overweg kan met services zonder input. Deze bug lossen we op alvorens verder te gaan.

6.2 Automatische compositie

Om de complexiteit van de automatische compositie te beheersen, nemen we dit aan in verschillende stappen. Om te beginnen bedenken we een (geoptimaliseerd) algoritme dat op basis van een service de oplossingsboom kan genereren. Vervolgens implementeren we dit algoritme los van de Composer om de correctheid ervan te testen. Tot slot combineren we dit algoritme met de Composer en doen we prestatietesten (zie Hoofdstuk 7) om de invloed van de optimalisatie te testen.

6.2.1 Probleemstelling

Om de gedachten te vestigen, beschrijven we eerst het probleem. Gegeven een service met een aantal inputs, willen we andere services vinden die deze inputs kunnen genereren. Deze gevonden services zullen op zich ook weer inputs nodig hebben, zodat we met een recursief probleem te maken hebben. Dit gaat zo verder tot we komen aan een "basisservice-die geen inputs nodig heeft of waarvoor we de invoer bij de gebruiker halen.
We merken op dat we dit probleem kunnen modelleren aan de hand van een *AND/OR-graaf* (zie Figuur 6.2). De inputs komen overeen met een AND-knoop, aangezien voor een service alle



Figuur 6.2: Voorbeeld van een AND/OR-graaf

inputs beschikbaar moeten zijn. De services corresponderen dan met de OR-knopen, aangezien er typisch meer dan één Web service zal zijn die de nodige input kan leveren.
Om *één* oplossing voor het compositieprobleem te vinden, volstaat het om bij elke OR-node één service uit te kiezen. We willen echter de mogelijkheid om aan de arts meteen de *beste* oplossing voor te stellen.

6.2.2 Ontwerpkeuzes

De 'beste' oplossing

Om een keuze te kunnen maken welke nu de beste oplossing is, kennen we aan elke service een gewicht toe, waarbij een *lager gewicht* staat voor een *betere service*. Dit laat toe het algoritme generiek te maken zodat het enkel afhangt van het gewicht. Hoe het gewicht bepaalt wordt (m.a.w. wat de definitie is van een goede service) kan dan vrij eenvoudig aangepast worden. Voorbeelden hiervan zijn: onderscheid tussen blad en interne knoop, soort match (exact, subsume), andere QoS-parameters (uitvoeringstijd, CPU belasting van de server, aantal simultane gebruikers, netwerktopologie,...) of de prijs. We hebben er voor gekozen om rekening te houden met twee soorten gewichten, nl. uitvoeringstijd en prijs van een Web service.
Wanneer aan elke service een gewicht is toegekend, zijn er nog verschillende manieren om te bepalen welke de beste boom is. Hierbij denken we aan minimaal totaal gewicht, minimale breedte of minimaal kritisch pad (waarbij de lengte van het langste pad minimaal is). We implementeren de eerste optie. Deze lijkt ons het meest toegepast en meest algemeen. Op die manier kunnen wij verschillende algoritmes ontwikkelen die volgens dit principe werken.

Algoritme

```
begin maakBoom startService, taboelijst
    voeg het gewicht van de service toe aan het totaal gewicht
    foreach input ∈ startService do
        voeg toe aan de taboelijst
        bepaal matching services (niet in taboelijst)
        foreach match ∈ input do
            maakBoom matchService, taboelijst
            if gewicht subboom < gewicht beste subboom then
                nieuwe beste pad
    return gewicht subboom
end
```

Algoritme 1: Niet-optimaal algoritme

Nu we weten wat de beste oplossing is, kunnen we een algoritme construeren dat deze oplossing bepaalt. We beginnen met een naïef maar duur (in rekentijd) algoritme, dat we daarna optimaliseren. Dit laat toe om later de correctheid van het geoptimaliseerd algoritme na te gaan

en benchmarks uit te voeren zodat we de snelheidswinst kunnen verifiëren.

Naïeve oplossing

Het naïef algoritme (Algoritme 1) komt in principe neer op een *depth-first search* van de oplossingsboom. Bij een OR-node zal van alle mogelijke subbomen het gewicht één na één bepaald worden en de kleinste subboom zal weerhouden worden. Hierdoor zal dus de volledige oplossingsboom doorlopen worden, zelfs als je in een bepaalde subboom al een hoger gewicht hebt dan je huidige beste subboom.

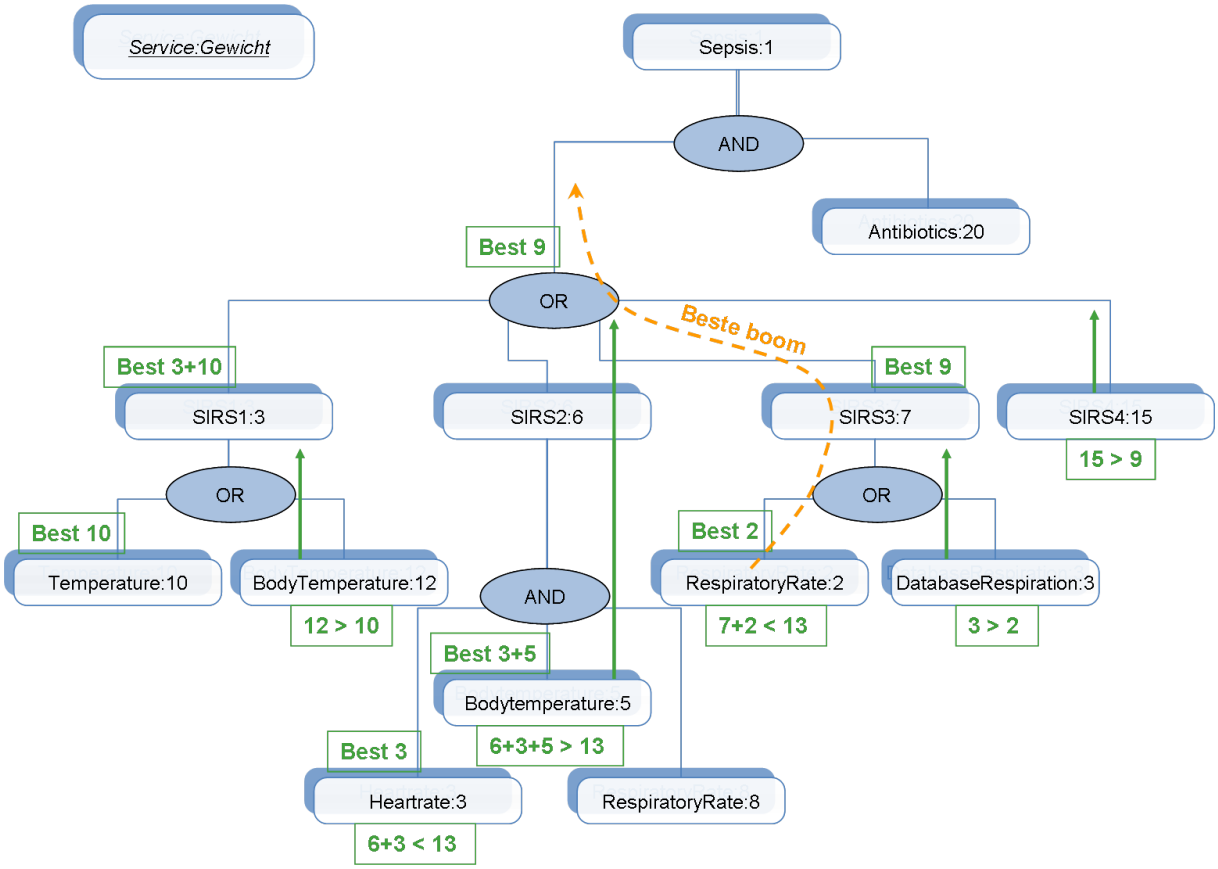
begin maakBoom $startService, taboelijst, gewichtBestePad$
voeg het gewicht van de service toe aan het totaal gewicht
switch *startService* **do**
case *gewicht subboom* $>$ *gewichtBestePad*
verlaat dit pad
case *eindknoop*
return gewicht
otherwise
foreach *input* $\in$ *startService* **do**
voeg toe aan de taboelijst
bepaal matching services (niet in taboelijst)
rangschik matching services op min gewicht
foreach *match* $\in$ *input* **do**
if *gewicht matchService* $>$ *gewichtBestePad* **then**
verlaat dit pad
maakBoom $matchService, taboelijst, gewichtBestePadh$
if *gewicht subboom* $<$ *gewicht beste subboom* **then**
nieuwe beste pad
else
probeer volgende matchService
return *gewicht subboom*
end

Algoritme 2: Optimaal algoritme

Optimalisatie

Het basisidee achter het geoptimaliseerd algoritme (Algoritme 2) is om zo snel mogelijk te detecteren dat je een slechtere oplossing aan het bekijken bent en daar dan ook mee te stoppen. In elke OR-knoop rangschikken we de mogelijke services naar oplopend gewicht. Daarnaast houden we voor een OR-knoop het gewicht bij van de beste subboom tot dan toe. Een eerste

optimalisatie bestaat erin na te gaan of het gewicht van de volgende service (de wortel van de volgende subboom) groter is dan dat van de beste subboom. Indien dit zo is, weten we (dankzij de rangschikking) dat we in deze OR-knoop geen betere subboom meer zullen vinden. Een tweede optimalisatie bereiken we door tijdens het opbouwen van een subboom het cumulatieve gewicht daarvan bij te houden. Als dit groter wordt dan het gewicht van de beste subboom, kunnen we deze mogelijkheid schrappen en overgaan naar de volgende subboom. Door deze twee optimalisaties hopen we vooral bij ingewikkelde oplossingsbomen rekenwinst te kunnen boeken (zie ook Figuur 6.3). Voor de resultaten van de prestatietesten verwijzen we de lezer naar Hoofdstuk 7.



Figuur 6.3: Werking van het optimaal algoritme

Na het algoritme los van de Composer getest te hebben, is het met succes verwerkt in het programma.

6.2.3 Tegengekomen problemen

Bij het opstellen van de algoritmes doken volgende problemen op:

- Er is de mogelijkheid tot oneindige lussen, in het geval dat een herhaling van een service voorkomt op een gegeven pad of wanneer een service hetzelfde type input heeft als zijn output. De oplossing die we hiervoor bedacht hebben, is het gebruik van een taboe-lijst. Op deze lijst plaatsen we alle inputs die reeds gebruikt zijn in een bepaalde subboom. Indien een kandidaat-service (lager in de subboom) deze input ook nodig heeft, wordt hij geschrapt en zal hij niet kunnen gebruikt worden binnen deze subboom. Het idee hierachter is dat ons probleem eigenlijk kan gemodelleerd worden als een lijst van inputs die moeten gerealiseerd worden. Wanneer een service een bepaalde input realiseert, maar hiervoor een input nodig heeft die hogerop in de boom ook al nodig was, zal hij het probleem niet verkleinen (doorgaans zal hij het zelfs vergroten als die service meer dan één input heeft).
- Er komen problemen zonder oplossingen voor, wanneer er voor een bepaalde subboom geen leafs (services zonder input) gevonden worden. We detecteren dit en vragen in dit geval naar user input. De boom wordt wel zo diep mogelijk opgebouwd.

6.2.4 Mogelijke uitbreidingen

Er kunnen extra algoritmes ontwikkeld worden die andere QoS parameters optimaliseren of eventueel rekening houden met meerdere QoS parameters tegelijkertijd (zie Sectie 6.5).

6.3 Dynamische compositie

Met de automatische compositie in werking, is een logische volgende stap *dynamische* compositie. De opzet hiervan is om tijdens de uitvoering van een compositie te detecteren dat een nodige service niet meer beschikbaar is. Er zal dan at runtime gezocht worden naar een nieuwe compositie om de uitgevallen service te vervangen. Meteen is ook duidelijk waarom we eerst automatische compositie dienden te realiseren.

Om de dynamische compositie te implementeren, moeten we de 'ProcessExecutionEngine' (OWL-S API) aanpassen. Deze behoort eigenlijk niet meer tot de Composer zelf, maar tot de OWL-S API, een library waarvan de Composer gebruik maakt. Om de zaken niet teveel door elkaar te halen, besluiten we een 'CustomProcessExecutionEngine' aan te maken die overerft

van de hierboven vermelde Engine. Wanneer er een 'ServiceNotAvailableException' opgegooid wordt, vangen we deze op en zetten de recovery procedure in gang:

- Op basis van de benodigde input wordt een nieuwe boom aangemaakt van services die tot deze input leiden;
- Vervolgens zetten we deze boom om naar een OWL-S beschrijving;
- Tot slot voeren we deze boom uit en geven het resultaat door alsof het afkomstig is van de gecrashte service.

We hebben daarnaast functionaliteit voorzien om de herstelprocedure efficiënter te laten verlopen. Wanneer er gemeenschappelijke stukken zijn tussen de herstelboom en de boom onder de gecrashte service, kunnen de bestaande resultaten hergebruikt worden. De services zullen met andere woorden niet dubbel uitgevoerd worden. Deze functionaliteit hebben we later ook voorzien bij normale uitvoering: wanneer in een boom verschillende services dezelfde input nodig hebben, zal deze ook hergebruikt worden.

6.3.1 Tegengekomen problemen

Bij de realisatie van de recovery procedure hebben we wat problemen gehad. Boosdoener bleek een 'HashMap' met tussenresultaten die bij het uitvoeren van de recovery gereset werd, waardoor deze resultaten hopeloos verloren waren.

6.3.2 Mogelijke uitbreidingen

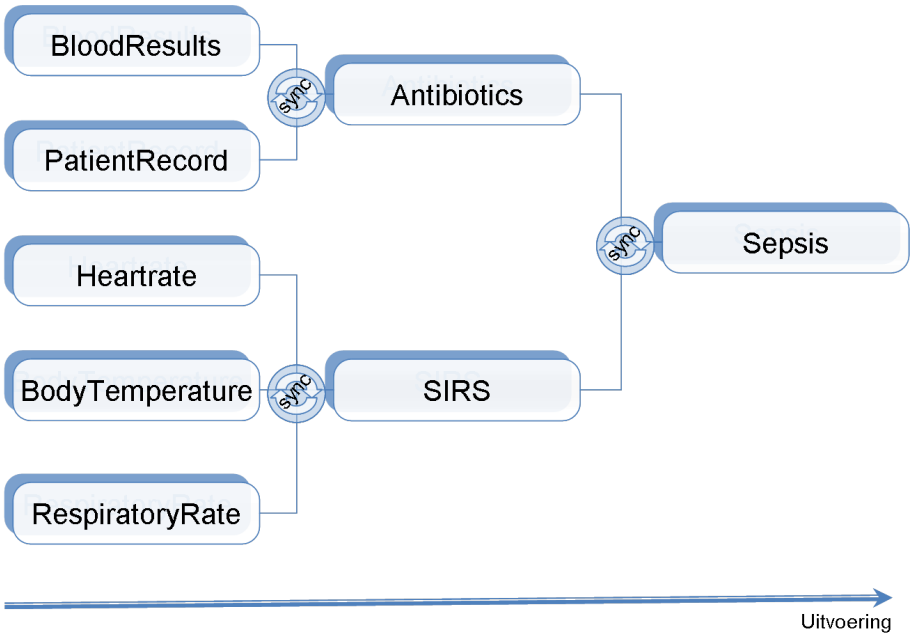
- De recovery boom tonen aan de gebruiker zodat die ook getuned kan worden.
- Indien user input nodig is, de uitvoering pauzeren en extra informatie vragen aan de gebruiker.

6.4 Parallelle uitvoering

De oorspronkelijke composer maakt gebruik van seriële uitvoering. Wanneer de compositieboom opgebouwd is, wordt deze diepte-eerst doorlopen en worden alle services op een rijtje

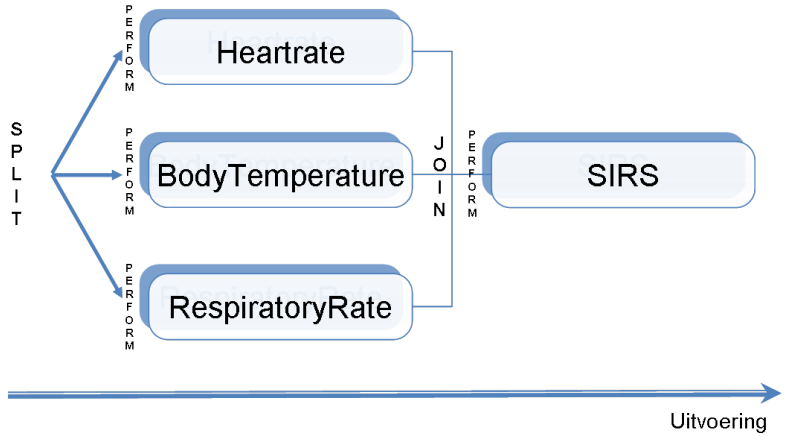
gezet. Vervolgens worden deze services één na één uitgevoerd. Dat werkt, alleen legt dit een hypotheek op de uitvoeringssnelheid.

Stel dat je in je boom een bepaalde service hebt met verschillende inputs. De subbomen van al deze inputs zijn eigenlijk onafhankelijk van elkaar (er is geen koppeling van outputs naar inputs). Dit bracht ons op het idee om deze verschillende subbomen parallel uit te voeren (zie Figuur 6.4), aangezien dit een behoorlijke snelheidswinst kan opleveren. Tevens geeft het aanleiding tot nieuwe en interessante QoS-algoritmes (zie Sectie 6.5).



