UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Tool support for programming human-robot interaction

Maarten Martens

Scriptie ingediend tot het behalen van de graad van master in de informatica, afstudeerrichting multimedia

PROMOTOR :

dr. Jan VAN DEN BERGH

2017
2018

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Tool support for programming human-robot interaction

Maarten Martens

Scriptie ingediend tot het behalen van de graad van master in de informatica, afstudeerrichting multimedia

PROMOTOR :

dr. Jan VAN DEN BERGH

Abstract

De vooruitgang in robotica vereist verbeterde communicatietechnieken om met robots te interageren. Multimodale interactie kan een oplossing bieden die toelaat robots op een robuuste en intuïtieve manier te besturen. Het koppelen van robots met multimodale frameworks vereist echter veel technische kennis. In multidisciplinaire vakgebieden als robotica en multimodale interactie zorgt dit ervoor dat onderzoek en prototyping van interacties met robots traag verloopt. Het voorzien van een generieke oplossing om interactie met een robot te definiëren zou ervoor zorgen dat prototyping en testing op een eenvoudige manier tot stand kan worden gebracht.

Het doel van deze thesis is het uitbreiden van een multimodaal framework zodat interactie met een robot op een eenvoudige manier kan worden toegevoegd. Hiervoor werd eerst een vergelijking gemaakt van een aantal verschillende multimodale frameworks. Deze vergelijking gaf aan dat *state-based* frameworks de meeste functionaliteit bevatten voor een aanvaardbare complexiteit. Uiteindelijk werd gekozen om Hasselt UIMS uit te breiden. Er werd tevens een vergelijking gemaakt tussen de verschillende robot software. Hierbij werd gekozen om ROS te gebruiken, aangezien deze het meest gebruikt wordt binnen het domein en tevens toelaat acties uit te voeren aan de hand van de actionlib uitbreiding.

Het resultaat van deze thesis is een plugin voor Hasselt UIMS, die gebruikers toelaat interacties met een robot te definiëren via het ROS protocol. Bovendien wordt actionlib ondersteund, waardoor acties kunnen worden toegekend aan robots. Hierdoor kan informatie over de vooruitgang worden teruggekoppeld naar Hasselt. Het configureren van deze interacties gebeurt aan de hand van een interface die speciaal ontwikkeld werd. Hierdoor is de tool eenvoudig bruikbaar door niet-technische gebruikers. De tool ondersteunt alle robots die ROS of actionlib gebruiken. De plugin is dus een stap in de goede richting om het prototypen van multimodale interactie met robots te vereenvoudigen.

Inhoudsopgave

1 Introductie

Robots worden reeds enkele decennia gebruikt om mensen te vervangen bij het uitvoeren van allerhande taken. Hun mogelijkheid om repetitieve taken met een vastgelegde nauwkeurigheid en snelheid uit te voeren is van groot belang binnen het huidige industrieële landschap. Denk aan grote fabrieken waar tientallen robots in serie worden opgesteld om een product te maken. De limiet van wat deze soort robots kunnen verwezenlijken is echter bereikt. Het uitvoeren van deze acties vereist programmatie en synchronisatie van de robot met zijn omgeving. Hierdoor blijft hun complexiteit redelijk laag. Om de grenzen van robots te verleggen, werd mens-robot interactie voorgesteld. Door mens-robot interactie te implementeren, kunnen de beste aspecten van zowel mens als robot worden genuttigd. Deze interactie zal echter op een natuurlijke, accurate en robuuste manier moeten worden gedefiniëerd.

Om dit te realiseren, kan multimodale interactie een oplossing bieden. Bij multimodale interactie worden verschillende kanalen gebruikt om informatie over te dragen. Door de informatie van de verschillende kanalen samen te interpreteren, kan een accurate weergave van de input of output worden gemaakt. Het implementeren van een multimodaal systeem is echter niet eenvoudig. Dit is mede te danken aan de complexiteit van multimodale interactie, een gebrek aan een standaard binnen het domein en de nood aan geavanceerde technologieën om informatie van kanalen te interpreteren. Er zijn reeds verschillende tools ontwikkeld die het implementeren van een multimodaal systeem proberen te vergemakkelijken.

Deze tools kunnen geclassificeerd worden als User Interface Management Systems (UIMS) [14, 18, 42, 38]. In een UIMS kunnen interacties op een hoog niveau worden gedefiniëerd, afgezonderd van de semantiek van de achterliggende applicatie. In deze thesis zal gekeken worden naar de Hasselt UIMS en hoe de koppeling met het Robot Operating System (ROS) kan worden geïmplementeerd. Het doel is het vereenvoudigen en versnellen van het prototypen en testen van multimodale interacties met robots.

2 Achtergrond

Vooraleer concrete doelen kunnen worden opgesteld voor deze thesis, is een goed beeld van de nodige aspecten vereist. In deze sectie zal een overzicht worden gegeven van de huidige staat van *Human-Computer interaction* (HCI) en de verschillende classificaties die hierin kunnen worden gemaakt. Vervolgens wordt gekeken naar multimodale interactie en wat de rol hiervan is in verband met *Human-Robot Interaction* (HRI). Daarna zullen *User Interface Management Systems* worden besproken, om ten slotte een overzicht te geven van de beschikbare robot software frameworks.

2.1 Human-Robot Interaction

Human-Robot Interaction (HRI) is een studiedomein dat zich richt op het begrijpen, ontwerpen en evalueren van robotische systemen die gebruikt worden door of met mensen [21]. Hierbij wordt de focus vooral gelegd op de interactie tussen mens en robot. Interactie tussen mens en robot vereist per definitie communicatie. Deze communicatie kan verschillende vormen aannemen, zoals praten, gebaren maken of communicatie via software. De gekozen vorm van communicatie is echter vaak gerelateerd aan de toepassing van de robot. Zo worden volgende categorieën gedefinieëerd [21]:

- **Remote Interaction** - De robot bevindt zich op een andere locatie dan de gebruiker. De Mars Rover is een robot die zowel spatiaal als temporaal gescheiden is van de aarde.
- **Proximate Interaction** - De robot bevindt zich op dezelfde locatie als de gebruiker. Voorbeelden van zulke robots zijn *service robots*.

Voor beide interacties kan een bijkomende onderverdeling worden gemaakt. Deze onderverdeling wordt gemaakt aan de hand van de toepassing van de robot.

- Mobiel - robots die mobiel moeten zijn. Voorbeelden zijn ontmijningsrobots, verkennings-drones of service robots.
- Fysieke handeling - robots die een fysieke handeling moeten uitvoeren. Voorbeelden zijn Industrieële robots of service robots.

- Sociale interacties - robots die sociale interacties moeten hebben. Voorbeelden zijn service robots. Wanneer een robot sociale interacties moet verrichten zal deze meestal lokaal worden gestuurd.

Een robot kan tot meerdere onderverdelingen behoren. Aan de hand van deze specificaties kan dan een gepaste interactiemethode worden gekozen. Zo zal voor een service robot, die behoort tot alle onderverdelingen, meestal gekozen worden voor *proximate interaction*.

Om mens-robot interactie zo vlot mogelijk te laten verlopen, is het belangrijk dat deze interactie een positief effect heeft op de bestuurbaarheid van de robot. Er is geen maatstaf die aangeeft of een interactie al dan niet positief is. Of een interactie positief is kan echter wel beschreven worden aan de hand van een aantal aspecten die rekening houden met het type robot. In de volgende secties wordt een kort overzicht gegeven van deze aspecten.

2.1.1 Autonomie

Autonomie is per definitie de mate waarin iets kan handelen zonder bestuurd te worden. Zo zijn mensen autonoom; ze bepalen zelf hun acties zonder die opgelegd te krijgen. Bij robots bestaat autonomie uit het mappen van inputs uit de omgeving naar bepaalde acties zoals bewegen, weergeven van informatie of spreken. Autonomie impliceert dus een mate van intelligentie; een robot ontvangt een aantal inputs en moet beslissen welke acties hij uitvoert als reactie op deze inputs. Er bestaan talloze formele definities van autonomie die een beschrijving geven aan de hand van kwantificeerbare waarden. Een voorbeeld van zo een waarde van autonomie die toepasbaar is op mobiele robots is *neglect tolerance*. Een hoge *neglect tolerance* geeft aan dat de robot in kwestie voor langere tijden alleen kan worden gelaten zonder menselijke interactie. Hoewel deze waarde een indicatie van autonomie geeft voor mobiele robots, is het voor sociale interacties of het weergeven van representaties echter nietszeggend.

Autonomie is enkel nuttig zolang het een positieve bijdrage levert aan de interactie tussen mens en robot. Hierdoor zal de implementatie van autonomie afhangen van de toepassing(en) van de robot in kwestie. Een sterke notie voor autonomie voor *human-centered* toepassingen is *Level of Autonomy (LOA). Level of autonomy* beschrijft in welke mate een robot zelf acties kan uitvoeren. Net zoals autonomie heeft *level of autonomy* veel definities. De meest geciteerde definitie is echter die van Tom Sheridan [39]. *Sheridan's scale* beschrijft 10 niveaus van autonomie:

1. Computer offers no assistance; human does it all.
2. Computer offers a complete set of action alternatives.
3. Computer narrows the selection down to a few choices.
4. Computer suggests a single action.
5. Computer executes that action if human approves.
6. Computer allows the human limited time to veto before automatic execution.
7. Computer executes automatically then necessarily informs the human.
8. Computer informs human after automatic execution only if human asks.
9. Computer informs human after automatic execution only if it decides to.
10. Computer decides everything and acts autonomously, ignoring the human.

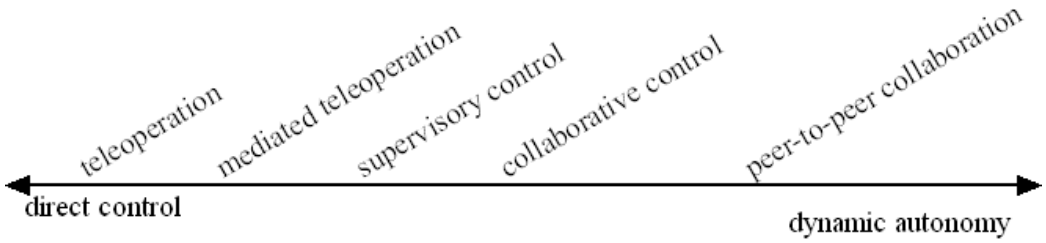
Belangrijk om op te merken is dat deze beschrijving van autonomie niet toepasbaar is op een volledig probleemdomein [25]. Het kan echter wel nuttig zijn bij deelproblemen van het domein. Zo kunnen verschillende aspecten van een robot verschillende niveaus van autonomie hebben. *Sheridan's scale* geeft een representatie van de gemiddelde autonomie van een robot over al zijn taken.

Hoewel dit interessant is om een algemeen beeld te krijgen over de autonomie van een robot, is het op vlak van mens-robot interactie niet beschrijvend genoeg. Een complementaire manier om autonomie te beschrijven is aan te geven welke soort interactie wordt gebruikt tussen mens en robot en in welke mate beide autonoom kunnen werken. De schaal weergegeven in figuur 2.1 geeft een idee van de autonomie van een robot voor bepaalde soorten interacties. Interacties waarvoor een robot weinig autonoom moet zijn bevinden zich links op de schaal. Dit zijn interacties waarbij de meeste controle bij de mens ligt. Naarmate we naar rechts opschuiven wordt de rol van de robot in de interactie groter.

Uiterst rechts op de schaal bevindt zich peer-to-peer collaboration. Bij deze soort van interactie moet een robot dynamische autonomie vertonen. Hierbij

komt kijken dat de robot sociale interacties moet ondersteunen zodat een efficiënte collaboratie mogelijk is. Om deze redenen is het mogelijk dat peer-to-peer collaboratie moeilijker te implementeren is dan volledige autonomie.

Belangrijk om op te merken is dat verschillende problemen opduiken over de schaal. Zo zal bij directe controle van een robot aandacht worden besteedt aan het ontwerpen van een interface die een zo laag mogelijke cognitieve kost heeft voor de gebruiker. Aan de andere kant van de schaal zal dan weer moeten worden onderzocht hoe robots op een natuurlijke en efficieënte manier kunnen communiceren met mensen om collaboratie toe te laten.



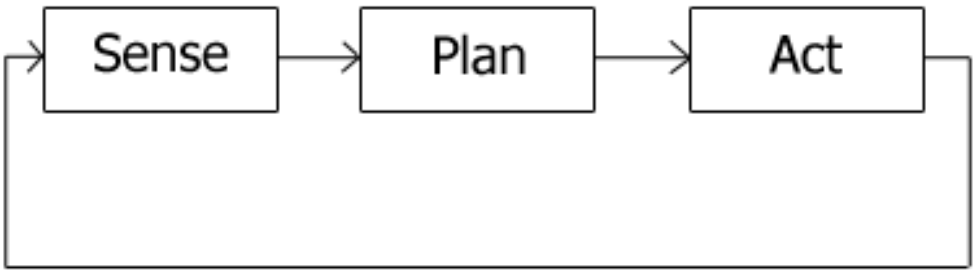
Figuur 2.1: *Levels of autonomy* [21]. De mate van autonomie die een robot moet vertonen hangt af van de gewenste interactie. Zo moet voor teleoperatie geen autonomie worden voorzien aangezien de robot rechtstreeks wordt gecontroleerd door een persoon. Aan de andere kant is het voor peer-to-peer collaboratie nodig dat de autonomie van de robot zich aanpast aan de huidige situatie.

De schaal geeft tevens het belang aan van een systeem waarmee mens en robot hun beste aspecten kunnen gebruiken op het juiste moment. zo'n systeem word *mixed-initiative interaction* genoemd, en is gedefinieërd als een flexibele interactie-strategie waarin elke agent (mens en robot) het meest geschikte bijdraagt op het meest geschikte moment [22].

2.1.2 Sense, (Plan), Act

Om autonomie te implementeren werden over de jaren heen een aantal paradigma's ontwikkeld. Het eerste paradigma was het sense-plan-act model [26], en werd gebruikt in een van de eerste autonome robot, Shakey [30]. De werking van het sense-plan-act model wordt weergegeven in figuur 2.2. Het model bestaat uit drie fases. In de eerste fase gaat de robot zijn sensoren *pollen* om een inputstaat te verkrijgen. Vervolgens wordt deze staat gebruikt

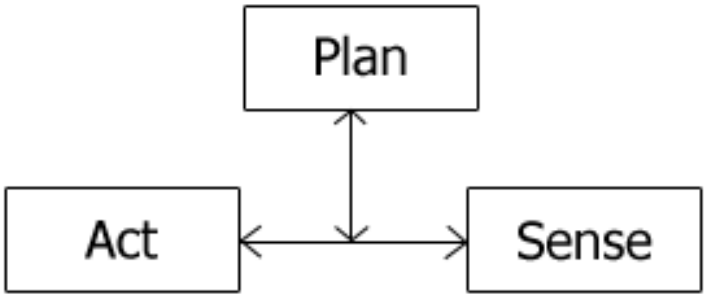
om de acties van de robot te plannen. Ten slotte worden de juiste signalen naar de actuators gestuurd waardoor de robot de geplande actie uitvoert.



Figuur 2.2: *Sense-plan-act paradigm* [26]. De robot loopt continue door de 3 primitieve fasen sense-plan-act.

Deze drie fases van het sense-plan-act model zijn tegenwoordig nog steeds fundamentele concepten bij het ontwikkelen van robots. Een voorbeeld van een model dat gebaseerd is op het sense-plan-act paradigma is *behavioral-based* autonomie. Bij dit model gaat men modules ontwikkelen die acties ondernemen als rechtstreekse reactie op inputs van de sensoren. Het gaat hier om een rechtstreekse *mapping* tussen sensor inputs en actuator bewegingen. Door een aantal van deze reactieve modules samen te voegen, kan een bepaald gedrag worden verkregen [2][7][8].

Tegenwoordig wordt autonomie vaak gerealiseerd door een hybride van de twee bovengenoemde paradigma's te implementeren. In zulke hybride systemen wordt de laag-level reactiviteit van de modules gescheiden van de hoog-level goals van de plan-fase, zoals wordt weergegeven in figuur 2.3



Figuur 2.3: *Hybrid paradigm* [26]. Bij een hybride architectuur wordt de hoog-level planningsfase gescheiden van de reactieve modules.

2.1.3 Informatieoverdracht

Autonomie is slechts een van de componenten die nodig is om een interactie nuttig te maken. Een tweede component is de manier waarop informatie wordt overgedragen tussen mens en robot. Net zoals bij autonomie zijn er een aantal karakteristieken die de efficiëntie van een interactie beschrijven. Zo geeft de interactietijd een indicatie van hoeveel tijd er nodig is om een bepaalde intentie of instructies aan de robot over te dragen [12]. Ook de cognitieve of mentale *workload* voor de mens is een belangrijke karakteristiek. Daarboven omschrijft *Situation awareness* hoeveel informatie over de huidige situatie wordt geproduceerd door de interactie. Indien de robot onderbrekingen heeft kan het zijn dat de *situation awareness* wordt gereduceerd.