Figuur 6.4: Parallelle uitvoering van de takken van een boom

Het voordeel is dat we dit vrij eenvoudig kunnen realiseren, er hoeft enkel een andere OWL-S beschrijving gegenereerd te worden. Om een bepaalde service uit te voeren, moeten we eerst alle inputs weten. Dit komt neer op alle subbomen parallel uitvoeren en synchroniseren (alle inputs moeten klaarstaan). Dit is net de functionaliteit aangeboden door de Split/Join construct uit OWL-S. Je krijgt dus een Sequence van eerst een Split/Join om aan je inputs te geraken en vervolgens een Perform om je eigenlijke service uit te voeren (zie Figuur 6.5). We hoeven dan enkel deze nieuwe beschrijving nog te laten uitvoeren door de OWL-S API en onze parallelle uitvoering is gerealiseerd.



Figuur 6.5: Split/Join van een samengestelde service

6.4.1 Tegengekomen problemen

Helaas zit er nog ergens een bug in de parallelle uitvoering waardoor deze soms wel en soms niet een exceptie geeft. Het probleem lijkt zich vooral voor te doen op machines met meer dan één core. De oorzaken zijn verschillend. Wij zijn al tot de vaststelling gekomen dat een aantal libraries (Jena, Pellet) parallelle uitvoering niet ondersteunen (niet thread safe). Verder is het zo, dat als de uitvoering te lang duurt (door te veel gestarte services) de Java library zelf een time-out genereert bij de threads. Deze worden dan als dood beschouwd waarna Java verder gaat met de uitvoering.

6.4.2 Mogelijke uitbreidingen

De volgende uitbreidingen zijn mogelijke oplossingen voor de hierboven beschreven problemen.

- Beperkte parallelle uitvoering zodat minder threads gestart worden.
- Het gebruik van de libraries die niet thread safe zijn synchroniseren om problemen te vermijden.
- Onderzoeken of de nieuwe versies van de libraries wel thread safe zijn en ze integreren in de Composer.

6.5 Algoritmes

Na het implementeren van de dynamische compositie was het voor ons interessant om een aantal extra algoritmes te ontwikkelen. We hadden al veel ideeën verzameld mede dankzij de parallelle mogelijkheden van de Composer. Daarnaast was het nuttig om de prestatie van het programma te testen. De resultaten van de testen zullen we in Hoofdstuk 7 in detail bespreken. Door het medisch karakter van de applicatie zullen eindknopen van de compositie meestal databankoperaties uitvoeren. Daardoor zullen de algoritmes services die user-input vereisen links laten liggen. Uitzonderingen daarbij zijn de Local First en Local Random algoritmes.

6.5.1 Lokale algoritmes

Naast de globale algoritmes die een algemene beste boom kunnen genereren, hebben we een aantal lokale algoritmes (Algoritme 3) ontwikkeld. De reden daarvoor is het feit dat ze makkelijk te implementeren zijn en een relatief goede boom kunnen genereren in een redelijke tijd. Dit terwijl de globale algoritmes vaak niet meer performant zijn bij grote bomen en/of een overvloed aan alternatieve OR-knopen.

begin maakBoom $startService, taboelijst$
  voeg het gewicht van de service toe aan het totaal gewicht
  foreach *input* $\in$ *startService* **do**
    voeg toe aan de taboelijst
    bepaal matching services (niet in taboelijst)
    neem eerste/random/goedkoopste/snelste matchService
    maakBoom $matchService, taboelijst$
  return *gewicht subboom*
end

Algoritme 3: Lokaal algoritme

Local First

Het Local First algoritme zal telkens de eerste service kiezen uit de lijst van OR-services. Deze kunnen totaal random geordend staan of op voorhand geordend worden volgens een welbepaald criterium (prijs, uitvoeringstijd, alfabetisch, ouderdom, etc).

Local Random

Het Local Random algoritme zal, zoals de naam het al zegt, een volledig random OR-service uitkiezen.

Local Minimal Weight

De Local Minimal Weight algoritmes zullen de lijst met OR-services eerst ordenen volgens een welbepaald criterium en daaruit de beste service uitkiezen. In ons geval hebben we gekozen voor twee mogelijkheden:

- minimale uitvoeringstijd
- minimale kost

Daarnaast zal het algoritme ernaar streven om services die user input vereisen te negeren. Dit zorgt voor een tijdsverlies tov. de Local First en Local Random algoritmes.

6.5.2 Globale algoritmes

De globale algoritmes zullen rekening houden met de volledige boomstructuur bij het opbouwen van de compositie. Zoals vermeld in Sectie 6.2.2 hebben we twee algoritmes uitgewerkt om onze boom automatisch te kunnen opbouwen. We hebben deze optimale en niet-optimale algoritmes verder uitgebreid en daaruit volgend een extra globaal algoritme uitgewerkt.

Minimal weight optimal

Het optimale algoritme (Algoritme 2) dat een minimale boom opbouwt, is in staat rekening te houden met het feit dat services meerdere gewichten kunnen hebben. Dit zal net als vroeger een minimale boom opbouwen, maar deze keer afhankelijk van het gewenste gewicht. We hebben een indeling gemaakt op basis van twee gewichten:

- minimale uitvoeringstijd
- minimale kost

Uitbreiding naar andere gewichten is slechts een kwestie van extra gewichten toe te voegen en te definiëren met welke het algoritme moet werken.

Minimal weight not optimal

Het niet-optimale algoritme (Algoritme 1) is opgesplitst in twee versies. De eerste versie zal, zoals het optimale algoritme, het volledige boomgewicht beschouwen alsof de compositie serieel uitgevoerd zal worden. De tweede versie is gekoppeld aan de mogelijkheid om de samengestelde service parallel uit te voeren. Deze zal dan ook enkel rekening houden met het gewicht van het traagste/duurste pad.

Indeling op basis van de gewichten:

- minimale uitvoeringstijd
- minimale kost

Indeling op basis van de uitvoeringswijze:

- serieel
- parallel

De opsplitsing was een eerste stap naar een meer ingewikkeld algoritme, namelijk Tune time-cost.

Tune time-cost

Dit algoritme (Algoritme 4) vertrekt van de parallelle niet-optimale versie. Eerst worden alle mogelijke bomen en gewichten uitgerekend waarbij men enkel het traagste pad in een boom onthoudt in plaats van de volledige boomstructuur. Op basis daarvan kiest men de snelste boom. Van deze boom is het kritisch pad (traagste pad) van belang. Men zal de andere takken tunen op basis van dit gewicht. De bedoeling is dat paden die sneller zijn eventueel vervangen kunnen worden door paden die trager (nog steeds minstens even snel als het traagste pad) maar goedkoper zijn.

6.5.3 Tegengekomen problemen

- Het Tune time-cost algoritme zal niet altijd een betere oplossing vinden dan de gewone niet-optimale versie. Er zijn verschillende redenen daarvoor:

```
begin maakBoom startService, taboelijst
    voeg het gewicht van de service toe aan het totaal gewicht
    foreach input ∈ startService do
        voeg toe aan de taboelijst
        bepaal matching services (niet in taboelijst)
        foreach match ∈ input do
            maakBoom matchService, taboelijst
            if time subboom < time beste subboom then
                nieuwe beste pad
    foreach input ∈ matchService do
        if cost alternatief subboom < cost gekozen subboom &
        time alternatief subboom <= time kritisch pad then
            kies alternatief subboom
    return gewicht subboom
end
```

Algoritme 4: Tune time-cost algoritme

- Te kleine boom of te weinig alternatieve services.
- Te kleine spreiding van de gewichten waardoor er geen betere alternatieven kunnen gevonden worden.
- Brede maar ondiepe boom. De breedte van een boom heeft geen invloed op het tuning proces maar de diepte wel. In een diepe boom is de kans groter dat men kleine services/bomen kan vervangen. Er is meer speelruimte.
- De niet-kritische bomen kunnen zeer klein uitkomen (tot één service) en met een beetje pech kan men geen goedkopere subboom vinden. Dit kan gebeuren in gevallen waar bij een OR-knoop services met en zonder inputs voorkomen. Dan is de kans groter dat men een service kiest zonder inputs, anders zou men een subboom moeten opbouwen die bijna altijd duurder zal uitkomen (als de gewichten een beperkte spreiding vertonen).

6.5.4 Mogelijke uitbreidingen

- Een optimaal algoritme dat een suboptimale boom (qua uitvoeringstijd, kost,...) opbouwt. Dit kan een soort afweging zijn tussen beste boom en snelheid.
- Het cachen van bomen en eventueel subbomen zodat bij dezelfde kenmerken/hetzelfde algoritme deze niet telkens van nul opgebouwd moeten worden. Dit kan zorgen voor

een serieuze tijdswinst. Hierbij kan men gebruik maken van de bestaande mogelijkheid om composities op te slaan. Het opslaan zal achter de schermen moeten gebeuren, met alle nodige informatie omtrent de boom (gebruikte algoritme, gewicht, etc). Verder zal de mogelijkheid voorzien moeten worden om de opgeslagen compositie volledig te kunnen tonen en tunen, aangezien momenteel enkel een samengestelde statische service ingeladen kan worden (voorgesteld door 1 blokje in de Composer).

- Men kan algoritmen ontwikkelen die rekening houden met de netwerktopologie en de load op de servers waar de Web services uitgevoerd worden. Deze algoritmes zullen load balancing moeten ondersteunen.

6.6 Bekomen architectuur

De uiteindelijke high level architectuur is te zien in Figuur 6.6. De belangrijkste component is de **Web Service Composer** zelf. Deze communiceert met de drie andere modules, namelijk :

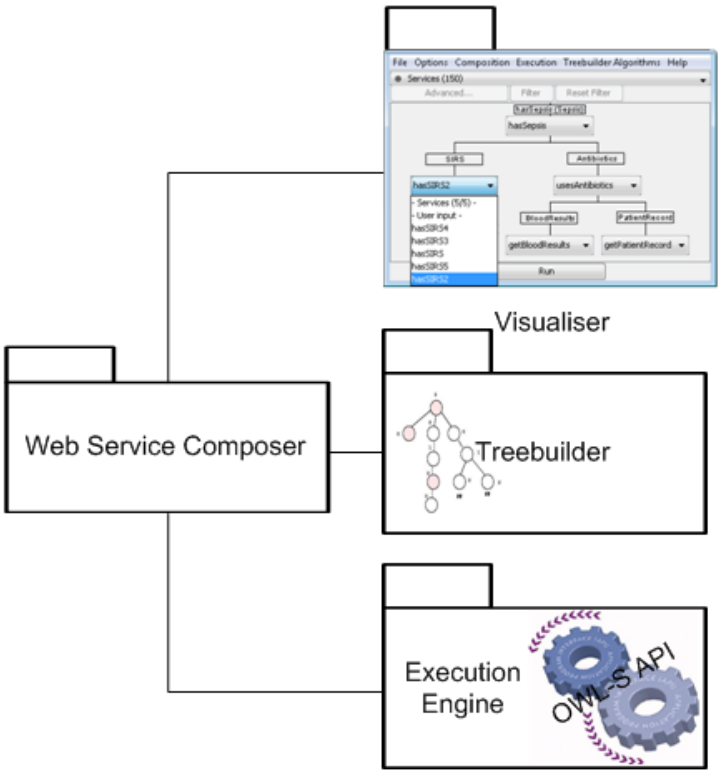
1. Een **Visualiser** zal een voorstelling geven van de compositie (zie Gebruikershandleiding in Appendix D). Daar kan de gebruiker verschillende instellingen kiezen zoals: *algoritme* om de boom op te bouwen, soort *compositie* (semi-automatisch, automatisch, dynamisch), manier van *uitvoering* (serieel, parallel), etc.

2. De **Treebuilder** omvat de beschikbare algoritmes om een compositie op de bouwen.

3. De **Execution Engine** van de OWL-S API is verantwoordelijk voor de uitvoering van de samengestelde service. Daarin wordt ook de dynamische recovery verzorgd.

In Figuur 6.7 kan men een meer gedetailleerd beeld bekijken van de architectuur van de Web Service Composer. Enkel de belangrijkste modules/klassen/functies zijn toegevoegd. Aan de hand daarvan zullen we de werking van de Composer bespreken. Er moet even benadrukt worden dat alle aanpassingen in de Web Service Composer zelf uitgevoerd zijn. Klassen en functies gedefinieerd door de OWL-S API werden in de Composer zelf overgeërft en uitgebreid. De 'CustomProcessExecutionEngine' is daar een voorbeeld van. Deze erft over van de OWL-S API klasse 'ProcessExecutionEngineImpl'.

Bij het opstarten van de Composer worden eerst alle beschikbare services ingeladen in de 'OWLKnowledgeBase (1)'. De gebruiker krijgt de GUI te zien waarbij hij zelf de standaard



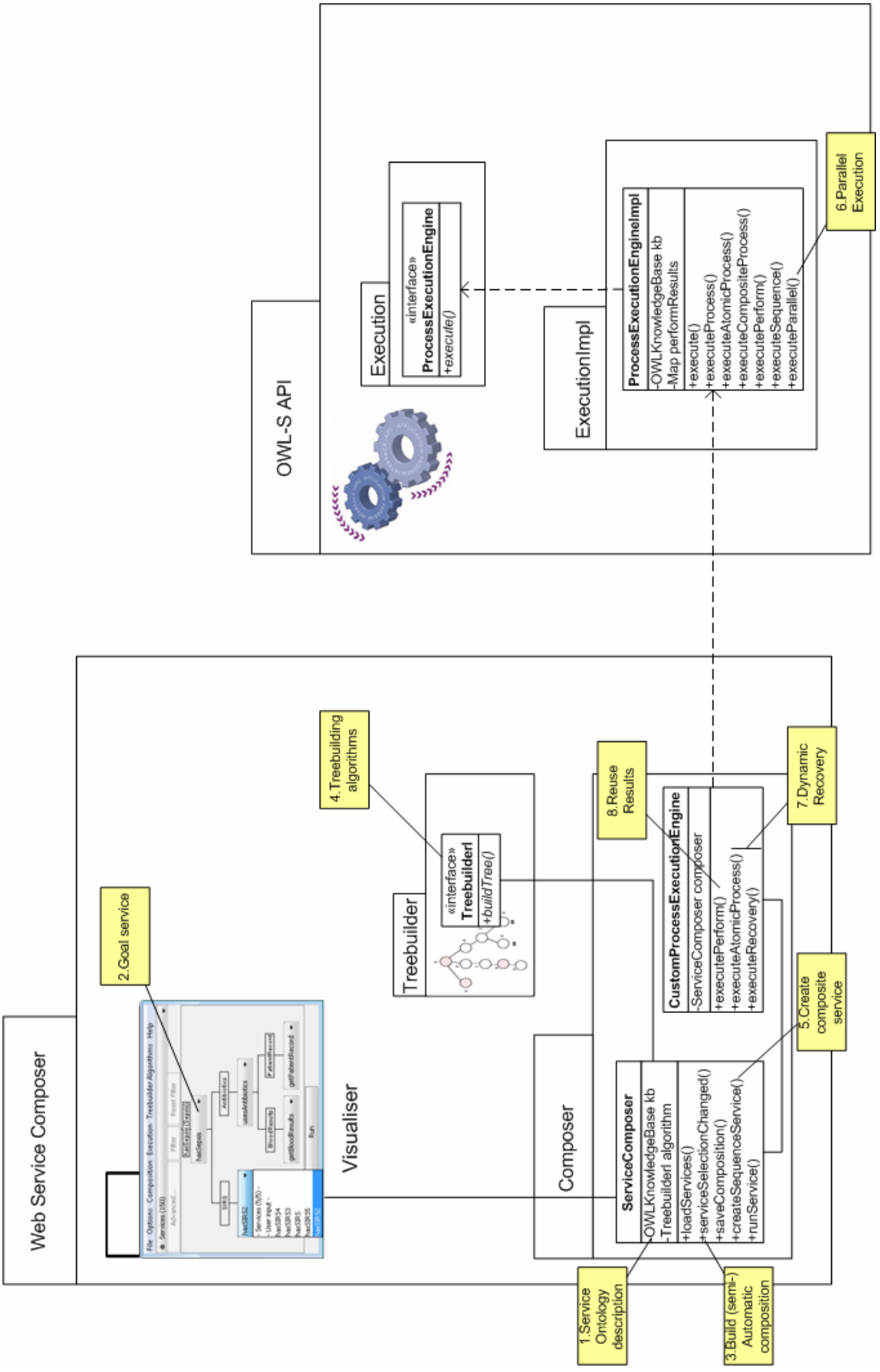
Figuur 6.6: High level architectuur

instellingen kan aanpassen. Uit de beschikbare services kan men vervolgens een gewenste service uitkiezen om uit te voeren, de 'Goal service (2)'. Indien de gebruiker voor *automatische compositie* heeft gekozen, hoeft hij niet meer tussen te komen. De Composer zal dan het gekozen algoritme oproepen via 'serviceSelectionChanged (3)' en de boom opbouwen ('buildTree (4)'). Deze zal vervolgens voorgesteld worden in de Visualiser. Indien gewenst kan de gebruiker dan manueel de boom tunen met services die hem beter aanstaan.

Als alles naar wens is, kan het uitvoeringsproces starten. De Composer zal vóór de uitvoering een samengestelde service aanmaken in 'createSequenceService (5)'. Afhankelijk van de instellingen zal daar een *seriële* of een *parallelle* OWL-S beschrijving opgebouwd worden.