Informatieoverdracht wordt gekenmerkt door 2 aspecten; Het medium en het formaat van de communicatie. Als medium worden over het algemeen 3 van de 5 zintuigen gebruikt. Deze zijn zicht, gehoor en aanraking. Een aantal voorbeelden van toepassingen voor deze zintuigen zijn als volgt:

- Visuele weergaves. Dit zijn vaak user interfaces die worden weergegeven op een scherm of augmented reality interfaces.
- Herkennen van gebaren. Zowel gezichts- als handgebaren kunnen worden herkend.

- Spraakherkenning en natuurlijke taal. Hierbij worden antwoorden via spraak of tekst gegeven. Hierbij wordt de nadruk vaak gelegd op *mixed-initiative* interactie.
- Het gebruik van *non-speech* audiosignalen. Deze soort interactie wordt vaak gebruik om gebruikers alert te maken van een gebeurtenis.
- Haptics. Het gebruik van fysieke feedback om een illusie van aanwezigheid te creëren.

Het formaat van de communicatie kan talloze vormen aannemen afhankelijk van het medium. Hier volgt een kort overzicht van de verschillende formaten en worden een aantal voorbeelden opgesomd:

- **Spraak** - bij spraak kan gekozen worden om de interactie te baseren op een formele taal en vervolgens te *scripten*, proberen een natuurlijke taal te ondersteunen of een subset van een natuurlijke taal te ondersteunen.
- **Haptics** - Voor haptische communicatie kan dan weer gekozen worden voor haptische vesten, haptische handschoenen, haptische feedback in het input-mechanisme (denk aan vibratie in een controller voor een spelconsole) of haptische iconen.
- **Audio** - Voor Audio-gebaseerde communicatie zijn er ook talloze opties, zoals informatie presenteren via audio, plaatsbepaling met behulp van 3D audio of het afspelen van waarschuwingssignalen.
- **Sociale informatie** - Sociale informatie kan overgedragen worden door gebaren te lezen, gezichtsexpressies, gepraat en natuurlijke taal, imitatie of het delen van een fysieke omgeving.
- **User interfaces** - User interfaces geven informatie weer door deze te tonen op allerlei media zoals schermen, augmented reality, virtual reality en traditionele venster-interacties.

Om de robuustheid, flexibiliteit, accuraatheid en ease-of-use van interacties te verbeteren, wordt recent veel onderzoek gedaan naar multimodale interfaces. Bij multimodale systemen worden inputs van verschillende kanalen gebruikt om zo informatie over te dragen. Multimodale interactie wordt verder besproken in sectie 2.2.

2.1.4 Training

Hoewel reeds veel onderzoek is verricht naar robot *learning* en *adaptation*, kreeg het trainen van mensen relatief weinig aandacht in het domein. Een van de doelen van HRI is nochtans het produceren van makkelijk te gebruiken systemen. Een mogelijke reden hiervoor is dat robots vaak ontwikkeld worden voor specifieke domeinen voor slechts korte periodes. Hierdoor is het aanleren van het systeem minder tijdrovend dan het implementeren van een makkelijk te gebruiken systeem[40][43]. Daarbij komt dat robot *learning* en *adaptation* als zeer nuttig worden beschouwd bij het designen en ontwikkelen van het gedrag van de robot.

Training is dus een belangrijk aspect binnen HRI en kan door de verschillende betrokkenen gebruikt worden om de mens-robot reactie te verbeteren. In de rest van deze sectie zullen de verschillende toepassingen van training worden besproken.

Ease-of-use: *Edutainment* robots zijn robots die gebruik worden binnen klaslokalen, musea, voor persoonlijk entertainment of voor thuisgebruik. Zulke robots moeten dus gebruikt kunnen worden door mensen met allerlei achtergronden. Daarom is het belangrijk dat deze robots zo weinig mogelijk training vereisen van hun operator. Training voor deze robots bestaat vaak uit een instructieboek, instructies door een onderzoeker of instructies verkregen van de robot zelf [31][41]. Onderzoek binnen deze tak bestaat vaak uit bestuderen hoe mensen omgaan met bepaalde robots; Zo werd een studie uitgevoerd waarbij mensen een ROOMBA moesten gebruiken zonder hen gebruiksinformatie te geven [20]. Ander belangrijk onderzoek in dit domein is onderzoeken hoe kinderen educatieve robots gebruiken in een leslokaal [23], Hoe gehandicapte kinderen omgaan met robots en het identificeren van interactiepatronen met gids robots in een museum.

Trainen van mensen: In tegenstelling tot het verminderen van de nodige training bij *edutainment* robots, vereisen sommige robots net een grondige training van hun operator. Dit is vaak het geval bij robots waarbij de operator een hoog risico loopt of waarvoor veel aandacht vereist is. Zulke robots worden meestal teruggevonden in militaire, politie, ruimtevaart of *search-and-rescue* toepassingen. Training voor deze soort robots bestaat vaak uit extensieve simulatietrainingen waarbij de nadruk wordt gelegd op het correct controleren van de robot, daar de operator vaak directe controle heeft over de robot. Andere belangrijke aspecten bij deze soort trainingen zijn het samenwerken met eventuele teamgenoten, veilig blijven tijdens het

gebruiken van de robot en het interpreteren van de video-feedback. Deze trainingen worden vaak gegeven aan mensen die reeds experts zijn in het vakdomein, maar kunnen evenzeer aan minder ervaren personen worden gegeven.

Trainen van designers: Een aspect dat vaak overkeken wordt is het trainen van designers. Wanneer designers de interface van een robot ontwikkelen, is het belangrijk dat ze op de hoogte zijn van het probleemdomein waarvoor de robot wordt ontwikkeld. Hierdoor hebben ze een beter overzicht van de functionele requirements van de robot en kunnen ze keuzes maken die de interface van de robot uiteindelijk ten goede zal komen. Voorbeelden van onderzoek naar het trainen van designers zijn *Murphy's workshops on search and rescue* [27] en tutorials en workshops voor het testen van robot interfaces [44].

Trainen van robots: Hoewel mensen trainen voor het besturen van een robot een voor de hand liggende keuze is, kan het trainen van de robots zelf niet worden uitgesloten. Er wordt veel onderzoek verricht naar robots die leren als deel van hun design [6][29] of zelfs tijdens hun live interacties [19][36]. Zo worden bijvoorbeeld de perceptuele, planning en autonome aspecten verbeterd door interacties te gebruiken om te leren. Deze manier van leren gebeurt op verschillende manieren; via demonstratie, *task learning* en *skill learning*. Binnen deze tak wordt ook onderzoek verricht naar leermethodes die gebruik maken van menselijke of dierlijke manieren van leren [3][37].

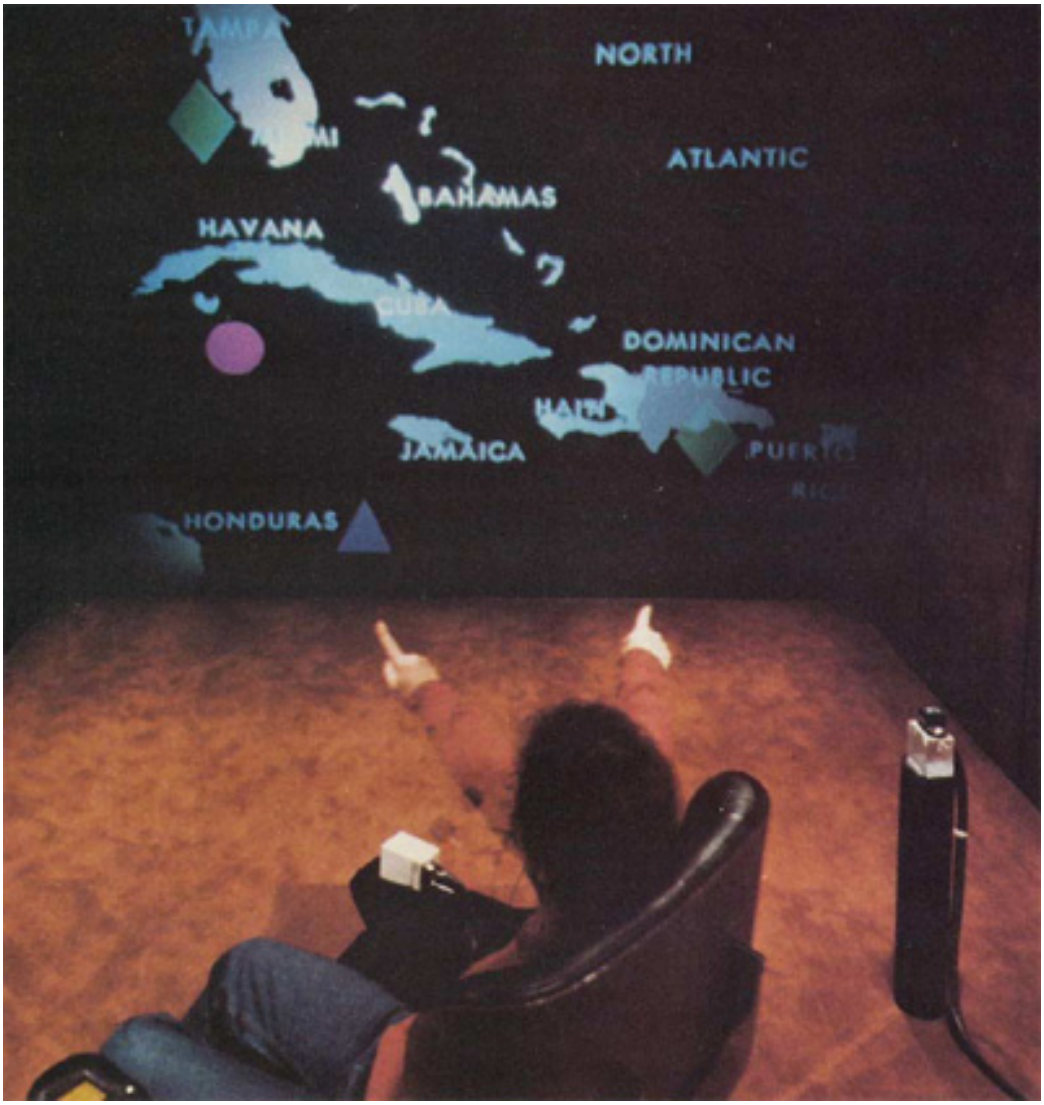
2.2 Multimodal Interaction

Menselijke interactie met de wereld is inherent multimodaal [9]. Mensen gebruiken verschillende zintuigen in zowel parallel als sequentie om passief en actief hun omgeving te verkennen en nieuwe informatie op te nemen. Zo worden geluidssignalen gebruikt om de locatie van een geluidsbron te identificeren, de grootte en kenmerken van de fysieke omgeving af te leiden of te verstaan wat iemand zegt. Door gebruik te maken van alle zintuigen kan een mens ongelofelijk veel informatie opnemen waarmee met de wereld kan worden geïnterageerd. In tegenstelling tot deze manier van interactie word bij interactie met een computersysteem vaak slechts één modaliteit gebruikt. Hoewel technisch gezien meerdere modaliteiten gebruikt worden, zijn deze voor het input of output kanaal bijna altijd hetzelfde. Zo wordt voor input vaak een combinatie van muis en toetsenbord gebruikt, terwijl output bijna vaak wordt weergegeven op een scherm.

Een multimodaal interface is een interactief systeem waarbij menselijke capaciteiten, zoals spraak, gebaren, aanraking, gezichtsuitdrukkingen en andere modaliteiten, gebruikt worden om een natuurlijke en interactieve manier van communicatie toelaten. Het doel van onderzoek naar multimodale interactie is het ontwikkelen van technologieën, interactiemethodes en interfaces die de huidige beperkingen met betrekking tot mens-computer interactie proberen weg te werken. In deze sectie wordt een korte geschiedenis van het domein gegeven, om vervolgens de problemen en opportuniteiten te bespreken. Hierbij wordt de focus gelegd op het probleem van integratie van de verschillende modaliteiten.

2.2.1 Geschiedenis

Het meest bekende voorbeeld van vroege multimodale interactie is het "Put That There"systeem van Richard Bolt [5], dat de waarde en opportuniteit van multimodale interfaces aangaf. Het systeem bestaat uit de *Media Room* waar spraak en gebaren geïntegreerd worden. Een gebruiker die zich in een stoel in de kamer bevindt kan hierdoor op een natuurlijke en efficiënte manier instructies aan het systeem geven. Als context voor het systeem word een data management systeem op een muur geprojecteerd (figuur 2.4).



Figuur 2.4: "Put that there" [5]. De Media Room waarin spraak en wijsgebaren worden geïnterpreteerd. Een *double exposure* effect geeft twee posities van de rechterarm weer.

De gebruiker kan binnen de *Media Room* zinnen als "put that there" of "create a blue square there" gebruiken om objecten in het systeem te manipuleren. Deze zinnen op zich zijn betekenisloos, aangezien aanwijzende woorden zoals "that" of "there" niet gebruikt kunnen worden om een object of locatie te identificeren. Door het toevoegen van een wijsgebaar bij het uitspreken van zulke woorden, zal het systeem deze woorden een betekenis kunnen geven, en wordt een multimodale interactie verkregen. Dit laat de gebruiker toe op een natuurlijke en expressieve manier met het systeem om

te gaan.

Het "put that there" systeem van Bolt heeft in de jaren 80 en 90 gezorgd voor een aantal multimodale systemen. Zo was CUBRICON [28] een kaart-gebaseerd tactisch *mission planning* systeem voor militaire doeleinden. Gebruikers konden door middel van een combinatie van natuurlijke taal, text en gebaren, interageren met het systeem. De output van het systeem bestond uit een combinatie van spraak, kaarten en grafische weergaven. Het Koons systeem [24] combineerde spraak, gebaren en staren in een kaart-gebaseerde applicatie. Ten slotte gebruikte het QuickSet systeem [10] pen en spraak als inputmodaliteiten. Het systeem draaide op een vroege tablet computer. De context van QuickSet was een militaire training simulatie voor het Amerikaanse *US Marine Corps*.

Uiteindelijk werden ook haptische input of feedback toegevoegd aan de beschikbare modaliteiten. Hierbij kwam dat de vooruitgang van mobiele apparaten een goed testplatform bood voor multimodale applicaties. Hoewel multimodale interactie als uitbreiding op de traditionele desktopervaring werd gezien, werd in het domein veel focus gelegd op post-WIMP interfaces. WIMP interfaces zijn traditionele desktop-interfaces waarbij windows, iconen, menus en pointers worden gebruikt.

2.2.2 Voordelen van multimodale interactie

Het doel van multimodale interactie is het ondersteunen van natuurlijke vormen van menselijke interactie zoals natuurlijke taal en gedrag. Hierdoor worden multimodale interfaces vaak ontworpen met een natuurlijke en efficiënte interactie in het achterhoofd. Hoewel slechts een klein aantal formele studies over voordelen van multimodale werden uitgevoerd, tonen verschillende minder formele studies aan dat multimodale interfaces een voordeel kunnen hebben over unimodale interfaces. Zo zouden gebruikers liever een multimodaal dan unimodaal interface gebruiken, wordt de flexibiliteit en robuustheid van de interface verbeterd of kunnen de input en output modaliteiten worden aangepast aan de gebruiker [46][47][33]. Bijkomende mogelijke voordelen van multimodale interfaces zijn [34]:

- Een verhoogde efficiëntie, vooral bij manipulatie van grafische informatie.
- Kortere en simpele spraakcommando's. Hierdoor zal tevens een robuuster interface worden verkregen.

- Verbetering van spatiale input ten opzichte van unimodale spraak interfaces. Hierbij wordt gedacht aan een pen of vinger om een locatie aan te geven.
- Verschillende combinaties van modaliteiten kunnen gebruikt worden om eenzelfde actie uit te voeren. Hierdoor kunnen allerlei mensen de interface gebruiken.
- Makkelijker om fouten te voorkomen en recht te zetten.
- Kan gebruikt worden om overmatig gebruik van een bepaalde modus te verhelpen.

Door de grote hoeveelheid combinaties van interfaces, taken, gebruikers en omgevingen zijn deze resultaten niet te veralgemenen. Er is echter wel een trend binnen de onderzoeken die aangeeft dat multimodale interactie een aantal voordelen voor de gebruikers met zich meebrengt.

2.2.3 Input en output modaliteiten

De termen die relevant zijn voor multimodale interactie kunnen verschillende betekenissen hebben afhankelijk van het domein waarin men zich bevindt. In deze sectie zullen de belangrijkste termen worden gedefiniëerd voor het HCI (en dus ook het HRI) domein.