Eens de samengestelde service aangemaakt is, kan de *OWL-S API* beginnen met de uitvoering ervan. De OWL-S beschrijving zal stap voor stap uit elkaar gehaald worden afhankelijk van de gebruikte controlestructuren. Hierbij is de 'executeParallel (6)' vooral van belang aangezien deze verschillende threads zal opstarten om subbomen *parallel* uit te voeren. Indien een service van de samengestelde boom onbeschikbaar blijkt te zijn, zal dit opgevangen worden door de 'CustomProcessExecutionEngine' die de *recovery* procedure zal starten. Men gaat op zoek naar alternatieve (samengestelde) services om de onbeschikbare service te vervangen.

Het spreekt voor zich dat hiervoor achter de schermen een nieuwe *automatische compositie* vanaf de gefaalde service gestart wordt door de Treebuilder. De gebruiker hoeft hier niets van te merken. Deze laatste stap verzorgt de *dynamische compositie*.
Een laatste aanpassing is de mogelijkheid om tussentijdse resultaten op te slaan. Deze kunnen nuttig zijn wanneer eenzelfde service meerdere keren in de compositie voorkomt of bij het falen van een service. Daarbij is de kans groot dat de alternatieve boom gebruik maakt van services die al in de oorsponkelijke compositie voorkwamen (aangezien in principe dezelfde berekening moet uitgevoerd worden). Het *hergebruik van tussentijdse resultaten (8)* zal een serieuze snelheidswinst (zie Sectie 7.2.4 met bespreking van de metingen) opleveren die de vertraging van de *recovery* procedure min of meer kan opvangen.



Figuur 6.7: Architectuur in detail met belangrijke onderdelen

Hoofdstuk 7

Prestatie van de dynamische compositie

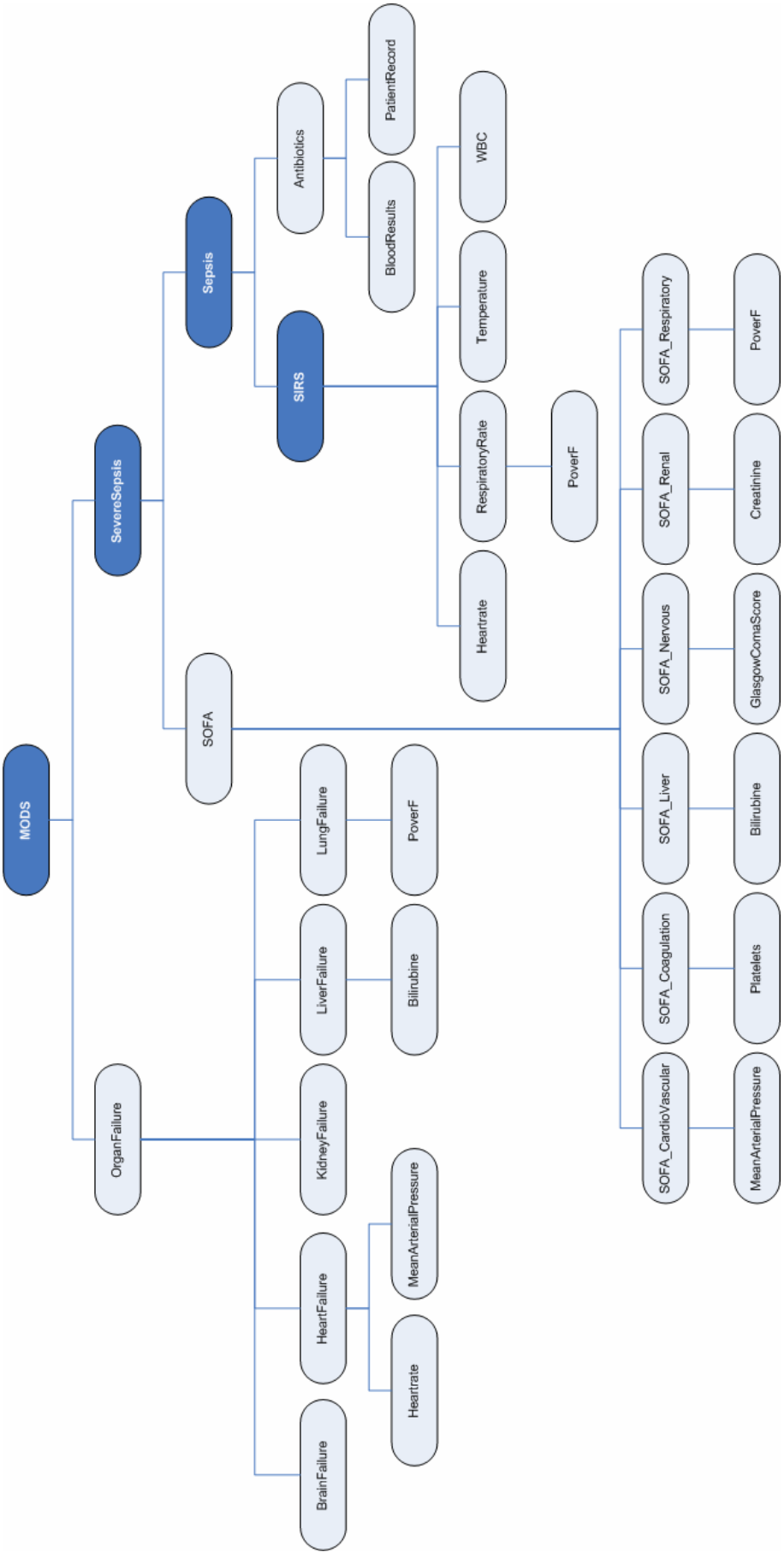
In dit hoofdstuk bespreken we de prestatietesten die we hebben uitgevoerd op de applicatie. We onderscheiden vier soorten testen: een vergelijking van de verschillende *treebuilding* algoritmes (hoe snel vinden we een compositie), de schaalbaarheid indien het aantal OR-services toeneemt, runtime testen (serieel versus parallel) en testen in verband met de recovery bij falende services.

We beginnen dit deel met een beschrijving van de testopstelling, vervolgens kijken we naar de bekomen resultaten en gaan we na of deze stroken met de verwachtingen (zoniet proberen we ook te verklaren waarom ze afwijken).

7.1 Testopstelling

7.1.1 Boom en services

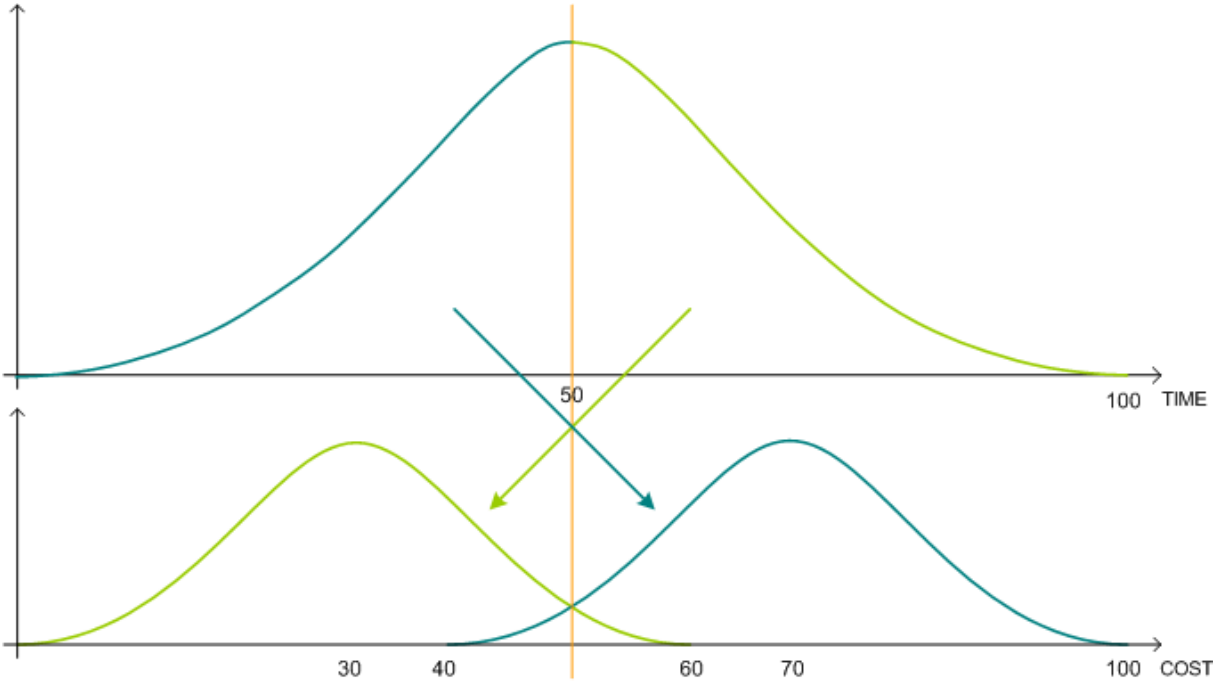
Om onze testen uit te voeren hebben we een boom nodig die voldoende groot (i.e. breed en diep) is. We hebben daarom gekozen voor de MODS (Multiple Organ Dysfunction Syndrome) case (zie Figuur 7.1 en Appendix C). Deze steunt op services die we reeds beschikbaar hadden en vormt dus een logische uitbreiding. De MODS boom is maximaal 6 niveaus diep, heeft een breedte tot 18 services en telt in totaal 35 nodes. Er zijn 30 verschillende services beschikbaar en bij elke service zijn er 5 of 10 mogelijke keuzes, wat een totaal van 150 of 300 services geeft.



Figuur 7.1: Volledige MODS boom

7.1.2 Toekennen van de gewichten

Aan elke service hebben we twee gewichten toegekend: enerzijds uitvoeringstijd, anderzijds kost. Beide gewichten kennen we per batch van 10 gelijkaardige services een Gaussiaanse distributie toe. Als we beginnen met de uitvoeringstijd, betekent dit dat de meeste services een gemiddelde uitvoeringstijd zullen hebben, met enkele uitzonderingen die heel snel of heel traag zullen zijn. Aangezien te verwachten is dat kostprijs en uitvoeringstijd aan elkaar gekoppeld zullen zijn, hebben we dit ook zo gemodelleerd. Voor de kostprijs werken we daarom met twee overlappende Gaussianen. Wanneer de uitvoeringstijd hoger dan gemiddeld is, kiezen we de 'goedkope' distributie voor de kost; bij een snellere uitvoeringstijd dan gemiddeld nemen we de 'duurdere' distributie. Op deze manier zorgen we ervoor dat globaal gezien de snellere services duurder zullen zijn dan de tragere. Door de overlappende kostdistributies sluiten we echter niet uit dat een snellere service toevallig eens goedkoper kan zijn dan een trager equivalent (zie Figuur 7.2).



Figuur 7.2: Gebruikte distributies voor het verdelen van de gewichten (tijd en kost)

7.1.3 Metingen

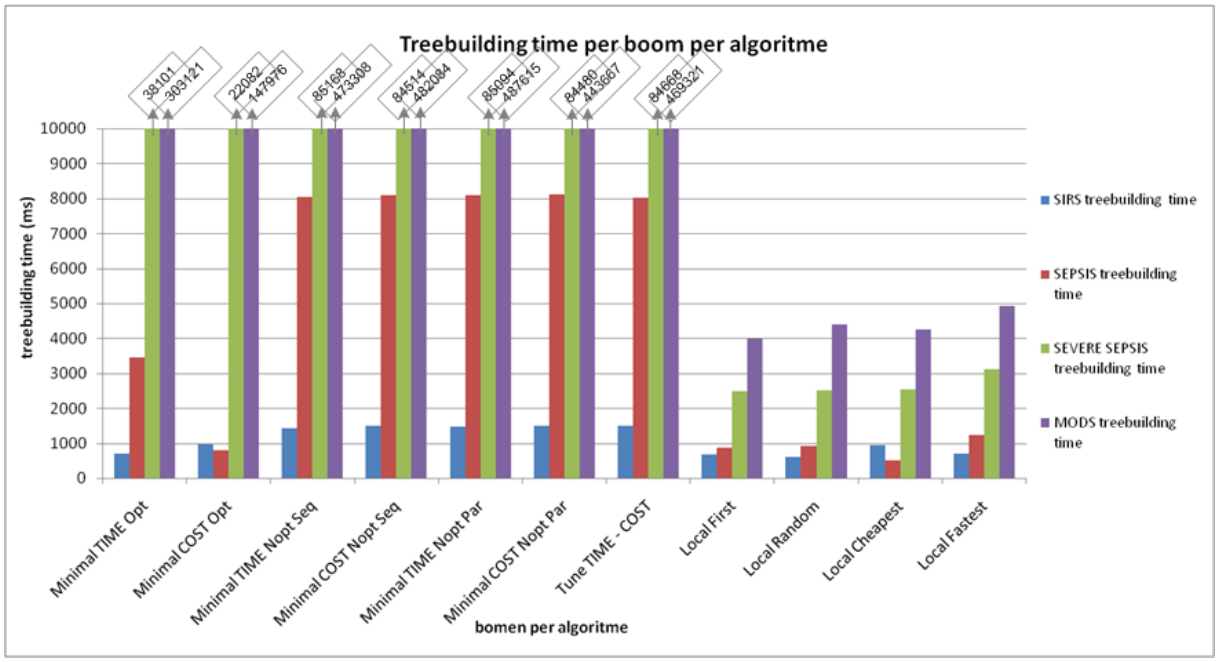
Om betrouwbare metingen te bekomen hebben we elke meting vijf of tien keer uitgevoerd. Dit laat ons toe om naast een gemiddelde waarde ook de standaarddeviatie te berekenen. Sommige metingen hebben we met 10 mogelijkheden per service gedaan; voor bepaalde bomen bleek echter dat dit extreem lang duurde en we zijn dan overgeschakeld op 5 keuzes per service. Dit wordt later meer in detail besproken.

7.2 Resultaten

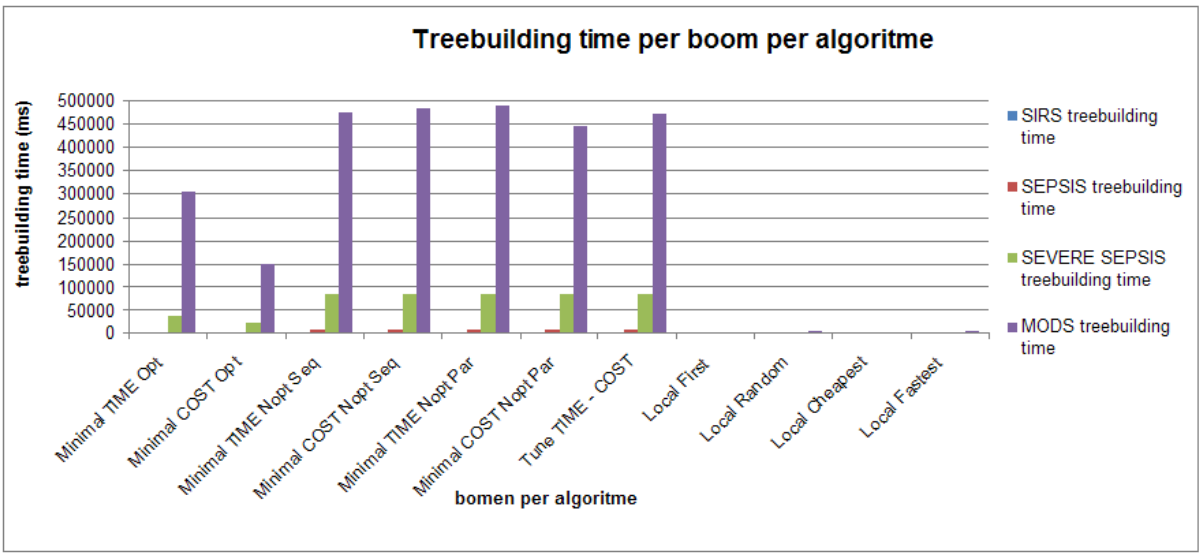
7.2.1 Treebuilding algoritmes

We bekijken hier hoe lang het duurt om een compositie met de verschillende algoritmes uit Sectie 6.5 op te bouwen. Daarnaast vergelijken we ook de gewichten (tijd en kost) van de bekomen oplossingen. Per algoritme hebben we metingen gedaan voor vier bomen (SIRS, Sepsis, Severe Sepsis en MODS) met stijgende boomgrootte waarbij men de keuze heeft tussen 5 services per node. In Figuur 7.1 kan men zien hoe deze eruitzien. Men moet daarbij wel rekening houden met het feit dat de MODS-boom de maximale vertakking toont terwijl dit niet noodzakelijk de enige juiste oplossing is. Bomen zoals SIRS en Antibiotics kunnen ook zonder inputs voorkomen.

In Figuur 7.3 kan men zien dat voor de SIRS-boom de globale algoritmes bijna even snel zijn als de lokale. Voor Sepsis is er al een verschil van meer dan 2s voor de optimale algoritmes en 7s voor de niet-optimale. De reden hiervoor is dat de lokale algoritmes onmiddellijk een keuze maken voor een bepaalde node, terwijl de globale de volledige boomstructuur beschouwen. Uitzondering is de Sepsis Minimal COST boom, maar de reden hiervoor is dat deze de SIRS en de Antibiotics bomen kiest zonder inputs. Voor kleine bomen slagen we er dus nog in om in redelijke tijd de boom op te bouwen, zelfs met een niet-optimaal globaal algoritme. Maar eens we overstappen naar wat grotere bomen (en in ons geval redelijk brede bomen) zijn de globale algoritmes niet meer aan te raden, aangezien er meer mogelijkheden te overlopen zijn door de grotere boomstructuur. Positief daarbij is het feit dat zelfs bij bomen als MODS, zie Figuur 7.4, het optimale backtracking algoritme minstens dubbel zo snel is dan de niet-optimale variant.

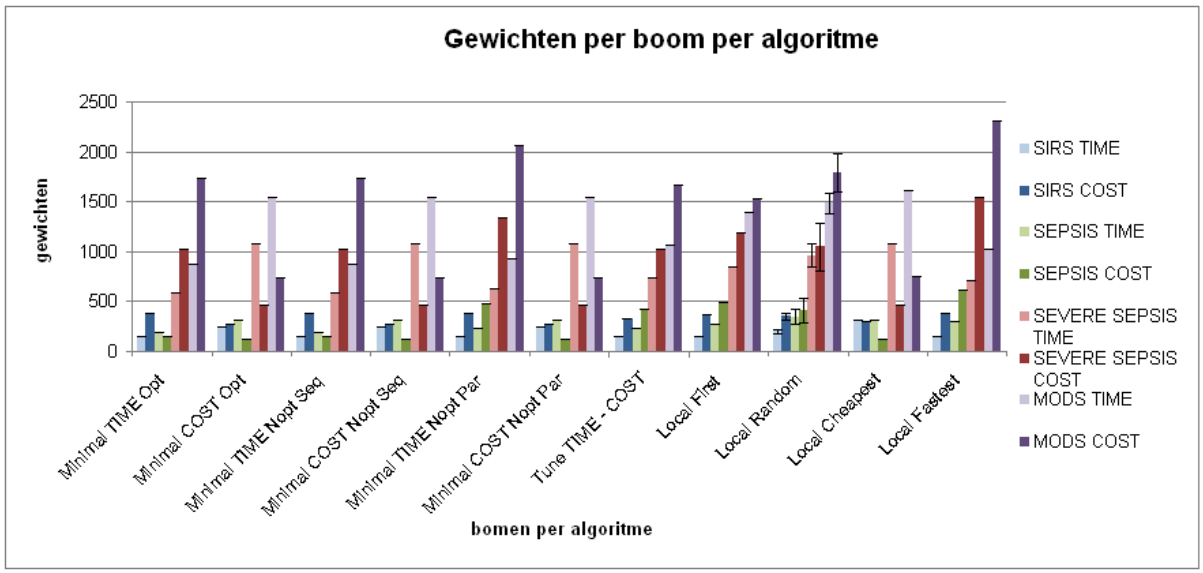


Figuur 7.3: Vergelijking uitvoeringstijden van de verschillende algoritmes



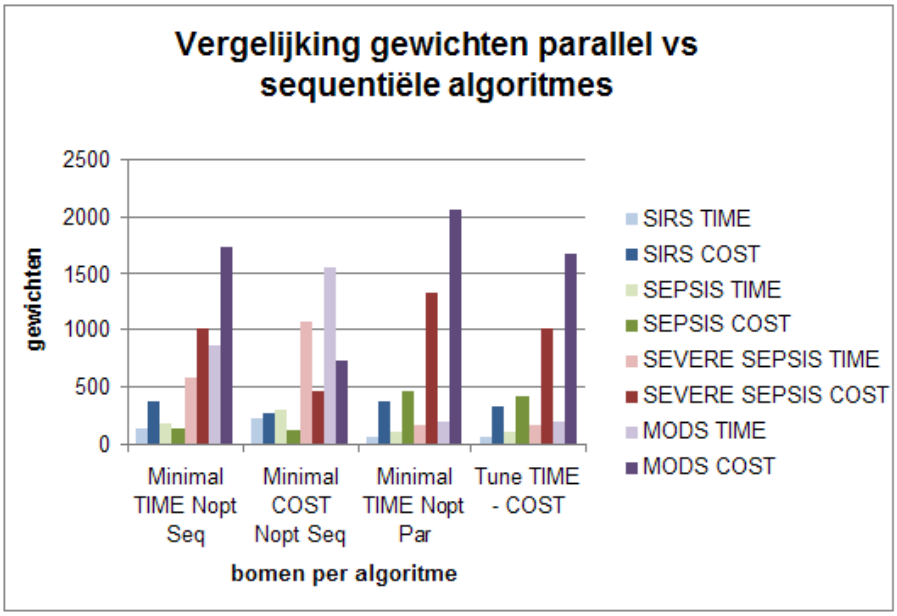
Figuur 7.4: Vergelijking uitvoeringstijden van de verschillende algoritmes, volledige schaal

In Figuur 7.5 kan men de bekomen gewichten van de bomen zien. Belangrijk hier is dat de globale optimale en niet-optimale seq algoritmes logischerwijs dezelfde boom zullen opbouwen die een optimale/minimale oplossing garandeert. In Figuur 7.6 is er ook een betere vergelijking tussen de niet-optimale seriële en parallelle oplossingen van de boom. Hierbij is belangrijk om op te merken dat het 'time' gewicht van de parallelle oplossing dat van het langste/traagste



Figuur 7.5: Gewichten van de bekomen boomstructuren met de verschillende algoritmes

pad is en niet van de volledige boomstructuur. Het parallelle algoritme is gemaakt voor de parallelle uitvoering van de boom (zie Sectie 7.2.3). Aangezien dit enkel rekening houdt met het traagste pad, kan het af en toe gebeuren dat de kost van de totale boom veel groter is dan deze van de seriële algoritme. Hier zal het Tune time-cost algoritme zijn nut bewijzen. We kunnen op de figuur onmiddellijk zien dat dit het langste pad niet zal vertragen, terwijl het ook nog in staat is om de kost naar beneden te halen via goedkopere alternatieven.



Figuur 7.6: Vergelijking niet-optimale algoritmes parallel vs. sequence en Tune time-cost

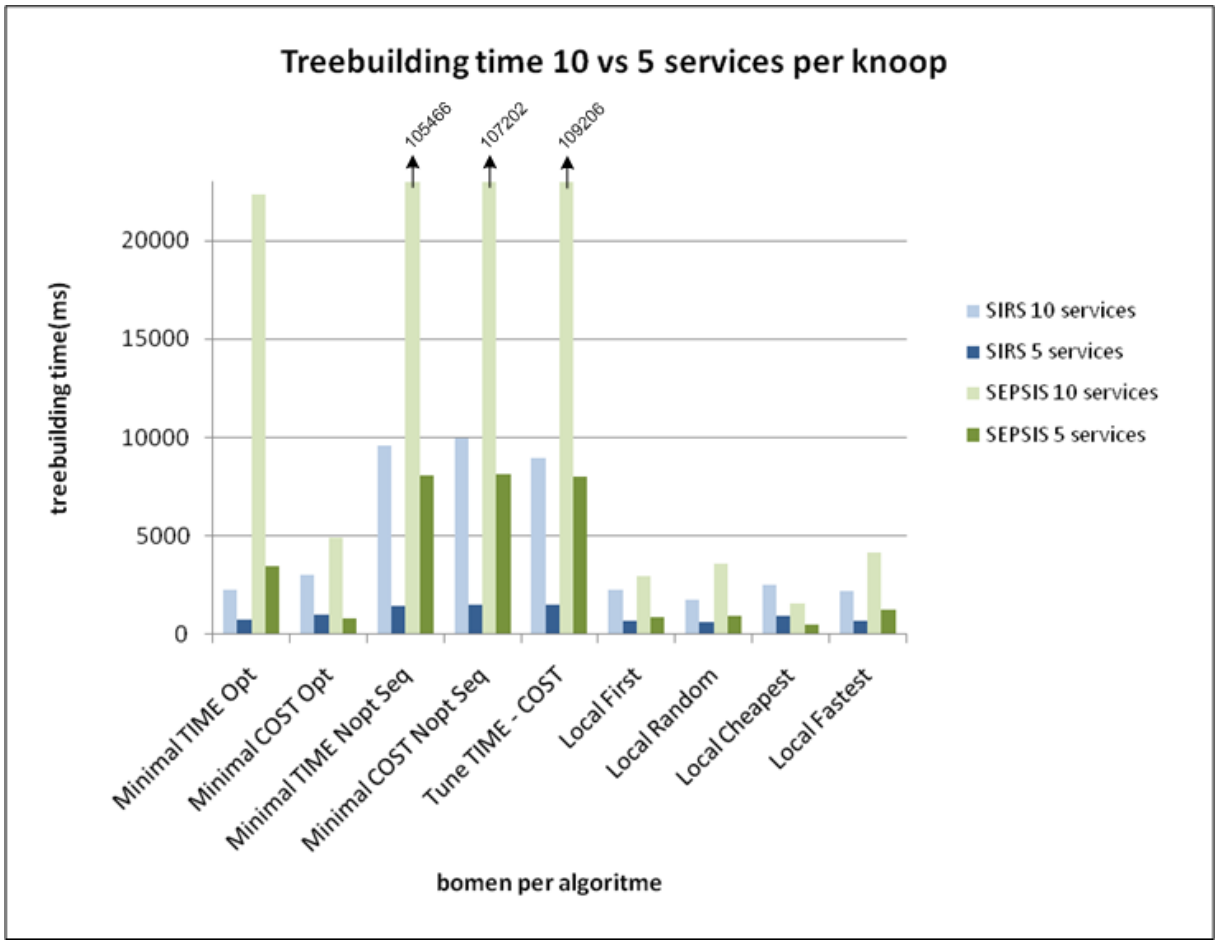
Bij de lokale algoritmes moeten we even opmerken dat de First en Random algoritmes af en toe bomen met user input genereren, wat meestal niet gewenst is bij een medische applicatie.

Besluit:

- Men gebruikt best lokale algoritmes indien men over teveel services per node beschikt en/of indien er kans is op een te grote boomstructuur.
- Men moet een afweging kunnen maken tussen een optimale boomstructuur en de uitvoeringstijd van het algoritme.

7.2.2 Schaalbaarheid

Zoals hierboven reeds vermeld, kan de uitvoeringstijd sterk oplopen bij de globale treebuilding algoritmes wanneer we het aantal keuzes per services laten toenemen (zie Figuur 7.7 met 5 of



Figuur 7.7: Vergelijking uitvoeringstijden van de verschillende algoritmes bij 5 en 10 OR services

10 services per node voor SIRS en Sepsis). We wilden dit graag proberen quantificeren, meer bepaald gingen we op zoek naar een metriek waarmee we de uitvoeringstijd kunnen voorspellen. In eerste instantie dachten we aan het aantal mogelijke oplossingen. Als we dit voor een

Service	SIRS	Sepsis	Severe Sepsis	MODS
#nodes	6	10	24	35
#levels	3	4	5	6

Tabel 7.1: Karakteristieken van de composities gebruikt voor de tests

aantal bomen berekenen (zie Tabel 7.2), merken we dat het aantal oplossingen inderdaad exponentieel stijgt met het aantal services per node (zoals we ook verwacht hadden bij een boom). Helaas is dit echter geen goede metriek om de uitvoeringstijd te voorspellen, aangezien de algoritmes niet één per één alle oplossingen overlopen. Een kleine test met enkele configuraties met vergelijkbaar aantal oplossingen bevestigde dit vermoeden (zie Tabel 7.3, de nummering van de services verwijst naar de superscripts in Tabel 7.2).

Services/node	SIRS	Sepsis	Severe Sepsis	MODS
1	1	1	1	1
2	64	1024	$\mathbf{16777216}^{4}$	3,43597E+10
3	729	59049	2,8243E+11	$\mathbf{5,00315E+16}^{6}$
4	4096	$\mathbf{1048576}^{2}$	2,81475E+14	1,18059E+21
5	15625	$\mathbf{9765625}^{3}$	$\mathbf{5,96046E+16}^{5}$	2,91038E+24
6	46656	60466176	4,73838E+18	1,71907E+27
7	117649	282475249	1,91581E+20	3,78819E+29
8	262144	1073741824	4,72237E+21	4,05648E+31
9	531441	3486784401	7,97664E+22	2,50316E+33
10	$\mathbf{1000000}^{1}$	1,00E+10	1E+24	1E+35

Tabel 7.2: Aantal oplossingen voor de compositie

Service	1	2	3	4	5	6
Tijd	5700	3950	8200	2650	75000	44000

Tabel 7.3: Treebuilding time in ms (met minimal runtime non-opt)

Een tweede metriek is het aantal keer dat de binnenste lus van het algoritme doorlopen wordt.

Het is namelijk in deze lus dat alle services recursief overlopen worden en uiteindelijk de beste geselecteerd wordt. De formule voor deze metriek wordt gegeven door

$$\sum_{i=1}^{\#inputs} \#runs_{input_i} \tag{7.1}$$

waarbij we $\#runs_{input_i}$ recursief berekenen via

$$\#runs_{input_i} = \#services\_voor\_i \cdot \sum_{j=1}^{\#inputs_i} \#runs_{input_j} + \#services\_voor\_i \tag{7.2}$$

Inderdaad, voor een bepaalde node zullen we per mogelijke service alle mogelijke inputs overlopen (de eerste term) en uiteraard zullen we voor de mogelijke services zelf ook in de lus komen (tweede term). Nemen we Antibiotics als voorbeeld (met 10 services per node), dan zullen we voor Antibiotics alle 10 services overlopen bij BloodResults en PatientRecord (totaal 20 services). Dit doen we ook bij Antibiotics2 tot en met Antibiotics10, 10 keer 20 runs maakt al 200. Daarbij tellen we de 10 keer dat we de lus uitvoeren voor de Antibiotics en we komen in totaal op 210.
Op basis van deze formule hebben we een gelijkaardige tabel geconstrueerd als hierboven (Tabel 7.4). Wanneer we nu metingen uitvoeren en op basis hiervan de tijd per service in de lus berekenen, verkrijgen we al meer voorspelbare waarden. We bekomen nog steeds geen constante waarde, maar dit is vrij logisch. Een gelijk aantal services overlopen zal langer duren bij een meer ingewikkelde boom dan bij één knoop met al deze services als kind.

7.2.3 Uitvoering van de samengestelde service

Voor de uitvoering van de compositie hebben we zoals gezegd twee mogelijke keuzes: serieel of parallel. Bij seriële uitvoering worden alle services één na één uitgevoerd. De verwachting is dan ook dat de uitvoeringstijd van de compositie evenredig is met de som van de uitvoeringstijden van de afzonderlijke services. Bij parallelle uitvoering worden onafhankelijke paden in de boom echter parallel uitgevoerd. Volgens de logica verwachten we hier dan ook een totale uitvoeringstijd die evenredig is met die van het kritisch (langste) pad in de boom.
Wanneer we de resultaten bekijken van onze metingen aangaande de uitvoering (zie Figuur 7.8), merken we dat deze veronderstelling ruwweg klopt. We stellen wel vast dat naarmate de boom groter wordt, de parallelle uitvoeringstijd steeds meer afwijkt van het ideale geval.

Services/node	SIRS	Sepsis	Severe Sepsis	MODS
1	5	9	23	34
2	12	36	148	352
3	21	87	483	1608
4	32	168	1160	4984
5	45	285	2335	12310
6	60	444	4188	26184
7	77	651	6923	50092
8	96	912	10768	88528
9	117	1233	15975	147114
10	140	1620	22820	232720

Tabel 7.4: Aantal oproepen van de inner-loop

Services/node	SIRS	Sepsis	Severe Sepsis	MODS
1	86	159	421	617
2	235	331	1703	4440
3	624	1723	10095	37099
4	708	3171	24275	134769

Tabel 7.5: Treebuilding time in ms (met minimal runtime non-opt)

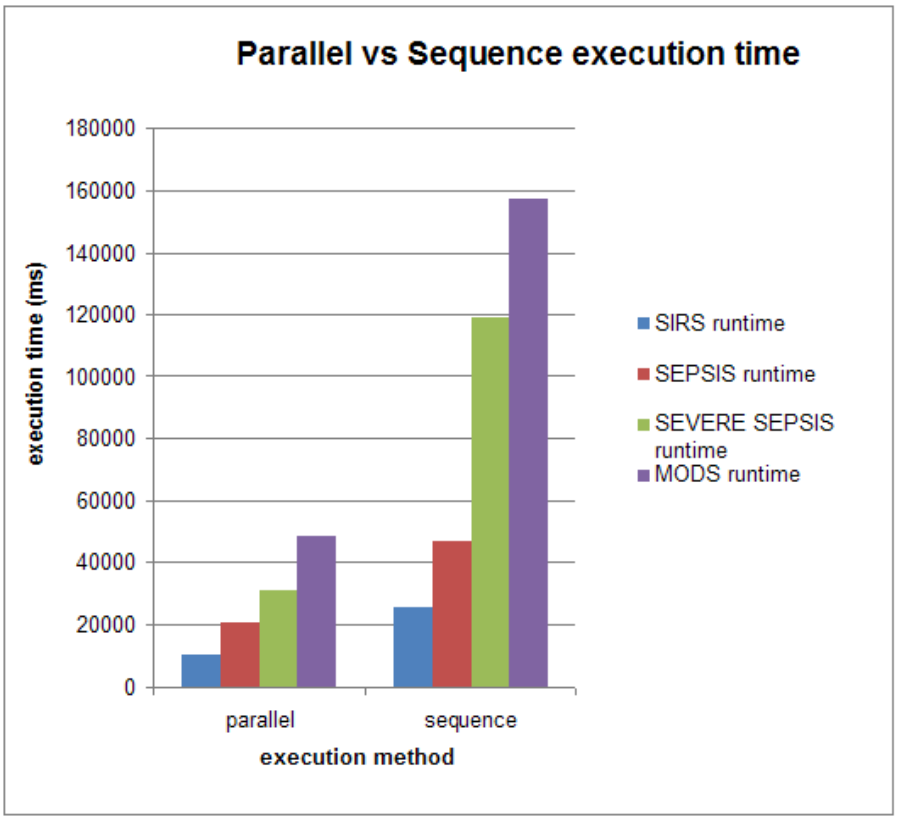
Een mogelijke verklaring hiervoor is dat al onze dummy Web services op eenzelfde machine draaien. Bij parallelle uitvoering worden heel wat threads gestart die allen tegelijkertijd de Web server aanspreken. Hierdoor kan het gebeuren dat de responstijd van een Web service wat langer is dan de nominale uitvoeringstijd.
De tijdwinst van het parallelle algoritme ten opzichte van het seriële zal het meest uitgesproken zijn bij grote en vooral bij brede bomen. Deze laatste lenen zich namelijk uitstekend tot parallellisaties.