Bij de perceptie van de wereld refereert een modaliteit naar het stimuleren van een van de vijf zintuigen: zicht, gehoor, aanraking, geur en smaak. Hoewel slechts een beperkt aantal systemen gebruik maken van smaak of geur, wordt modaliteit in HCI op dezelfde manier gedefiniëerd. Een *channel* wordt dan beschreven als een interactietechniek waarbij een gebruiker een apparaat hanteert met een bepaald effect. Een aantal voorbeelden van *channels* zijn tekst, geluid, spraakherkenning, beeld en aanwijzen en klikken met een pointer. Tabel 2.1 geeft een overzicht van de meestgebruikte modaliteiten en een aantal voorbeeld *channels*.

Modality	Example
Visual	Face location Gaze Facial expression Lipreading Face-based identity (and other user characteristics such as age, sec, race, etc.) Gesture (head/face, hands, body) Sign language
Auditory	Speech input Non-speech audio
Touch	Pressure Location and selection Gesture
Other sensors	Sensor-based motion capture

Tabel 2.1: Modaliteiten en voorbeelden met betrekking tot human-computer interactie [4].

Een multimodaal systeem kan dan beschreven worden als een systeem dat gebruik maakt van meerdere modaliteiten of meerdere *channels*. Multimodale systemen kunnen in een aantal aspecten verschillen: De gebruikte input modaliteiten, de gebruikte *channels*, het gebruik van parallelle of serieële modaliteiten, herkenningstechnieken, integratietechnieken en de ondersteunde applicaties.

2.2.4 Ontwikkelen van multimodale systemen

Het ontwikkelen van multimodale systemen is niet eenvoudig. Reeds bestaande ontwikkeltechnieken van WIMP applicaties kunnen vaak niet worden toegepast. Daarboven zullen gemaakte keuzes worden beïnvloed door de omgeving van het systeem. Hierbij wordt rekening gehouden met factoren zoals de verschillende modaliteiten, context, taken en gebruikers. in de praktijk wordt vaak gebruik gemaakt van een aantal mythes en richtlijnen. Deze bleken in de praktijk niet enkel handig bij het vergemakkelijken van het ontwikkelproces, maar ook bij het uitvoeren van onderzoek in het domein.

Een aantal interessante richtlijnen zijn als volgt [35]:

- Multimodale systemen moeten worden ontworpen zodat zoveel mogelijk soorten gebruikers en contexten worden ondersteund. Ontwerpers

moeten de modaliteiten of combinaties van modaliteiten gebruiken die het beste zouden werken in de gebruiksomgeving

- Er moet rekening worden gehouden met veiligheid en privacy. Bij openbare systemen moeten modaliteiten worden voorzien die de gebruiker toelaat zijn informatie op een veilige en private manier over te dragen
- Het systeem moet de cognitieve en fysieke capaciteiten van de gebruiker ten volste proberen te benutten
- Multimodale interfaces moeten zich aanpassen aan de gebruiker en de context waarin het zich bevindt. Zo zouden andere modaliteiten kunnen worden gebruik voor mensen met verschillende leeftijd of handicaps. Deze informatie kan bijvoorbeeld in user profiles worden bijgehouden
- Net zoals bij HCI is consistentie belangrijk
- Goede foutpreventie en foutafhandeling

Buiten deze richtlijnen werden een aantal mythes van multimodale systemen opgesomd en ontkracht [32]. Een aantal veelzeggende mythes die ontkracht werden zijn als volgt:

- *Gebruikers zullen een multimodaal systeem altijd multimodaal gebruiken.* Gebruikers zullen echter een mix van unimodale en multimodale kanalen gebruiken. Het is dus belangrijk dat goed uitgewerkte unimodale interfaces worden voorzien.
- *Multimodale inputs komen op hetzelfde moment voor.* Vaak worden multimodale inputs in sequentie verkregen in plaats van parallel.
- *Verschillende modaliteiten moeten een overlap van informatie bieden.* Het bieden van aanvullende informatie per modaliteit zou belangrijker kunnen zijn dan overlap.
- *Verhoogde efficiëntie is het grootste voordeel van multimodale systemen.* Hoewel multimodale systemen de efficiëntie kunnen verhogen, is dit niet altijd het geval. Voordelen zijn een vermindering in fouten, flexibiliteit en gebruiksvriendelijkheid.
- *Het combineren van error-prone modaliteiten maakt multimodale systemen onbetrouwbaar.* Door meerdere modaliteiten te combineren kan het doel van de interactie beter worden bepaald. Hierdoor wordt het systeem net meer betrouwbaar.

2.2.5 Integratie

Om een multimodale input te verkrijgen, worden signalen van verschillende modaliteiten gecombineerd. Aan de hand van integratie wordt er een betekenis gegeven aan deze combinaties. Deze kan echter afhangen van context, gebruiker, taak of tijd. Daarom is het samenvoegen of *integreren* van modaliteiten een moeilijke taak. De manier waarop dit gebeurt kan over het algemeen twee vormen aannemen. Zo kunnen modaliteiten vroeg of laat geïntegreerd worden.

Bij vroege integratie zullen signalen van verschillende modaliteiten samen geanalyseerd worden om een bepaalde semantiek te bepalen. Omdat deze signalen echter vaak zeer verschillend zijn, is het herkennen van bepaalde events in deze stroom van signalen niet eenvoudig. Vooral het temporeel aspect van de verschillende modaliteiten zorgt voor een probleem; Sommige inputs genereren data op discrete momenten, terwijl andere inputs stromen van informatie beschikbaar stellen. Hierdoor is vroege integratie een technisch uitdagende oplossing.

Bij late integratie worden de inputs van de modaliteiten apart geanalyseerd. Pas na het herkennen van de individuele events, worden deze samen bekeken om een multimodale input te verkrijgen. Hierdoor wordt de taak van inputherkenning bij de individuele modaliteiten gelegd. Aangezien er reeds goede implementaties bestaan van unimodale input herkenning (spraakherkenning, gesture herkenning, gaze, etc..), is late integratie makkelijker te implementeren. De Hasselt UIMS (zie sectie 2.3.5) implementeert late integratie.

Buiten vroege en late integratie, bestaat er ook een hybride techniek die midlevel integratie wordt genoemd. Hierbij worden inkomende signalen slechts eenvoudig verwerkt en geclassificeerd vooraleer ze worden samengevoegd.

Zowel vroege als late integratie hebben hun voor- en nadelen. Zo zal late integratie makkelijker te implementeren zijn omdat input per modulariteit wordt bepaald. Hier bestaat echter wel de kans da belangrijke inter-modale betekenis verloren gaat. Vroeger integratie daarentegen, zou deze inter-modale betekenissen wel kunnen definiëren en waarnemen. Dit is echter geen eenvoudig probleem, en zal een complexere oplossing vereisen.

2.2.6 Future work & uitdagingen

Hoewel Multimodale interactie een veelbelovend onderzoeksdomein is, moet er nog veel werk worden verricht vooraleer deze systemen onze huidige interactietechnieken vervangen. Naar de toekomst toe zullen zowel de interactietechnieken als de ontwikkeling van integratietechnieken een belangrijke focus zijn. Zo kan machine learning een belangrijke rol spelen bij het verbeter van interactietechnieken zoals spraakherkenning, gezichtsherkenning en gebarenherkenning. Hierbij wordt gedacht aan aanpasbaarheid, leermogelijkheid en personalisering. Wanneer de inputtechnieken onbetrouwbaar worden door slechte omstandigheden, zal het multimodaal systeem zijn nut verliezen. Daarom is het belangrijk dat herkenningstechnieken betrouwbaar zijn. Hier wordt gedacht aan spraak-, gezichts- en gebarenherkenning.

Ook bij het integreren van de verschillende modaliteiten moet nog veel onderzoek worden verricht: Huidige systemen integreren maximaal 2 modaliteiten. Grootschalige onderzoeken zullen gestart moeten worden waarbij een groter aantal modaliteiten op hetzelfde moment worden gebruikt. Hierbij zal vooral rekening moeten worden gehouden met de cognitieve mogelijkheid van de gebruikers. Zo zouden verschillende interactietechnieken een verschillende cognitieve last veroorzaken. Wanneer de cognitieve last van bepaalde interactietechnieken wordt vastgelegd, kan het systeem zich aanpassen aan de cognitieve capaciteit van de gebruiker.