Besluit:

- Het gebruik van parallelle uitvoering kan serieuze tijdwinsten opleveren, zeker in het geval van brede bomen.

Services/node	SIRS	Sepsis	Severe Sepsis	MODS
1	17	18	18	18
2	20	9	12	13
3	30	20	21	23
4	22	19	21	27

Tabel 7.6: Tijd in ms per oproep van de inner-loop

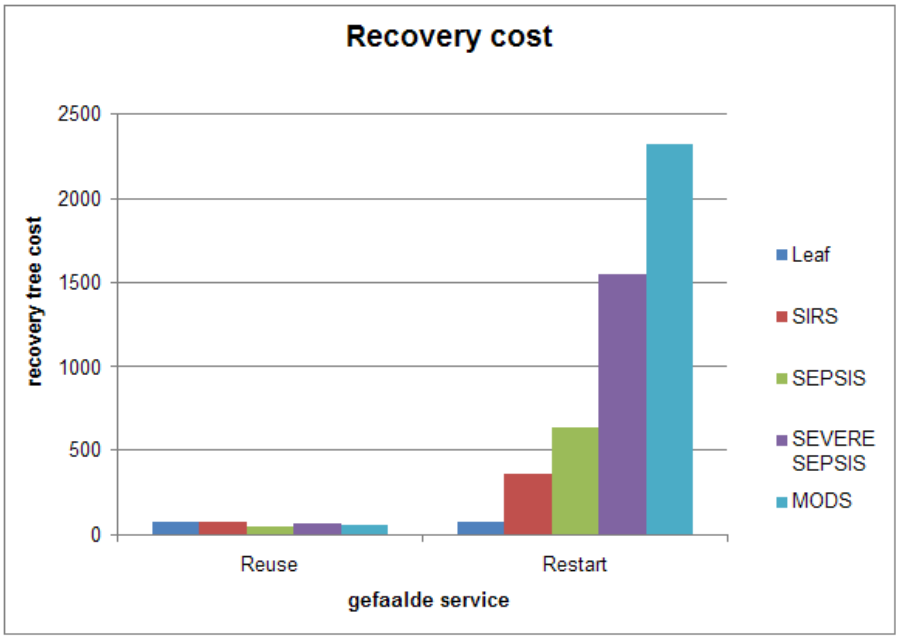


Figuur 7.8: Vergelijking uitvoeringstijden bij seriële of parallelle uitvoering

7.2.4 Recovery procedure

Een laatste test betreft de recovery procedure. Aangezien het opbouwen van de recovery boom en het uitvoeren ervan op dezelfde wijze verloopt als hierboven, heeft het weinig toegevoegde waarde die parameters opnieuw te meten. Wat wel nuttig is om te meten, is de invloed van het hergebruik van services. Wanneer services die reeds uitgevoerd zijn opduiken in de recovery boom, bestaat namelijk de mogelijkheid om de resultaten te hergebruiken in plaats van

de service opnieuw uit te voeren. Dit levert niet alleen tijdwinst op, maar kan ook een kostenbesparing betekenen in het geval er per uitvoering van de Web service moet betaald worden. Aangezien onze bomen allemaal een vrij gelijkaardige structuur hebben, kan deze methode voor deze use case een enorme winst opleveren. In het beste geval is de recovery boom gelijk aan de subboom die moet vervangen worden, op de gefaalde service na. Enkel deze laatste moet dan uitgevoerd worden, voor het overige kunnen alle resultaten hergebruikt worden.
Een minder gunstig geval is wanneer de service die de gefaalde service vervangt, totaal andere inputs heeft. In dit geval is het mogelijk dat de recovery boom allemaal nieuwe services bevat en reuse bijgevolg onmogelijk is.
Bij onze test hebben we de gefaalde service steeds dichter bij de root gesitueerd, zodat de recovery boom steeds groter werd. We merken dan ook (zie Figuur 7.9 en 7.10) dat zonder reuse zowel de kost- als tijdsparameter exponentieel stijgen, vermits elke service uit de recovery boom opnieuw moet uitgevoerd worden. In het geval van reuse merken we dat zowel kost als uitvoeringstijd ongeveer constant blijven, ondanks de toenemende grootte van de recovery boom. De verklaring hiervoor is dat om de hierboven vermelde redenen enkel de gefaalde service zal moeten vervangen worden.



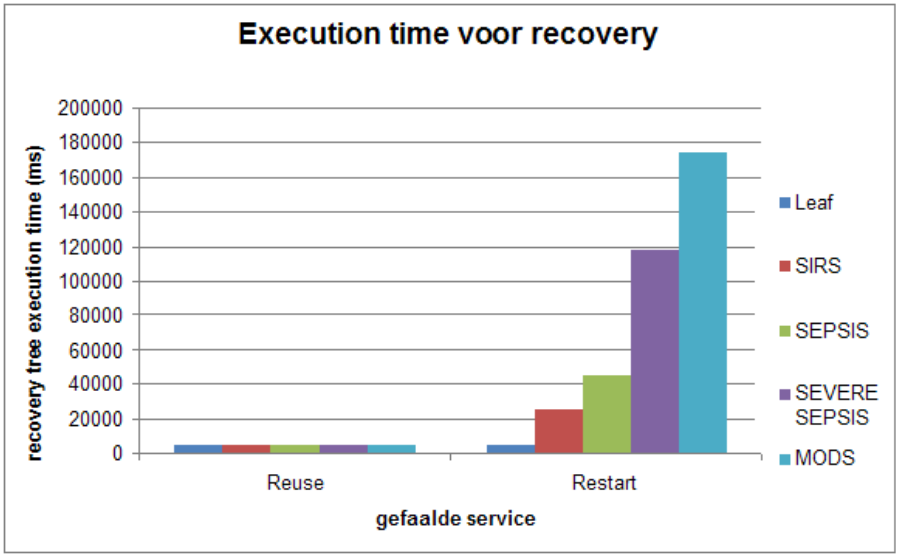
Figuur 7.9: Prijswinst bij hergebruik van services bij recovery

Besluit:

- Het hergebruik van resultaten van uitgevoerde services kan zowel kost als uitvoeringstijd

drukken.

- Dit effect is het meest geprononceerd wanneer de gefaalde boom en recovery boom homogeen zijn.



Figuur 7.10: Tijdwinst bij hergebruik van services bij recovery

Hoofdstuk 8

OWL-S vs. BPEL

Naast OWL-S bestaan er nog andere technologieën om tot compositie van Web services te komen. BPEL (Business Process Execution Language) is er één van. We plaatsen beide even naast elkaar om de gelijkenissen en verschillen duidelijk te maken.

8.1 Standaarden en toolsupport

Een eerste vaststelling is dat BPEL eerder commerciëel gericht is, terwijl OWL-S meer academisch is. Dit komt ook tot uiting in de toolsupport: voor BPEL zijn er een aantal grote spelers actief (zoals OpenESB[1], Oracle[2], IBM[3]...), terwijl het voor OWL-S meestal gaat om kleinere open source projecten. Deze projecten zijn niet allemaal even stabiel en hebben vaak geen ondersteuning.

De BPEL community is zodanig groot dat er bijna altijd oplossingen bestaan voor BPEL-gerelateerde problemen. Maar OWL en vooral OWL-S zitten nog in hun kinderschoenen. Men zal een grondig onderzoek moeten uitvoeren naar het probleem en het volledige Web uitpluizen om iets te vinden dat eventueel de oplossing zou kunnen vormen.

[1] https://open-esb.dev.java.net/

[2] http://www.oracle.com/technology/products/ias/bpel/index.html

[3] http://www.ibm.com/software/integration/wbisf

8.2 Leercurve

Verder is het ook zo dat OWL-S iets complexer is dan BPEL. Dankzij de uitstekende tools kan je in BPEL met beperkte voorkennis beginnen 'programmeren'; het volstaat via 'drag-and-drop' de juiste componenten te selecteren en samen te gooien. Voor OWL-S moet je om te beginnen al een basiskennis hebben van ontologieën en de verschillende onderdelen van OWL-S. Als je aan compositie wil doen, komen hierbij nog de reasoners die ook vrij complex zijn. De initiële investering die men zou moeten steken in OWL-S is zeker niet verwaarloosbaar (onder meer ontologieën aanmaken, een OWL-S beschrijving voorzien voor elke Web service met eventueel pre- en postcondities, etc).

8.3 Compositie van Web services

Zoals gezegd worden zowel OWL-S als BPEL gebruikt voor compositie van Web services. Toch verschillen deze twee technologieën in aanpak. Bij BPEL modelleert de gebruiker een business proces door middel van een workflow. Hierbij kan hij verschillende Web services oproepen en bewerkingen uitvoeren op het resultaat daarvan. Het BPEL proces wordt vervolgens beschikbaar gemaakt via een WSDL beschrijving. Merk op dat deze compositie puur statisch is en enkel op syntactisch niveau. Met dit laatste bedoelen we dat er geen semantiek wordt toegekend aan de resultaten. Stel bijvoorbeeld dat een Web service de operatie 'getTemperature' aanbiedt. Dan weet de computer enkel dat hij een 'double' als resultaat krijgt, maar hij weet niet wat die double betekent. Het is de gebruiker die aan die waarde de interpretatie van temperatuur geeft. Hieruit blijkt dat dynamische compositie via BPEL zonder semantiek en zonder tussenkomst van de gebruiker nog niet mogelijk is.

OWL-S is echter een ander verhaal. Dankzij ontologieën kan er aan bestaande Web services semantiek toegevoegd worden. Deze extra info zorgt er voor dat computers zelf op zoek kunnen gaan naar de benodigde Web services en dynamisch een compositie maken met het gewenste gedrag. Dit kan onder meer gebeuren aan de hand van reasoners. Binnen OWL-S zelf is er ook ondersteuning voorzien voor (statische) compositie. Het is namelijk mogelijk om via controlestructuren atomaire processen te combineren tot samengestelde processen (zie Onderdeel 2.3.1).

8.4 Runtime recovery van een samengestelde service

Een grote voordeel van BPEL is dat deze in staat is om van een compositie van verschillende services een nieuwe BPEL service met een eigen WSDL te genereren. OWL-S daarentegen zal enkel een semantische beschrijving genereren van de compositie. Daarin wordt verwezen naar WSDL's van de afzonderlijke services. Beide samengestelde services zijn even bruikbaar, alleen verwacht OWL-S meer synchronisatie voor de verschillende services (jijzelf of de OWL-S API moet de samenstellende WSDL's oproepen), terwijl dit bij BPEL achter de schermen gebeurd. Dit voordeel kan echter een nadeel vormen voor BPEL. Aangezien men een WSDL beschrijving genereert, zorgt men ervoor dat de compositie volledig statisch is. Indien er achteraf iets misgaat (bv. een service die ondertussen uitgevallen is) kan men deze compositie niet meer aanpassen. Dit is echter geen probleem voor OWL-S. Doordat de services achter de schermen door de API één voor één opgeroepen worden, is men in staat een gefaalde service onmiddellijk te onderscheppen en met behulp van semantiek te vervangen door een andere service of compositie van services. Dit kan volledig transparant gebeuren voor de gebruiker.

8.5 Uitwerking compositie

In Tabel 8.1 hebben wij een kort overzicht gemaakt van de verschillen in uitwerking tussen de BPEL en de OWL-S thesis.

Kenmerken	OWL-S	BPEL
Semantiek	OWL-S beschrijving	Pseudo semantiek in WSDL, op basis van naamgeving
Applicatie	Web Service Composer B.4.2 OWL-S API B.2.1	OpenESB
Visualisatie	Boomstructuur in Web Service Composer	Extra pakket (jGraph), zonder structuur
Startpunt	Semi-automatische compositie	Statische compositie
Vordering compositie	Dynamische compositie	Automatische compositie
Type compositie	Achterwaarts redeneren vanuit het doel	Achterwaarts redeneren vanuit het doel met ingeven van bepaalde inputs
Samengestelde service	OWL-S beschrijving met afzonderlijke services	BPEL en WSDL beschrijving
Laden van services	Bij start van Composer	Service discovery
Reuse	Samengestelde OWL-S	Samengestelde WSDL
Compositiealgoritmes	Verschillende lokale en globale	1 lokale
Extra functionaliteit	Pre- en postcondities Filtering op service parameters	

Tabel 8.1: Vergelijking OWL-S en BPEL thesis

Hoofdstuk 9

Conclusies en verder onderzoek

Dit laatste hoofdstuk bevat de conclusies die we kunnen trekken uit ons werk en onze ervaringen. Dit leidt onvermijdelijk tot een blik in de toekomst en reflectie over welke weg het best wordt ingeslaan om dit onderzoek verder te zetten. We eindigen dan ook met een overzicht van mogelijke uitbreidingen.

9.1 Conclusies

Standaarden voor op het Web worden positief toegejuicht. Het is pas door ermee aan de slag gaan, zoals wij in deze thesis, dat je ze beter leert kennen en appreciëren. Een standaard kan er in theorie nog zo mooi uitzien, als er geen toolsupport voor is of hij zich niet leent tot praktische applicaties, is hij waardeloos.
We hebben in de loop van deze scriptie aangetoond dat OWL-S zeker niet naar de catacomben van het W3C moet verwezen worden. Deze standaard is uitermate geschikt voor de semantische beschrijving van Web services. We zijn er in geslaagd om, puur op basis van deze semantiek, algoritmes te ontwerpen die volledig autonoom een compositie kunnen maken.
Deze algoritmen bleken interessant genoeg om een zijsprong te maken. Naast louter een compositie maken, werd er ook nog rekening gehouden met QoS-vereisten. Deze gingen van eenvoudigweg één parameter optimaliseren, tot het tunen van verscheidene parameters tegelijk. Aangezien al snel bleek dat naïeve algoritmes niet opgewassen waren tegen de explosieve groei van de bomen, bekeken we meer optimale algoritmen.

9.1.1 De hybride oplossing

Wanneer we al het voorgaande in overweging nemen, kunnen we bijna niet anders dan besluiten dat een hybride oplossing van OWL-S en BPEL het beste is om compositie te realiseren [22]. Beide technologieën kunnen complementair gebruikt worden. Bij deze oplossing wordt het beste van de twee werelden gebruikt, zodat de sterktes van beide standaarden naar voor komen. Het grootste pluspunt van BPEL is dat het een veel gebruikte standaard is. Er bestaan genoeg stabiele tools voor en de support is fantastisch. OWL-S daarentegen is nog zeer nieuw. De tool support is redelijk klein en er bestaat tot nu toe geen enkele echt stabiele tool. Dat maakt het werken ermee met OWL-S zeer moeizaam aangezien er van in het begin een zeer steile leercurve is.
Aan de andere kant mogen we niet vergeten dat OWL al serieus verspreid is op het Web en een fameuze bloeiperiode kent. We geloven in de mogelijkheden van deze nieuwe technologie die in staat is om een volledige dynamische compositie uit de grond te stampen met weinig tussenkomst van de gebruiker. OWL-S belooft een prachtige standaard te worden met tal aan mogelijkheden en flexibiliteit. Statisch is het verleden, dynamisch de toekomst!

Deze hybride oplossing zou volgende vorm kunnen aannemen:

1. Compositie met behulp van de semantische beschrijving in OWL-S. Hier valt er niet echt een keuze te maken aangezien BPEL totaal geen ondersteuning biedt voor semantiek en de compositiealgoritmes over de betekenis van in- en uitvoer moeten beschikken.

2. Vervolgens kan de OWL-S compositie omgezet worden naar BPEL. Dit is al op verschillende plaatsen beschreven in de literatuur [23, 24, 25]. Een bijproduct van deze omzetting is dat BPEL ook voor de compositieservice een WSDL beschrijving genereert. Hierdoor kan ook software zonder support voor OWL-S gebruik maken van de nieuwe compositie. Dit is een extra voordeel, zeker in de overgangsfase van zuivere WSDL naar WSDL+OWL-S beschrijving van Web services.

3. De uitvoering kan door de BPEL engine gebeuren. Deze lijkt ons stabieler dan de OWL-S API en er is ook meer support voor. Daarnaast zitten een aantal grote namen achter de ontwikkeling ervan (o.a. Sun) en wordt deze dus nog steeds verder ontwikkeld en verbeterd. Dit in tegenstelling tot de OWL-S API, waarvan de ontwikkeling lijkt stilgevallen te zijn.

4. De BPEL engine tot slot zal moeten uitgebreid worden met recovery, zodat bij falen van een service een nieuwe compositie kan gemaakt worden op basis van 1.

9.2 Verder onderzoek

De mogelijkheden tot verder onderzoek worden hier kort opgesomd en staan gerangschikt al naar gelang de component waarop ze van toepassing zijn.

9.2.1 Web Service Composer

In deze sectie geven we een overzicht van mogelijke startpunten voor verdere uitbreidingen aan de Composer:

- De Composer wordt uitgebreid met een *palette* met de OWL-S controlestructuren. Dit geeft de arts meer controle over de workflow door middel van beslissings- en iteratie-mogelijkheden (bv. een 'als-dan' constructie). Daarnaast kan het programma aangevuld worden met AND/OR structuren (vóór een service wordt uitgevoerd moeten alle/sommige voorgaande services eerst uitgevoerd worden). Ook aritmetische en (un)equality operatoren kunnen ondersteund worden.(zie 6 in Onderdeel 5.2.2)

- In plaats van de compositie enkel op basis van een input-output relatie te construeren, kan ook rekening gehouden worden met pre- en postcondities.

- Er kan nagegaan worden of de GUI meer gebruiksvriendelijk en ergonomisch kan gemaakt worden ten aanzien van de artsen.

- Om na te gaan welke services als bouwblok beschikbaar zijn, willen we de applicatie koppelen aan Omar, een ebXML registry/repository. De Web services die al in Omar zitten hoeven dan enkel uitgebreid te worden met een OWL-S beschrijving. Op deze manier is het beheer van de Web services mooi gecentraliseerd en hoeven ze niet onmiddellijk ingeladen te worden in de Composer.

- Er zou eventueel een soort van parser kunnen gemaakt worden om service informatie op een gestructureerde manier uit de OWL-S beschrijving te halen en op te slaan in de databank.

- Daarnaast wordt een koppeling gemaakt met de databank met patiënt- en artsinformatie. Op deze manier wordt het programma context-afhankelijk en kunnen betere suggesties gemaakt worden voor de bouwblokken. Zo heeft bijvoorbeeld een cardioloog weinig aan een gespecialiseerde nieragent.

9.2.2 Algoritmen voor automatische compositie

- Er kunnen extra algoritmes ontwikkeld worden die meer QoS parameters optimaliseren of die andere trade-offs maken.
  - Een optimaal algoritme dat een suboptimale boom opbouwt. Dit kan een soort afweging zijn tussen de beste boom en snelheid.
  - Men kan algoritmen ontwikkelen die rekening houden met de netwerktopologie en de load op de servers waar de Web services uitgevoerd worden. Deze algoritmes zullen load balancing moeten ondersteunen.
- Het cachen van bomen en eventueel subbomen zodat bij dezelfde kenmerken/hetzelfde algoritme deze niet telkens van nul opgebouwd moeten worden. Dit kan zorgen voor een serieuze tijdswinst. Hierbij kan men gebruik maken van de bestaande mogelijkheid om composities op te slaan. Het opslaan zal achter de schermen moeten gebeuren, met alle nodige informatie omtrent de boom (gebruikte algoritme, gewicht, etc). Verder zal de mogelijkheid voorzien moeten worden om de opgeslagen compositie volledig te kunnen tonen en tunen, aangezien momenteel enkel een samengestelde statische service ingeladen kan worden.

9.2.3 Dynamische compositie

Volgende veranderingen verhogen vooral de interactiviteit met de gebruiker tijdens de recovery.

- De recovery boom tonen aan de gebruiker zodat die ook getuned kan worden. In de huidige toestand wordt immers direct het resultaat van het automatisch compositie-algoritme gebruikt, zonder dat daar manueel veranderingen kunnen doorgevoerd worden.

- Indien user input nodig is, de uitvoering pauzeren en extra informatie vragen aan de gebruiker. Momenteel wordt een recovery boom met gebruikersinvoer namelijk niet toegelaten.

9.2.4 Parallelle uitvoering

De volgende uitbreidingen zijn vooral oplossingen voor de synchronisatieproblemen besproken in Sectie 6.4.

- Beperkte parallelle uitvoering zodat minder threads gestart worden.
- Het gebruik van de libraries die niet thread safe zijn synchroniseren om problemen te vermijden.

9.3 Slot

Bovenstaande suggesties illustreren duidelijk dat het eind van deze scriptie zeker niet het eind is van het onderzoek omtrent dynamische compositie. We hopen dan ook dat het resultaat van onze thesis kan dienen als basis voor verdere ontwikkelingen.

Bijlage A

Voorbeeld OWL-S beschrijving

Hieronder wordt een voorbeeld getoond van de OWL-S beschrijving van een medische Web service. Meer bepaald gaat het hier om een service die de 'Respiratory Rate' van een patiënt controleert op basis van zijn PCO_2 over FiO_2 waarde. Aandachtspunten (ook in de code aangeduid via de nummering):

1. het importeren van de medische ontologie Medical.owl
2. de semantische beschrijving van de inputs en outputs
3. koppeling aan de Web service door het verwijzen naar de WSDL in de Grounding
4. transformatie tussen ontologie en XSD datatypes via de XSL Transformaties

Opmerkingen staan in de code vermeld als HTML comments (tussen <!−− en −−>).

```
<?xml version="1.0"?>
<!DOCTYPE uridef [
  <!ENTITY service "http://www.daml.org/services/owl-s/1.1/Service.owl#">
  <!ENTITY profile "http://www.daml.org/services/owl-s/1.1/Profile.owl#">
  <!ENTITY process "http://www.daml.org/services/owl-s/1.1/Process.owl#">
  <!ENTITY grounding "http://www.daml.org/services/owl-s/1.1/Grounding.owl#">
  <!ENTITY mind "http://www.mindswap.org/2004/owl-s/1.1/MindswapProfileHierarchy.owl">
  <!ENTITY onto "http://localhost/owl/Medical.owl#">
]>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
    xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
```

```
    xmlns:list="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
    xmlns:swrl="http://www.w3.org/2003/11/swrl#"
    xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:expression="http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns="http://localhost/owl/RespiratoryRate_PoverFService.owl"
    xmlns:mind="&mind;#"
  xml:base="http://localhost/owl/RespiratoryRate_PoverFService.owl">
  <owl:Ontology rdf:about="">
     <owl:imports rdf:resource="&service;"/>
     <owl:imports rdf:resource="&profile;"/>
     <owl:imports rdf:resource="&process;"/>
     <owl:imports rdf:resource="&grounding;"/>
<owl:imports rdf:resource="&mind;"/>
     <owl:imports rdf:resource="&onto;"/>  <!-- (1) IMPORT ONTOLOGIE -->
  </owl:Ontology>

<!-- beschrijving van de onderdelen van de OWL-S beschrijving -->
  <service:Service rdf:ID="calculateRespiratoryRate_PoverFService">
    <service:presents>
      <profile:Profile rdf:ID="calculateRespiratoryRate_PoverFProfile"/>
    </service:presents>
    <service:describedBy>
      <process:AtomicProcess rdf:ID="calculateRespiratoryRate_PoverFProcess"/>
    </service:describedBy>
    <service:supports>
      <grounding:WsdlGrounding rdf:ID="calculateRespiratoryRate_PoverFGrounding"/>
    </service:supports>
  </service:Service>

<!-- begin van het Service Profile -->
  <profile:Profile rdf:about="#calculateRespiratoryRate_PoverFProfile">
    <service:presentedBy rdf:resource="#calculateRespiratoryRate_PoverFService"/>
    <profile:serviceName>calculateRespiratoryRate_PoverF</profile:serviceName>
    <profile:textDescription>Calculates Respiratory Rate based on PoverF</profile:textDescription>
    <profile:hasInput>
     <process:Input rdf:ID="PoverF">    <!-- (2) SEMANTISCHE BESCHRIJVING INPUT -->
        <rdfs:label>PoverF</rdfs:label>
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://localhost/owl/Medical.owl#PoverF</process:parameterType>
      </process:Input>
    </profile:hasInput>
    <profile:hasOutput>    <!-- (2) SEMANTISCHE BESCHRIJVING OUTPUT -->
```

```
      <process:Output rdf:ID="return">
        <rdfs:label>RespiratoryRate</rdfs:label>
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://localhost/owl/Medical.owl#RespiratoryRate</process:parameterType>
      </process:Output>
    </profile:hasOutput>
  </profile:Profile>

<!-- begin van het Service Model -->
  <process:AtomicProcess rdf:about="#calculateRespiratoryRate_PoverFProcess">
    <rdfs:label>calculateRespiratoryRate_PoverFProcess</rdfs:label>
    <service:describes rdf:resource="#calculateRespiratoryRate_PoverFService"/>
    <process:hasInput rdf:resource="#PoverF"/>
    <process:hasOutput rdf:resource="#return"/>
  </process:AtomicProcess>

<!-- begin van de Service Grounding -->
<!-- (3) VERSCHILLENDE VERWIJZINGEN NAAR DE WSDL -->
  <grounding:WsdlGrounding rdf:about="#calculateRespiratoryRate_PoverFGrounding">
    <service:supportedBy rdf:resource="#calculateRespiratoryRate_PoverFService"/>
    <grounding:hasAtomicProcessGrounding>
      <grounding:WsdlAtomicProcessGrounding
       rdf:ID="calculateRespiratoryRate_PoverFAtomicProcessGrounding"/>
    </grounding:hasAtomicProcessGrounding>
  </grounding:WsdlGrounding>
  <grounding:WsdlAtomicProcessGrounding
   rdf:about="#calculateRespiratoryRate_PoverFAtomicProcessGrounding">
    <grounding:owlsProcess rdf:resource="#calculateRespiratoryRate_PoverFProcess"/>
    <grounding:wsdlOperation>
      <grounding:WsdlOperationRef>
        <grounding:operation rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://localhost:8080/RespiratoryRate_PoverF/RespiratoryRate_PoverFService?WSDL#
        calculateRespiratoryRate_PoverF</grounding:operation>
        <grounding:portType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://localhost:8080/RespiratoryRate_PoverF/RespiratoryRate_PoverFService?WSDL#
        RespiratoryRate_PoverFPort</grounding:portType>
      </grounding:WsdlOperationRef>
    </grounding:wsdlOperation>
    <grounding:wsdlDocument rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
    >http://localhost:8080/RespiratoryRate_PoverF/RespiratoryRate_PoverFService?
    WSDL</grounding:wsdlDocument>
    <grounding:wsdlInputMessage rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
    >http://services.respiration/#calculateRespiratoryRate_PoverF</grounding:wsdlInputMessage>
    <grounding:wsdlOutputMessage rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
    >http://services.respiration/#calculateRespiratoryRate_PoverFResponse
    </grounding:wsdlOutputMessage>
```

```
    <grounding:wsdlInput>
      <grounding:WsdlInputMessageMap>
        <grounding:owlsParameter rdf:resource="#PoverF"/>
        <grounding:wsdlMessagePart rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://localhost:8080/RespiratoryRate_PoverF/RespiratoryRate_PoverFService?WSDL#
        PoverF</grounding:wsdlMessagePart>
        <!-- (4) XSL TRANSFORMATIE VOOR INPUT -->
        <grounding:xsltTransformationString><![CDATA[
          <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
            xmlns:medical="http://localhost/owl/Medical.owl#">
            <xsl:template match="//medical:PoverF">
              <xsl:value-of select="medical:PoverFValue"/>
            </xsl:template>
          </xsl:stylesheet>
        ]]></grounding:xsltTransformationString>
      </grounding:WsdlInputMessageMap>
    </grounding:wsdlInput>
    <grounding:wsdlOutput>
      <grounding:WsdlOutputMessageMap>
        <grounding:owlsParameter rdf:resource="#return"/>
        <grounding:wsdlMessagePart rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://localhost:8080/RespiratoryRate_PoverF/RespiratoryRate_PoverFService?WSDL#
        return</grounding:wsdlMessagePart>
        <!-- (4) XSL TRANSFORMATIE VOOR OUTPUT -->
        <grounding:xsltTransformationString><![CDATA[
          <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
            <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
            <xsl:template match="/">
              <xsl:variable name="X1" select="/"/>
                <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                  xmlns:medical="http://localhost/owl/Medical.owl#">
                    <medical:RespiratoryRate>
                      <medical:RespiratoryRateValue>
                        <xsl:value-of select="$X1"/>
                      </medical:RespiratoryRateValue>
                    </medical:RespiratoryRate>
                </rdf:RDF>
            </xsl:template>
          </xsl:stylesheet>
        ]]></grounding:xsltTransformationString>
      </grounding:WsdlOutputMessageMap>
    </grounding:wsdlOutput>
  </grounding:WsdlAtomicProcessGrounding>
</rdf:RDF>
```

Bijlage B

Overzicht OWL-S Tools

B.1 Editors

B.1.1 Protégé[26]

Stanford University

Protégé is een ontology editor met onder meer support voor OWL, RFD en XML schema. De editor laat toe om op grafische wijze ontologieën aan te maken. Zo kan je vlot Classes definiëren, Properties vastleggen en Classes instantiëren. Deze tool is zeer gebruiksvriendelijk en stabiel; hij telt dan ook een groot aantal gebruikers. Daarnaast bestaat er ook een plug-in die speciaal gericht is op het aanmaken van OWL-S Service Descriptions (zie volgende sectie).

B.1.2 Protégé OWL-S editor plug-in[27]

SRI International
Grit Denker

De OWL-S editor plugin[1] maakt van Protégé een zeer krachtige tool voor het werken met OWL-S web services. Protégé op zich is een ontology editor met onder meer support voor OWL, RFD en XML schema. De plugin voegt hier ondersteuning voor OWL-S aan toe. Hij laat toe

[1] Tutorial: http://owlseditor.semwebcentral.org/documents/tutorial.pdf

om via een GUI op intuïtieve wijze de verschillende onderdelen van een OWL-S web service (Service, Profile, Process en Grounding) te definiëren. Om samengestelde processen te maken is er een handige visual editor voorzien. Via drag and drop kan men zo controlestructuren en atomaire processen samenvoegen tot een nieuw proces. Naast de control flow is het ook mogelijk om daar de data flow vast te leggen. Een andere feature is dat men op basis van een WSDL beschrijving van een web service een skelet kan laten genereren voor de OWL-S beschrijving. Als laatste vermelden we nog de mogelijkheid om een service uit te voeren. Dit kan momenteel enkel voor atomaire processen waarvan de grounding gekoppeld is aan een bestaande web service. In de toekomst wil men ook de uitvoering van samengestelde processen ondersteunen.

Tijdens het werken met de plugin kwamen wel enkele minpuntjes boven water. Zo werkt de huidige versie van de plugin blijkbaar niet met de laatste versie van Protégé, maar enkel met versie 3.2.1 (en de laatste release van de plugin dateert van september 2006). Verder is het mappen van OWL-S parameters naar WSDL messages via XSLT transformaties niet goed geïmplementeerd, wat tot fouten leidt in het OWL-bestand. Dit zorgt er ook voor dat het uitvoeren niet meer lukt. Manueel de fouten corrigeren in het bestand biedt ook geen soelaas aangezien de plugin deze terug overschrijft met de foute versie. Dit zorgt dus voor serieuze beperkingen wat het uitvoeren van OWL-S processen betreft.

B.1.3 OWL-S Editor[28]

University of Malta
James Scicluna, Charlie Abela

Dit programma[2] laat toe om een OWL-S Description aan te maken, te bewerken en te laten valideren. De focus ligt vooral op het automatisch genereren van een OWL-S Description op basis van een WSDL beschrijving van een web service. Hiervoor is een wizard voorzien die de gebruiker stap voor stap begeleidt. In deze wizard zit ook een tooltje om visuele compositie mogelijk te maken van Atomic Processes; het is echter niet zo handig in gebruik als de composer van Protégé (en tevens minder uitgebreid). Het bewerken van een OWL-S Description gebeurt door rechtstreeks in het OWL bestand wijzigingen aan te brengen via de ingebouwde

[2]Tutorial: http://www.semantech.org/owlsEdit/OWL-S_Editor_User_Manual.pdf

editor met syntax highlighting. Er zijn dus geen voorzieningen om de informatie grafisch voor te stellen en te koppelen (zoals dat bij Protégé wel het geval is). Er is nog heel wat future work, zo zijn er bij de visual composer bijvoorbeeld nog maar een paar constructs geïmplementeerd. Op de website is echter geen version history terug te vinden, evenmin als een datum wanneer de laatste aanpassing gebeurd is. Het is dus moeilijk vast te stellen of dit project nog actief is.

B.1.4 CODE[29]

CMU
Naveen Srinivasan, Katia Sycara

CODE (CMU's OWL-S Development Environment)[3] is een grafische plugin voor Eclipse die in staat is om OWL-S files aan te maken of automatisch te genereren van WSDL descripties. De grafische omgeving biedt minder mogelijkheden dan de Protégé OWL-S editor (zie Sectie B.1.2) en het gebruik ervan is minder intuïtief. Daarentegen werkt de code generatie zeer proper. Deze genereert volledig correct de vier nodige bestanden vanuit WSDL. Het probleem daarbij is echter dat de Service niet kan omgaan met de koppeling met zijn eigen ServiceModel.

Wij achten deze technologie zeer beloftevol indien die verder uitgewerkt en uitgebreid zal worden.

B.1.5 Semantic Web Author[30]

H. Peter Alesso

De Semantic Web Author biedt ondersteuning voor markup talen zoals XML, RDF en OWL. Het geheel bestaat uit een teksteditor met automatische codegeneratie, een parser voor het valideren van files en een web development omgeving. Een zeer basis programma dat wel stabieler is dan de meeste bestaande grafische editors, maar het ondersteunt OWL-S niet.