2.3 User Interface Management Systems

Een User Interface Management System is een type architectuur waarbij de implementatie van de user interface losstaat van de eigenlijke functionaliteit van de applicatie. Het idee hierachter is dat de interface en de functionaliteit twee onderdelen zijn die volledig gescheiden van elkaar te implementeren zijn. Hierdoor kan de user interface makkelijk worden aangepast en uitgebreid zonder een direct effect op de semantiek of logica van de achterliggende applicatie. Bij het onderzoek naar mens-robot interactie wordt veel tijd besteed aan het ontwikkelen en testen van de, vaak multi-modale, user interfaces. Dit is duur en tevens niet eenvoudig. In deze context kan een UIMS gebruikt worden om snel prototypes uit te bouwen en het onderzoek te versnellen. In deze sectie zal eerst een categorisatie-methode besproken worden die ons toelaat een UIMS objectief te beoordelen. Vervolgens zullen een aantal systemen vergeleken worden aan de hand van deze methode. Ten slotte zullen eventuele tekortkomingen van de methode worden toegelicht.

2.3.1 Categorisatie-methode

Om een vergelijking te kunnen maken tussen de verschillende beschikbare tools, moeten deze objectief beoordeeld kunnen worden. Cuenca e.a. beschrijven een categorisatie-methode om tools in te delen in een van 3 categoriën [13]. Met een tool wordt een onafhankelijke applicatie bedoeld die bepaalde functionaliteit *triggered* in een gekoppelde client applicatie wanneer een bepaalde sequentie van events wordt afgevuurd. De categorisatie van deze tools gebeurt aan de hand van hun *scope*. Deze scope wordt gedefiniëerd als de hoeveelheid programmeerwerk nodig om bepaalde functionaliteit te verkrijgen.

Hoewel de tools uiteenlopende architecturen hebben, kan een bepaalde set functionaliteit in elk van de tools worden teruggevonden. Deze functionaliteiten zijn als volgt:

- **Input Recognizers**: De componenten van een tool die inputs herkent en omzet naar een bepaald formaat
- **Fusion Engines**: De component van een tool die verschillende events samenvoegd
- **Dialog managers**: De component van een tool die een grafische weergave van de dialoog weergeeft

- **Fission component**: De component van een tool die toelaat om acties pas te triggeren wanneer aan bepaalde condities is voldaan

De *scope* van de tool wordt dan gedefiniëerd als de set van functionaliteiten waarvoor een deel van de functionaliteit door de tool kan worden afgehandeld. Voorbeelden van de scope van een tool zijn Input Recognizers, Fusion Engine, Dialog Manager, Dialog Manager, Fission Component, enz.. De *scope* beschrijft dus de mate waarin het ontwikkelen en testen van multimodale interfaces vergemakkelijkt wordt.

Om te bepalen welke functionaliteit door de tool wordt ondersteund, werden volgende criteria gedefiniëerd:

- *(1) The toolkits that facilitate the implementation of the fusion engine are those that allow for composite events in their visual models.*
- *(2) The toolkits that facilitate the implementation of the dialog manager are those allowing the depiction of the system's states*
- *(3) The toolkits that facilitate the implementation of the fission component are those including language constructs for concurrency and synchronization.*

Door een aantal tools te analyseren werden drie categoriën ontdekt. Elk van deze categoriën omvat een subset van de verschillende functionaliteiten die tot de scope kan behoren:

- **Flow-based**: {Input Recognizers}
- **State-based**: {Input Recognizers, Fusion Engine, Dialog Manager}
- **Token-based**: {Input Recognizers, Fusion Engine, Dialog Manager, Fission Component}

Een tool kan nu eenvoudig geclassificeerd aan de hand van de visuele representatie. Om de bovenstaande categorisatie-methode verder bij te lichten zullen in de volgende secties een voorbeeld besproken worden voor elk van de categoriën.

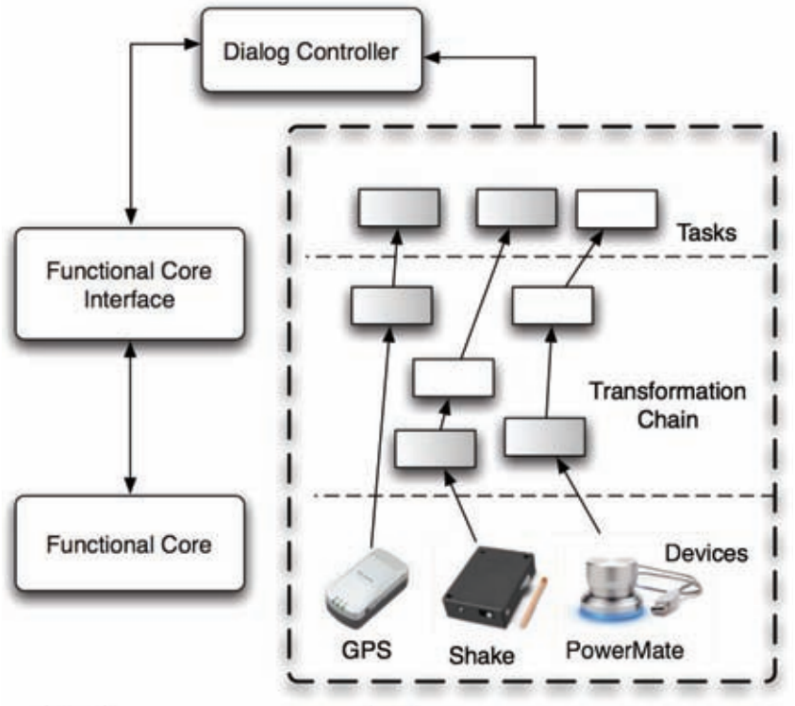
2.3.2 OpenInterface

OpenInterface is een component-gebaseerde tool die ontwikkeld werd om snelle ontwikkeling van multimodale interfaces toe te laten. Omdat multimodale interactie snel aan het groeien is, worden veel applicaties en tools ontwikkeld om de nieuwste input technologiën te ondersteunen. Deze tools en applicaties zijn vaak alleenstaande programma's die ontwikkeld worden voor een specifiek doel. OpenInterface probeert de manier van werken te generaliseren door gebruikers generieke componenten aan te bieden, alsook ze toe te laten zelf componenten te implementeren. Het OpenInterface framework bestaat uit twee componenten: De OI Kernel en de OpenInterface Interaction Development Environment (OIDE).

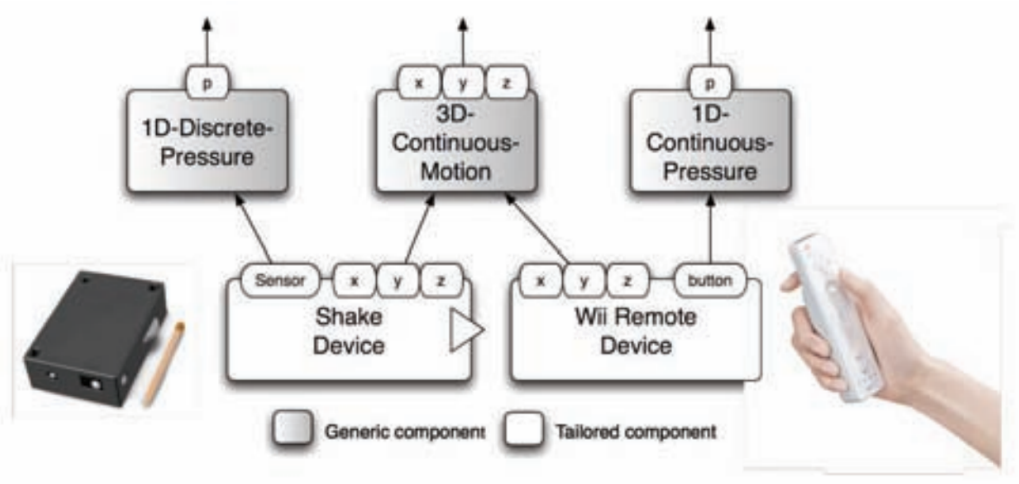
De OI Kernel ondersteunt componenten die geschreven zijn in allerlei talen (Java, C++, Matlab, Python, .NET). Deze heterogeniteit geeft de gebruiker veel vrijheid bij het ontwikkelen van zijn componenten. Ook wordt het overzetten van reeds bestaande code hierdoor vergemakkelijkt. Zo bevat de kernel functionaliteit om componenten te maken uit code zonder deze te veranderen. De Kernel beheert tevens het aanmaken en verbinden van de componenten door de dynamisch geladen *pipelines* te interpreteren. Deze *pipelines* worden gedefiniëerd in de OIDE.