[3] Tutorial: http://www.cs.cmu.edu/~softagents/papers/srinivasan_naveen_2005_1.pdf

B.1.6 ODE SWS[31, 32, 33]

ESPERONTO
Asunción Gómez Pérez, Rafael González Cabero, Manuel Lama Penín

ODE SWS is een grafische interface geïntegreerd in WebODE[34] die in staat is web services te beschrijven op een abstract niveau door gebruik te maken van Problem Solving Methods (PMS). Dit houdt in dat men enkel een beschrijving moet geven van de web services die men nodig heeft om een bepaald probleem op te lossen. Eens deze beschrijving gemaakt is, kan deze vertaald worden naar een ontologie. Op deze manier is men in staat om beschrijvingen te hergebruiken. Deze tool is zeer nuttig indien men web services van nul wil aanmaken. Men moet daarvoor geen uitgebreide kennis hebben van de onderliggende ontologie.

B.2 API's

B.2.1 Java OWL-S API[35]

Mindswap
Evren Sirin

De OWL-S API voorziet een Java interface om programmatisch OWL-S Service Descriptions te lezen, te schrijven en uit te voeren. Vooral in de laatste mogelijkheid waren we geïnteresseerd, gezien de beperkingen bij de Protégé plugin. Er is niet alleen ondersteuning voorzien voor atomaire processen met een WSDL-grounding, maar ook samengestelde processen worden (in beperkte mate) ondersteund. Het is via deze API dat we onze HelloWorld case (zie Sectie 2.4) succesvol hebben uitgevoerd (gebruik makend van XSLT).

B.3 Matchers

B.3.1 OWL-S Matcher (OWLSM)[36, 37]

Michael C. Jaeger

Deze OWL-S Matcher gebruikt het service profile van web services om OWL-S descripties met elkaar te vergelijken. Het resultaat ervan is een geordende lijst van services met graad van overeenkomst. Op basis daarvan kan men services met elkaar combineren. De matcher is een Java tool met een op Swing gebaseerde GUI, die de gebruiker in staat stelt om een service provider en requester te selecteren en te vergelijken. De tool is zeer eenvoudig opgebouwd, maar we waren niet in staat om het resultaat ervan te zien aangezien we er niet in geslaagd zijn om een service te laden.

B.3.2 OWL-S MX Matchmaker[38]

German Research Center for Artificial Intelligence
Benedikt Fries en Matthias Klusch

De OWL-S MX Matchmaker is de meest geavanceerde matcher. Op basis van een OWL-S beschrijving geef je aan naar welk soort service je op zoek bent. Vervolgens zal de Matchmaker in zijn lijst met beschikbare services op zoek gaan naar services die overeenkomen met de gevraagde specificatie. Naast exacte matches kan hij ook bijvoorbeeld een "subsumes-relatie ontdekken, meer ingewikkelde algoritmen zijn ook mogelijk. Tot slot presenteert hij een mooi overzicht van de gevonden services, geclassificeerd op basis van nauwkeurigheid van de match. Deze matchmaker zal uitgebreid aan bod komen in Sectie 4.2 Matchers.

B.4 Composers

B.4.1 Semantic Web Service Composer[39]

IBM Research
Rama Akkiraju

De Semantic Web Service Composer[4] is erop gericht om op een automatische manier web services met elkaar te matchen en te combineren, zodat het geheel voldoet aan bepaalde requirements. Indien men geen service kan vinden die voldoet aan deze requirements, maakt de

[4]Toolkit: http://alphaworks.ibm.com/tech/ettk.

engine gebruik van AI planningsalgoritmen om een compositie te vinden van services die er wel aan voldoen. Men ondersteunt semantische markup talen zoals RDF, DAML+OIL en OWL. OWL-S wordt echter nog niet ondersteund aangezien men gekozen heeft voor WSDL-S.

B.4.2 Web Service Composer[40]

MindSwap
Evren Sirin

De Web Service Composer is gemaakt om semi-automatische compositie van web services te ondersteunen. Men beschikt over een grafische interface die een soort van boomstructuur opbouwt. Bij elke stap krijgt de gebruiker een lijst van alle mogelijke services die kunnen toegevoegd worden in de compositie. De verkregen compositie kan ook onmiddellijk uitgevoerd worden met behulp van de WSDL grounding van de services. Deze composer is zeer belangrijk voor ons aangezien hij het beste uitgangspunt is voor uitbreidingen naar dynamische compositie.

B.4.3 OWLS-Xplan[41]

German Research Center for Artificial Intelligence
Matthias Klusch

OWL-S Xplan is een tool die in staat is om volledig automatisch een compositie te maken. Hiervoor specifieer je eerst de initiële toestand (het domein) en het doel (het probleem) dat je wil bereiken, beide onder de vorm van een OWL ontologie (zie verder voor een voorbeeldje). Op basis van deze twee gegevens en een lijst met OWL-S beschrijvingen van web services, zal XPlan een pad zoeken van initiële toestand naar doel. Hij zet eerst de OWL-S beschrijvingen om naar PDDL (planning domain description language), waarna hij door middel van AI planning een compositie maakt. Deze AI planner is gebaseerd op een FastForward planner met HTN (Hierarchical Task Network) planning.
Naast volledig automatische planning is er ook de mogelijkheid om manueel te plannen. Je kan dan zelf de compositie maken, volledig of slechts een deel ervan. Tot slot biedt Xplan ook ondersteuning voor dynamisch plannen. Dit betekent dat er tijdens het plannen events kunnen

gebeuren, waardoor een herplanning nodig is. Moest er bijvoorbeeld een web service off line gaan, zal Xplan op zoek gaan naar een alternatieve compositie.

Zie ook Sectie 4.3 voor een voorbeeld en screenshot (Figuur 4.2).

B.5 Brokers

B.5.1 OWL-S Broker

CMU

Katia Sycara, Massimo Paolucci, Julien Soudry, Naveen Srinivasan

De OWL-S Broker is een Middle Agent die extra functionaliteit vertoont naast het matchen van twee services. Hij is in staat om vertalingen te doen tussen ontologieën, veilige communicatie tussen services te voorzien, ... We konden er echter nergens een implementatie van vinden om te testen, dus wij vermoeden dat het om een proof-of-concept gaat.

B.5.2 SEA Broker + SCA, SDA, SGA[42]

We, the Body and the Mind

Adetti Research Lab

Deze broker is de enige gesofisticeerde werkende Middle Agent die wij gevonden hebben. Hij is verdeeld in vier functionele bouwblokken die elk een deel van de communicatie verzorgen bij het matchen van services. De Semantic Execution Agent (SEA) is de werkende entiteit die twee services met elkaar verbindt. Deze gebruikt het Generic Context System (SGA) om extra informatie (beschikbaarheid, uitvoeringstijd, ...) te verzamelen over de beschikbare services om op intelligente wijze een keuze te kunnen maken. Daarnaast hebben we de Semantic Composition Agent(SCA) die de compositie verzorgt van meerdere services om een bepaalde taak uit te voeren. Deze wordt geholpen door de Semantic Discovery Agent(SDA), die op zoek zal gaan naar geschikte services voor de compositie.

B.6 Matchmakers

B.6.1 OWL-S Matchmaker[43]

CMU
Katia Sycara

De OWL-S Matchmaker is ontworpen om service providers en service requestors bijeen te brengen. Service providers moeten zich registreren aan de hand van een OWL-S Service Profile. Service requestors geven parameters op van de service die ze zoeken, de Matchmaker zal dan op zoek gaan naar een semantische match (waarbij er een trade off is tussen performance en nauwkeurigheid). Op het moment van schrijven gaf de server echter een HTTP 500 error.

B.7 UDDI repositories

B.7.1 OWL-S Plugin for Axis[44]

TU Berlin
Michael C. Jaeger

Deze tool is een plugin voor Apache Axis. Hij laat toe om een OWL-S Service Description beschikbaar te stellen via <serviceURL>?owls, naar analogie met WSDL files. Vanwege het beperkte nut voor ons op dit moment hebben we dit nog niet getest.

B.8 Annotators

B.8.1 ASSAM Web Service Annotator[45]

University College Dublin
Andreas Heß

The ASSAM (Automated Semantic Service Annotation with Machine learning) WSDL Annotator is een applicatie die gebruikt kan worden voor het annoteren van web services. Annotaties

kunnen daarna geëxporteerd worden in OWL-S. WSDL files kunnen geannoteerd worden met een OWL ontologie met behulp van een point-and-click-interface. Een speciale feature van ASSAM is machine learning: na de training fase kan ASSAM zelf voorstellen maken voor het annoteren van datatypes in WSDL. ASSAM is nog in ontwikkeling en zou best niet gebruikt worden voor commerciële producten.

B.9 XSLT mappers

B.9.1 DL Mapping Tool[46]

University of St. Gallen
Joachim Peer

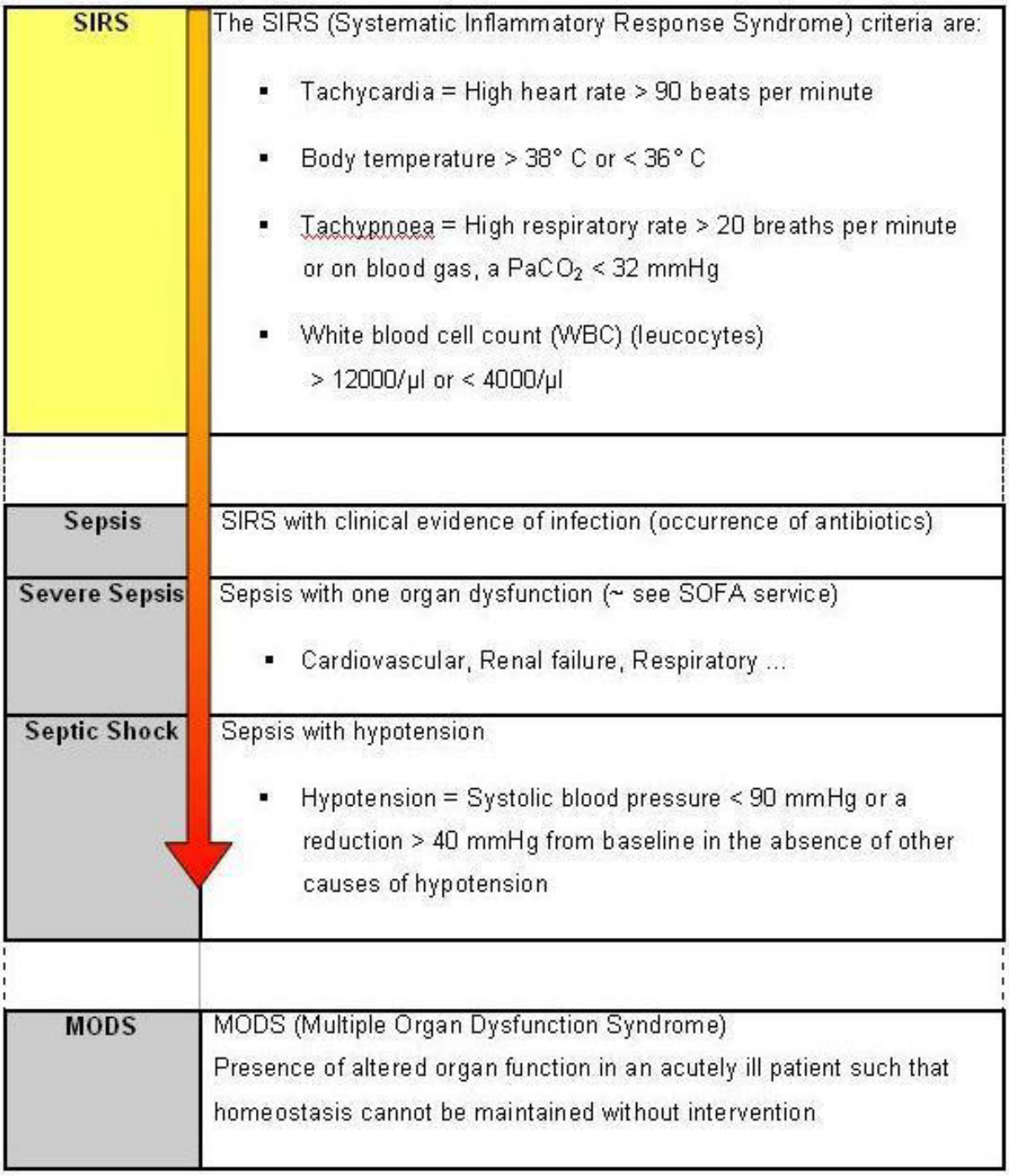
De DL Mapping Tool is specifiek geschreven voor het aanmaken van XSLT tranformaties die XML kunnen omzetten in OWL en vice versa. Deze tool zorgt voor een snelle creatie van documenten met behulp van een editor. Momenteel is er enkel ondersteuning van XSL maar andere mapping technieken kunnen ingeplugd worden in de architectuur. De tool is ook een hulpmiddel om je mappings te testen en te verifiëren en kan eventueel gebruikt worden voor het tunen van je code. Het is geschreven in Java2 en Eclipse SWT en is daardoor niet helemaal platformonafhankelijk.

Bijlage C

Verduidelijking medische use case

In onze case komen vier medische ziektebeelden [21] aan bod (Figuur C.1). Ze worden hier kort uitgelegd.

- **SIRS:** Bij patiënten op intensieve zorg komt regelmatig een veralgemeend inflammatoir syndroom of SIRS voor. Deze reactie treedt op als reactie op triggers zoals een ernstige infectie, een zware operatie, een acute ontsteking van de alvleesklier, uitgebreide brandwonden of een zwaar ongeval.
- **Sepsis:** De combinatie van SIRS met klinisch bewijs van een infectie.
- **Severe Sepsis:** Sepsis met falen van een orgaan (bv. hart, nieren,...).
- **MODS:** Als het meerdere organen betreft spreekt men van multi-orgaan falen.



Figuur C.1: Medische use case

Bijlage D

Gebruikershandleiding van de Web Service Composer

In dit hoofdstuk zal een korte gebruiksaanwijzing bij de Web Service Composer gegeven worden.

D.1 Voorbereidend werk

Vóór men met het opstarten van de Web Service Composer begint, moeten er een aantal instellingen gebeuren. In de *\resources* folder vindt men een aantal files die bij het opstarten ingelezen worden. De *services.txt* file bevat de URL's naar de OWL-S files van de in te laden services.

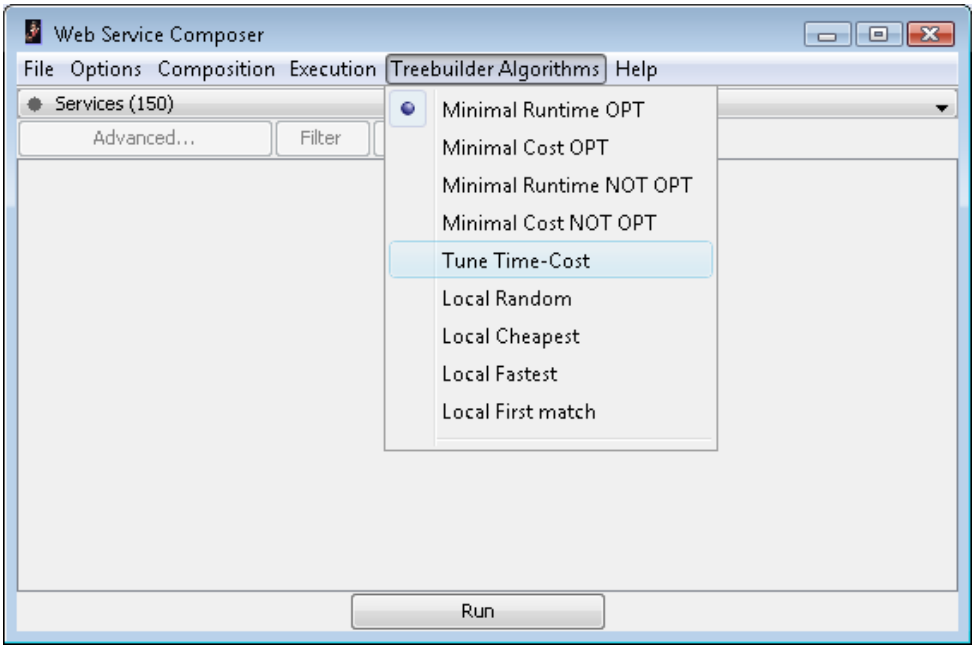
```
http://localhost/owl/RespiratoryRate_PoverF.owl
```

Daarnaast kan men eventueel in de *services_weight.txt* de gebruikte gewichten van de Web services definiëren.

```
http://localhost/owl/RespiratoryRate_PoverF.owl 53.0 9.0
```

Momenteel werkt de Composer met twee gewichten, nl. *time* en *cost*. Ze worden in deze volgorde geplaatst naast de URL van de OWL-S beschrijving.

D.2 Werken met de Web Service Composer



Figuur D.1: Keuze van het gewenste algoritme voor het maken van de compositie

Na het opstarten van de Composer zijn alle standaard instellingen samen met de nodige Web services ingeladen. De gebruiker heeft de volgende menu's ter beschikking (zie figuur D.1):

1. **File**
   - *Create OWL-S from WSDL:* automatische generatie van een OWL-S beschrijving voor een Web service met bestaande WSDL.
   - *Load OWL-S Description:* inladen van een opgeslagen OWL-S beschrijving.
   - *Save Composition:* opslaan van de OWL-S beschrijving van een samengestelde service.
   - *Exit*
2. **Options:** Hierin zitten een aantal instellingen zoals het uitschrijven van timing-informatie, het hergebruik van tussenresultaten inschakelen, de gebruikte reasoner voor de OWL-S beschrijvingen instellen, etc.
3. **Composition:** Men kan kiezen tussen een semi-automatische of een automatische compositie. Bij automatische compositie hoeft men enkel de doelservice te definiëren. De Com-

poser zal dan zelf met behulp van een bepaald algoritme (zie puntje 5) een volledige compositie samenstellen van services die nodig zijn om de gevraagde service uit te voeren. De gebruiker kan dan eventueel nog manueel aanpassingen doorvoeren en alternatieve services kiezen.