De OIDE laat gebruikers toe componenten aan elkaar te schakelen en zo *pipelines* te definiëren die een multimodale actie beschrijven. OI voorziet een bibliotheek met componenten die allerlei functies uitvoeren zoals drivers voor hardware, interactie technieken, multimodale fusie en combinaties van componenten die door gebruikers worden samengesteld. Deze bibliotheek zit ingebouwd in de interface en kan dus eenvoudig geraadpleegd worden om snel nieuwe functionaliteit toe te voegen. Bovendien biedt de OIDE de mogelijkheid om zelf componenten te maken. Deze combinatie van voorgedefiniëerde en zelfgemaakte componenten maakt OpenInterface heel flexibel.

De OpenInterface *pipelines* bestaan uit drie lagen: *Device*, *Transformation Chain* en *Tasks*. De *device* laag bevat de componenten die signalen van inputapparaten (software en hardware) doorgeeft aan de *transformation* componenten. Deze componenten gaan deze signalen vervolgens transformeren en op hun beurt doorgeven aan de *task* laag, waar beslist wordt welke taak of actie zal worden uitgevoerd. Elk van deze lagen kan op zijn beurt aangepaste componenten bevatten die door de gebruiker zijn gedefiniëerd. In figuur 2.5 wordt de relatie tussen de lagen weergegeven.



Figuur 2.5: Het OpenInterface framework bestaat uit 3 lagen. Input komt in het framework via de *device*-laag, waarna deze inputs getransformeerd worden tot bruikbare data in de *transformation*-laag. Ten slotte wordt aan de hand van deze data beslist welke actie moet worden ondernomen in de *tasks*-laag.



Figuur 2.6: De componenten in OpenInterface worden gebruikt om een flow te definiëren waarbij output van de ene component als input dient voor de volgende. Op deze manier worden input-signalen omgevormd tot een te ondernemen actie.

Zoals te zien is in figuur 2.6, houdt OpenInterface geen state bij doorheen de uitvoering van een actie, en kan het geclassificeerd worden als een flow-gebaseerde tool. De *scope* van OpenInterface is dus {InputRecognizer}.

2.3.3 HephaisTK

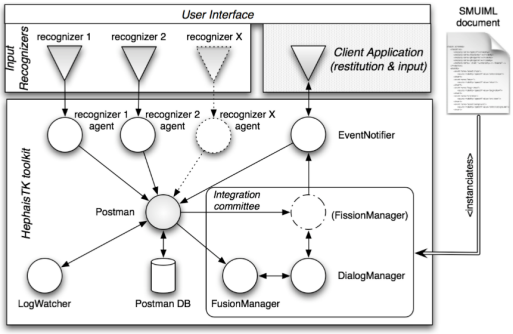
HephaistTK [18] is een grafische toolkit die ontwikkeld werd om snel prototypes van multimodale interfaces uit te werken zonder deze manueel te moeten implementeren. De toolkit werd ontworpen met het oog op het makkelijk koppelen aan een client applicatie. Wanneer gekoppeld met Hephais, zal de client applicatie notificaties ontvangen van multimodale events.

HephaisTK bestaat enerzijds uit de IMBuilder, een grafische user interface waarin multimodale interacties geconfigureerd kunnen worden. Deze configuraties worden opgeslagen in een XML-bestand. die interageert met de achterliggende *engine*. IMBuilder communiceert met het achterliggend multimodaal framework. Dit framework, genaamd MEngine, vormt de verschillende inputs om naar multimodale events die worden uitgestuurd naar de client applicatie.

HephaisTK gebruikt een agent-gebaseerd framework. Elk van de verschillende software-agenten is verantwoordelijke voor een specifieke taak. De *re-*

cognizer agenten zijn verantwoordelijk voor het herkennen van inputs en sturen deze door naar de *Postman* agent. De Postman verzamelt alle inputs en slaat deze op in een lokale database. Andere agenten in het framework kunnen de Postman laten weten dat ze geïntereseerd zijn in bepaalde data, waardoor ze geïnformeerd worden wanneer zulke data binnenkomt.

Buiten deze generieke agenten bevat het framework drie belangrijke agenten, het *'integration committee'*, die bestaat uit een fusion agent, een dialog agent en een fission agent. De fission agent is tevens verantwoordelijk van het informeren van de client applicatie van de multimodale events. Een overzicht van de architectuur van het framework wordt weergegeven in figuur 2.7

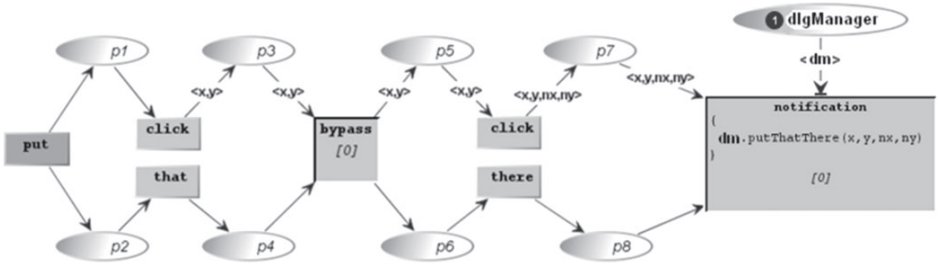


Figuur 2.7: De architectuur van het agent-gebaseerd multimodaal framework van HephaisTK [18]

Om HephaisTK te koppelen met een client applicatie, moet een configuratie worden opgesteld die de verschillende interacties met de applicatie beschrijft. Hierboven moet ook worden aangegeven welke input modaliteiten gebruikt zullen worden in de multimodale interacties. Met dit doel in ogen werd voor HephaisTK een speciale DSL ontwikkeld, genaamd SMUIML. Deze taal beschrijft de verschillende input events, hoe deze samengevoegd moeten worden en welke multimodale events de appliactie wilt ontvangen van SMUIML.

Ten slotte werd functionaliteit toegevoegd die toelaat input events te simuleren. Deze events kunnen op specifieke tijdstippen getriggered worden, waardoor specifieke use-cases eenvoudig en snel getest kunnen worden.

Aangezien de *'integration committee'* zowel een dialog als een fission manager bevat, kunnen we concluderen dat de scope van HephaisTK {Dialog Manager,



Figuur 2.8: [13]De put-that-there dialoog geïmplementeerd in PetShop.

Fission Engine} is. Hierdoor wordt het geclassificeerd als een state-based toolkit.

2.3.4 PetShop

PetShop is een component-gebaseerde toolkit die ontwikkeld werd om de tekortkomingen van het CORBA (*Common Object Request Broker Architecture* op te lossen. PetShop gebruikt onderliggend petri-nets om dit te realiseren. De belangrijkste voordelen ten opzichte van huidige oplossingen voor het definiëren van CORBA systemen zijn het toevoegen van gedragsspecificaties en het verminderen van low-level programmatiewerk. Hoewel PetShop niet specifiek ontwikkeld werd om multimodale systemen te ontwikkelen, biedt de makkelijk uitbreidbare omgeving een goed platform om dit te ondersteunen.

Een systeem wordt binnen PetShop gedefiniëerd door objecten die met elkaar communiceren aan de hand van de services en requests die ze implementeren. De configuratie van zo een systeem wordt ook wel een OPN (Open Petri Net) genoemd. De configuratie van een OPN wordt in de graphische interface weergegeven aan de hand van twee componenten: *places* en *transitions*. De flow van het systeem wordt bijgehouden door zogenaamde tokens, die tussen de *places* bewegen aan de hand van de *transitions*. De manier waarop deze tokens door de transities bewegen wordt gedefiniëerd aan de hand van *transition rules*. In figuur 2.8 wordt een OPN weergegeven die het bekende put-that-there scenario beschrijft.

De *transition rules* voor een OPN zijn eenvoudig: *places* zijn verbonden met *transitions* aan de hand van een van drie soorten pijlen. Zo zijn er gewone pijlen, testpijlen en verhinderende pijlen. Een pijl bevat in zijn annotatie de hoeveelheid tokens waarmee wordt gewerkt, alsook de informatie die wordt

meegegeven. Zo geeft de pijl van de *click* transitie naar P1 aan dat er 1 token wordt geproduceerd die de x en y coordinaten van de click-actie bevatten.

Een *transition* zal enkel worden afgevuurd wanneer alle inkomende pijlen voldoen aan hun voorwaarden. Voor de gewone en test-pijlen is deze voorwaarde simpelweg dat de *place* het aantal geannoteerde tokens moet bevatten. Bij een verhinderende pijl moet de *place* minder dan het geannoteerd aantal tokens bevatten. Bij het afvuren van een *transition* worden tokens van inkomende normale pijlen geconsumeerd door de *transition*. Testpijlen zorgen er dus voor dat tokens niet geconsumeerd worden. Wanneer een *transition* is afgevuurd, zullen uitgaande pijlen aangeven hoeveel tokens in elke *place* wordt geproduceerd, en welke data wordt meegegeven. Deze info wordt opnieuw bij de pijlen geannoteerd.