4. **Execution:** Er is hier keuze tussen het seriëel of parallel uitvoeren van een samengestelde service. Afhankelijk van de keuze zullen de algoritmes uit puntje 5 ook aangepast worden.

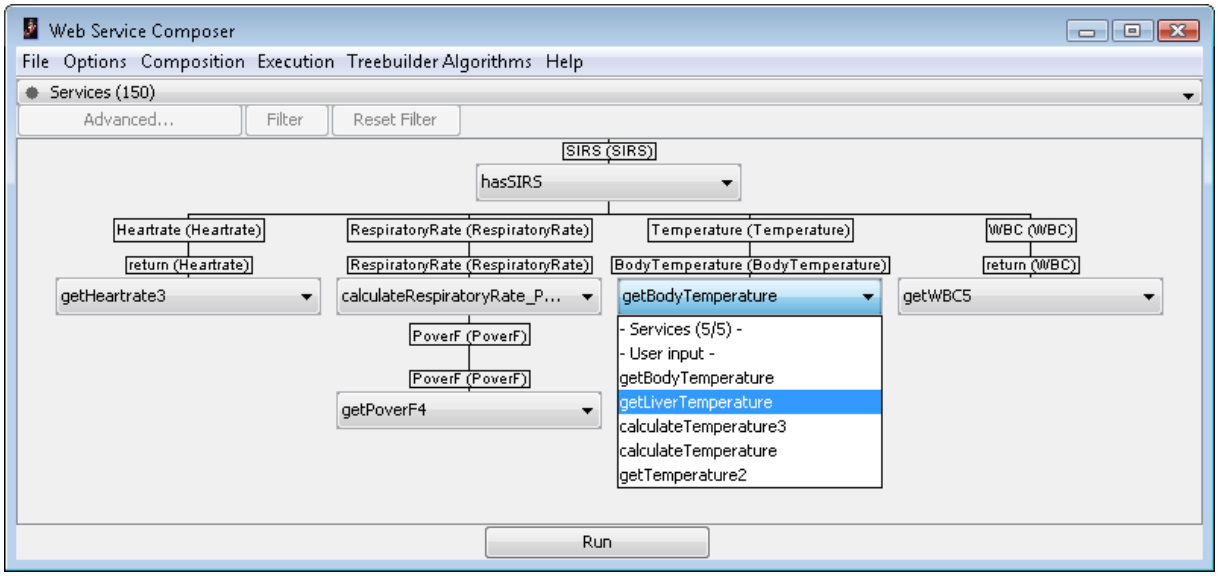
5. **Treebuilder Algorithms:** Dit menu laat de gebruiker toe zelf een algoritme te kiezen voor het opbouwen van de boomstructuur van de samengestelde service. Dit zijn de mogelijkheden tot nu toe:

   - *Minimal Runtime Opt* bepaalt een boomstructuur met minimale uitvoeringstijd mbv. backtracking.
   - *Minimal Cost Opt* bepaalt een boomstructuur met minimale kost mbv. backtracking.
   - *Minimal Runtime Not Opt* bepaalt een boomstructuur met minimale uitvoeringstijd (minimaal totaal gewicht bij sequentiële uitvoering, minimaal kritisch pad bij parallelle uitvoering) door alle mogelijkheden te overlopen.
   - *Minimal Cost Not Opt* bepaalt een boomstructuur met minimale kost door alle mogelijkheden te overlopen.
   - *Tune Time-Cost* bepaalt een boomstructuur met minimale uitvoeringstijd maar verlaagt de kost zoveel mogelijk.
   - *Local Random* bepaalt een boomstructuur door lokaal een random service te kiezen in elke node.
   - *Cheapest* bepaalt een boomstructuur door lokaal de goedkoopste service te kiezen in elke node.
   - *Local Fastest* bepaalt een boomstructuur door lokaal de snelste service te kiezen in elke node.
   - *Local First match* bepaalt een boomstructuur door lokaal de eerste service te kiezen in elke node.

6. **Help**

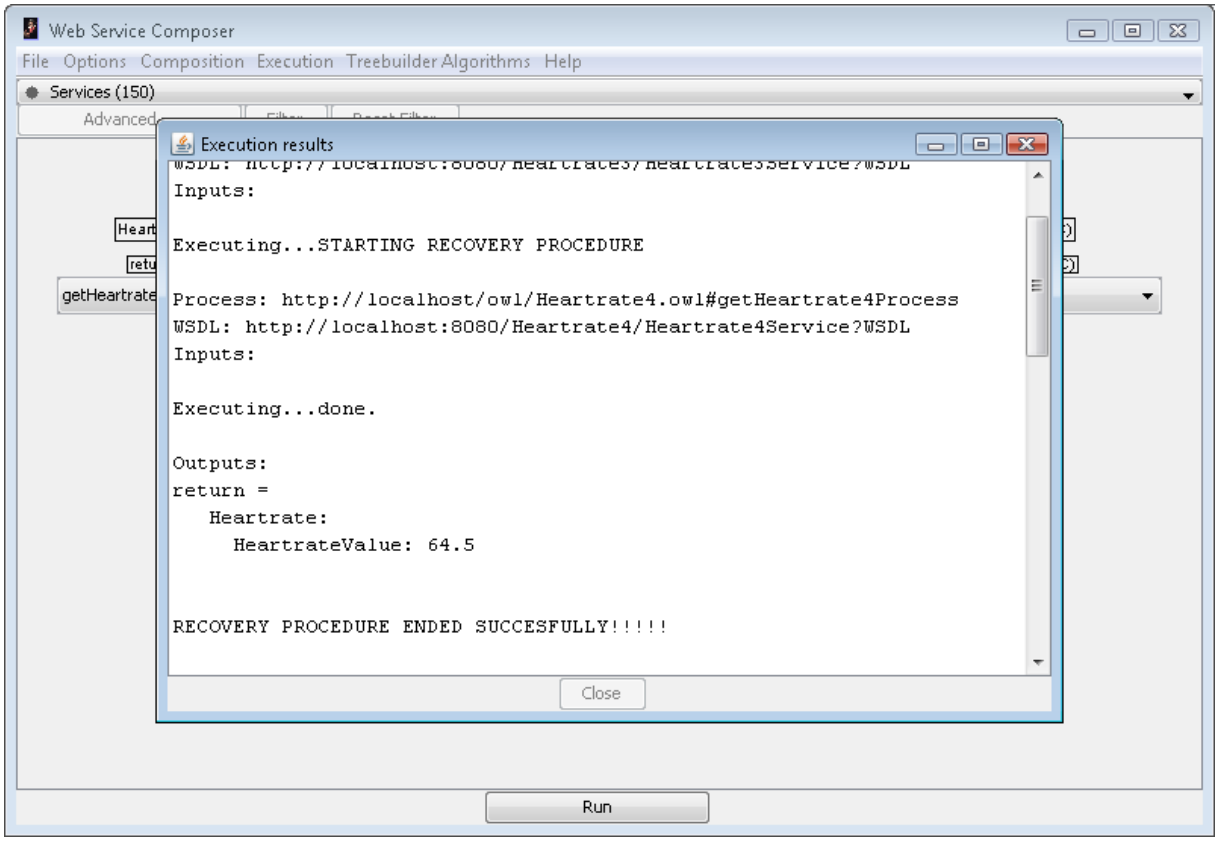
- *About*

Om een compositie aan te maken, moet de gebruiker de Web service selecteren die hij zou willen uitvoeren. De Compsoer zal dan een compositie opbouwen, eventueel met tussenkomst van de gebruiker indien de optie *Semi* aanstaat. Bij *Auto* zal vanzelfsprekend een volledig automatische boomstructuur opgebouwd worden. Men is wel nog altijd in staat om bij elke node een alternatieve service te kiezen (zie Figuur D.2). De mogelijkheden staan opgelijst in een combobox.



Figuur D.2: Het manueel aanpassen van een compositie

Daarna kan men de bekomen compositie uitvoeren. Indien er eventueel onbeschikbare services zijn, zal de Composer zelf een alternatief subboom opbouwen zodat de gebruiker niet gestoord wordt tijdens de uitvoering. Figuur D.3 toont een voorbeeld van deze recovery-procedure.



Figuur D.3: Bij een fout wordt automatisch de recovery-procedure gestart

Bijlage E

Initiatieven rond Software Agents en OWL-S

DAML Services[8]

De home pagina van het DAML project waar de OWL-S standaard ontstaan is. Een goed startpunt voor een overzicht van de standaard, voorbeeldjes en links naar allerlei tools.

Robotics Institute[47]

In dit instituut zijn er verschillende projecten (Daml-S (Semantic) Matchmaker, RETSINA Semantic Web Calendar Agent), geschreven in OWL-S, terug te vinden.

Intelligent Software Agents Lab[48]

Deze vakgroep is een onderdeel van het Robotics Institute. Ze beschikt over uitgebreid onderzoeksmateriaal en tools voor OWL-S. Het wordt aangeraden om deze site als startpunt te gebruiken indien men interesse heeft in de mogelijkheden van OWL-S.

The MINDSWAP Group[49]

Mindswap is een onderzoeksgroep voor het Semantisch Web. Hier is de source code van de meeste OWL-S tools, waaronder de OWL-S API en de Web Service Composer, te vinden.

AgentLink[50]

Dit programma rond software agents is gestart onder toezicht van de Europese Commissie.

ECS IAM[51]

Vakgroepen met uitgebreid onderzoek naar de dynamische compositie van software agents.

We, the Body, and the Mind[42]

Het CASCOM project [52] bestaande uit de SEA Broker besproken in Sectie 4.4.1.

Bibliografie

[1] OWL-S: Semantic Markup for Web Services, W3C Member Submission 22 November 2004, http://www.w3.org/Submission/OWL-S/

[2] Deborah L. McGuinness and Frank van Harmelen, Editors, OWL Web Ontology Language Overview, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/owl-features/

[3] Michael K. Smith, Chris Welty, and Deborah L. McGuinness, OWL Web Ontology Language Guide, W3C Recommendation 10 February 2004, http://www.w3.org/TR/owl-guide/

[4] Mike Dean, Guus Schreiber, Sean Bechhofer, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein, OWL Web Ontology Language Reference, W3C Recommendation 10 February 2004, http://www.w3.org/TR/owl-ref/

[5] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks, Editors, OWL Web Ontology Language Semantics and Abstract Syntax, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/owl-semantics/

[6] Jeremy J. Carroll and Jos De Roo, Editors, OWL Web Ontology Language Test Cases, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/owl-test/

[7] Jeff Heflin, Editor, OWL Web Ontology Language Use Cases and Requirements, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/webont-req/

[8] OWL-S Home Page,
http://www.daml.org/services/owl-s/

[9] OWL-S tools,
http://www.daml.org/services/owl-s/tools.html

[10] SemWebCentral, Open Source Tools for the Semantic Web,
http://projects.semwebcentral.org/

[11] Massimo Paolucci, Presentation about OWL-S Tools and Applications, The OWL-S Coalition.

[12] Matthias Klusch, Andreas Gerber, Semantic Web Service Composition Planning with OWLS-Xplan.

[13] Evren Sirin and Bijan Parsia and James Hendler, Composition-driven Filtering and Selection of Semantic Web Services.

[14] Evren Sirin, Bijan Parsia, and James Hendler, Filtering and Selecting Semantic Web Services with Interactive Composition Techniques, Published by the IEEE Computer Society IEEE INTELLIGENT SYSTEMS 1541-1672/04.

[15] Evren Sirin, James Hendler, and Bijan Parsia, Semi-automatic, Composition of Web Services using Semantic Descriptions.

[16] Massimo Paolucci, Julien Soudry, Naveen Srinivasan and Katia Sycara, A Broker for OWL-S Web services, The Robotics Institute, Carnegie Mellon University.

[17] Katia Sycara, Massimo Paolucci, Julien Soudry, and Naveen Srinivasan, Dynamic Discovery and Coordination of Agent-Based Semantic Web Services, Carnegie Mellon University, MAY - JUNE 2004 Published by the IEEE Computer Society 1089-7801/04/.

[18] António Luís Lopes, Luís Miguel Botelho, Executing Semantic Web Services with a Context-Aware Service Execution Agent, We, the Body, and the Mind Research Lab of ADETTI-ISCTE.

[19] António Lopes, Luís Botelho, SEA: a Semantic Web Services Context-aware Execution Agent, We, the Body, and the Mind Research Lab of ADETTI-ISCTE.

[20] Massimo Paolucci, Katia Sycara, Takuya Nishimura, Naveen Srinivasan, Toward a Semantic Web e-commerce, Witold Abramowicz, Gary Klein (eds.), Business Information Systems, Proceedings of BIS, 2003, Colorado Springs, USA.

[21] Per-Olof Nyström, The systemic inflammatory response syndrome: definitions and aetiology, Journal of Antimicrobial Chemotherapy (1998) 41, Suppl. A, 1-7.

[22] Paolo Traverso and Marco Pistore, Automated Composition of SemanticWeb Services into Executable Processes, Springer-Verlag Berlin Heidelberg 2004.

[23] Conversion From OWL-S To BPEL Online Demo,
http://www.laits.gmu.edu:8099/OWLS2BPEL/

[24] Muhammad Ahtisham Aslam, Sören Auer, and Jun Shen, From BPEL4WS Process Model to Full OWL-S Ontology, Demos and Posters of the 3rd European Semantic Web Conference (ESWC2006), Budva, Montenegro, 11th 14th June, 2006.

[25] Behzad Bordbar, Gareth Howells, Michael Evans, and Athanasios Staikopoulos, Model Transformation from OWL-S to BPEL Via SiTra.

[26] Protégé, Stanford University,
http://protege.stanford.edu/

[27] Grit Denker, OWL-S Protégé-based Editor, SRI International,
http://owlseditor.semwebcentral.org/

[28] James Scicluna, Charlie Abela, OWL-S Editor, University of Malta,
http://staff.um.edu.mt/cabe2/supervising/undergraduate/owlseditFYP/OwlSEdit.html

[29] Naveen Srinivasan, Katia Sycara, CMU's OWL-S Development Environment, CMU,
http://projects.semwebcentral.org/projects/owl-s-ide/

[30] H. Peter Alesso, Semantic Web Author,
http://www.semantic-search.com/downloadswa.aspx

[31] Asunción Gómez Pérez, Rafael González Cabero, Manuel Lama Penín,ODE SWS, ESPERONTO project,
http://kw.dia.fi.upm.es/odesws/

[32] Óscar Corcho, Mariano Fernández-López, Asunción Gómez-Pérez, and Manuel Lama (2003), ODE SWS: A Semantic Web Service Development Environment, VLDB-Workshop on Semantic Web and Databases, 203-216. Berlin, Germany.

[33] Asunción Gómez-Pérez, Rafael González-Cabero, and Manuel Lama (2004), A Framework for Design and Composition of Semantic Web Services, AAAI Spring Symposium on Semantic Web Services, 113-120. Stanford, USA.

[34] Raúl García Castro, WebODE, Ontology Engineering Group, http://kw.dia.fi.upm.es/wpbs/

[35] Evren Sirin, Mindswap OWL-S Java API, Mindswap, http://www.mindswap.org/2004/owl-s/api/

[36] Michael C. Jaeger, OWL-S Matcher, http://OWLsm.projects.semwebcentral.org/

[37] Michael C. Jaeger, The DAML-S Matcher from TUB, http://ivs.tu-berlin.de/Jaeger/damlsmatcher

[38] Benedikt Fries and Matthias Klusch, OWL-S MX Matchmaker, http://www-ags.dfki.uni-sb.de/ klusch/owls-mx/index.html

[39] Rama Akkiraju, Semantic Web Service Composer, IBM Research, http://alphaworks.ibm.com/tech/wssem

[40] Evren Sirin, Web Service Composer, MindSwap, http://www.mindswap.org/~evren/composer/

[41] Matthias Klusch, OWLS-Xplan, http://projects.semwebcentral.org/projects/owls-xplan/

[42] We, the Body, and the Mind, http://we-b-mind.org/

[43] Katia Sycara,DAML-S Matchmaker, CMU, http://www.cs.cmu.edu/~softagents/daml_Mmaker/daml-s_matchmaker.htm

[44] Michael C. Jaeger,OWL-S Plugin for Axis, TU Berlin, http://www.mcj.de/mirrors/owlsplugin/

[45] Andreas Heß, ASSAM Web Service Annotator, University College Dublin,
http://moguntia.ucd.ie/projects/annotator/download

[46] Joachim Peer, DL Mapping Tool, University of St. Gallen,
http://lists.w3.org/Archives/Public/www-ws/2003May/0126.html

[47] Robotics Institute,
http://www.ri.cmu.edu/

[48] Katia Sycara, Joseph Giampapa,Research Staff, Technical Staff and Students, Intelligent Software Agents Lab,
http://www.cs.cmu.edu/~softagents/

[49] The MINDSWAP Group,
http://www.mindswap.org/

[50] AgentLink,
http://www.agentlink.org/

[51] Intelligence, Agents, Multimedia,
http://www.iam.ecs.soton.ac.uk/

[52] Thorsten Möller, Heiko Schuldt, Andreas Gerber and Matthias Klusch, Next-generation applications in healthcare digital libraries using semantic service composition and coordination, HEALTH INFORMATICS J 2006; 12; 107.

Lijst van figuren

Lijst van tabellen

Lijst van Algoritmes