Hierboven kunnen *transitions* condities bevatten. Deze condities werken op de inkomende data, en kunnen de uitvoering van een transitie uitstellen tot aan bepaalde condities wordt voldaan. Deze condities kunnen gebruikt wordt om de flow van het systeem dynamisch te maken.

Een multimodaal systeem kan, zoals werd aangegeven in figuur 2.8, tot in detail beschreven worden aan de hand van een OPN. Zo wordt de interactiviteit van het systeem weergegeven door de flow van tokens doorheen de OPN. De input- en output events die het systeem kan waarnemen en triggeren, worden door de *transitions* ondersteund. Ten slotte kan ook de state van het systeem weergegeven worden aan de hand van de verdeling van de tokens over de *places*.

Aangezien PetShop een token-based toolkit is, zou de *scope* van de toolkit {Fusion Engine, Dialog Manageer, Fission Engine} moeten zijn. Input recognizers en de Fusion Engine worden ondersteund door middel van de *transitions*. Zo kunnen verschillende unimodale *transitions* tokens genereren, die op hun beurt geconsumeerd worden om een multimodaal event te genereren. De verdeling van tokens over de *places* weergeeft de staat van de dialoog. Ten slotte laten de verschillende soorten pijlen en de condities op *transitions* toe om events enkel af te vuren wanneer aan bepaalde condities wordt voldaan. We kunnen dus concluderen dat de scope van deze token-based toolkit inderdaad gelijk is aan {Fusion Engine, Dialog Manageer, Fission Engine}.

De Fission Engine is belangrijk om een robuuste multimodale interactie op te stellen. De toevoeging van tokens, verschillende soorten pijlen en conditionele *transitions* zorgen echter voor een verhoogde complexiteit. Dit zorgt

ervoor dat het begrijpen en configureren van een multimodaal systeem in een token-based toolkit niet voor de hand liggend is. Aangezien een van de belangrijkste doelstellingen van deze thesis een gebruiksvriendelijke tool is, kan dus beargumenteerd worden dat een token-based oplossing niet ideaal is.

2.3.5 Hasselt UIMS

Hasselt is een tool die het definiëren en testen van multimodale interacties ondersteunt.

- Composite events - Composite Event Description Language (CEDL)
- System Response Definition Language (SRDL) om te definiëren wat er gebeurt wanneer bepaalde events getriggered worden.
- State van het systeem wordt weergegeven aan de hand van een of meerdere finite state machines, die gemodeleerd wordt met de Humane Machine Dialog Definition Language (HMD2L).
- beschrijven hoe de verschillende talen met elkaar werken.
- UIMS bevat een editor, runtime environment en debugger.
- Plugins om functionaliteit toe te voegen.

Hasselt is een event-gebaseerde tool die ontwikkeld werd om het eenvoudig prototypen en testen van multimodale systemen toe te laten. Daarboven geeft het ontwikkelaars de optie om plugins te implementeren, die vervolgens gebruikt kunnen worden om bepaalde functionaliteit eenvoudig te importeren. Hierdoor kunnen plugins gedeeld worden binnen de community om snel functionaliteit aan een systeem toe te voegen. Hasselt omvat drie *Domain-Specific Languages* (DSLs), die elk gebruikt worden om een specifieke taak te vergemakkelijken.

Inputs worden door Hasselt geïmplementeerd door middel van Events. Zo worden unimodale events geïmplementeerd in plugins. Hasselt bevat reeds plugins die muis, keyboard, gesture en gezichtsuitdrukkingen omzetten naar evens. Deze unimodale events kunnen vervolgens binnen Hasselt in parallel of sequentie gecombineerd worden tot *Composite* events. Deze *composite* events stellen multimodale inputs voor. Het opstellen van *composite* events gebeurt aan de hand van de *Composite Event Definition Language* (CEDL). CEDL biedt een aantal speciale tekens die gebruikt kunnen worden om de relatie tussen unimodale events weer te geven:

+ AND operator om aan te geven dat beide events op hetzelfde moment moeten worden afgevuurd.

; FOLLOWED BY operator om aan te geven dat de events elkaar moeten opvolgen

| OR operator om aan te geven dat een van de twee events moet worden afgevuurd

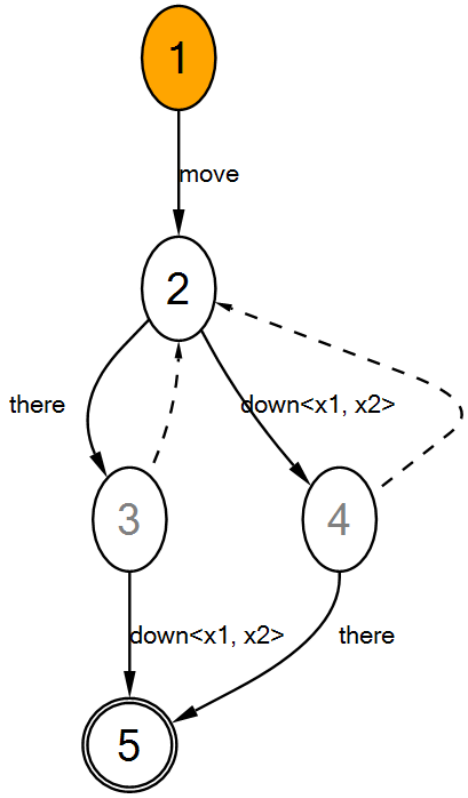
* ITERATION operator om een herhalende event weer te geven

Een voorbeeld van een composite event wordt weergegeven in figuur 2.9.

```
1 event moveEvent = speech.move; (speech.there + mouse.down<x1, x2>)
```

Figuur 2.9: Deze composite event 'move' zal afgevuurd worden wanneer de gebruiker *move* uitspreekt en vervolgens in parallel *there* zegt en ergens klikt.

Hasselt biedt een ingebouwde CEDL editor met auto-completion om *composite* events eenvoudig te configureren. Na het definiëren van de composite events, zal Hasselt Finite State Machines (FSMs) genereren die semantisch equivalent zijn. De gegenereerde FSM voor het 'move' event uit figuur 2.9 wordt weergegeven in figuur 2.10



Figuur 2.10: Hasselt genereert voor elke *Composite* event een FSM, die de gebruiker vervolgens kan gebruiken om enerzijds een overzicht te krijgen van de event. Ook kan deze FSM de huidige state van het event weergeven at run-time. De nummers die in de nodes staan worden gebruikt in de SRDL editor om aan te geven wanneer bepaalde acties moeten worden uitgevoerd.

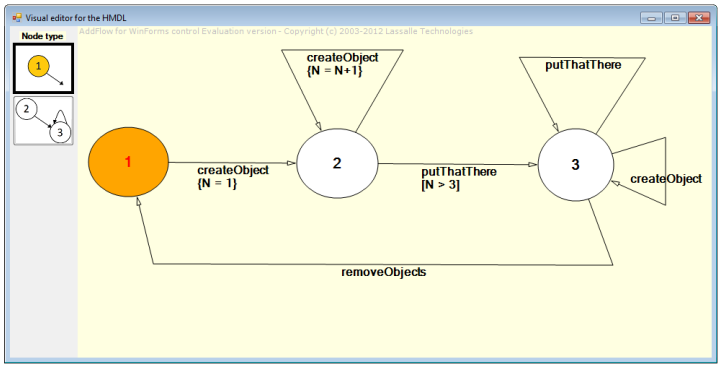
Hasselt laat toe om acties te definiëren die zullen worden uitgevoerd wanneer de state van de FSM verandert. Ook hiervoor voorziet Hasselt een DSL: *System Response Description Language* (SRDL). Aan de hand van SRDL kan de gebruiker per FSM (en dus per composite event) beschrijven welke acties moeten gebeuren op elke state van de interactie. Een gebruiker kan acties definiëren op een van twee verschillende componenten van een FSM: De *nodes* en de *links*. Een voorbeeld van een SRDL configuratie voor het 'move' event wordt weergegeven in figuur 2.11.

```
1  wrt ce.moveEvent
2     @link(1, speech.move):
3        speak: 'where?'
4     @node(5):
5        speak: 'moving'
6        call: clientAction.move<x1, x2>;
```

Figuur 2.11: De SRDL code die het 'move' event beschrijft. het *link* keyword geeft aan dat de actie gebeurt op de overgang tussen twee nodes. *node* geeft aan dat de actie in een node moet gebeuren. Deze code zal ervoor zorgen dat Hasselt feedback geeft wanneer de gebruiker "moveüitspreekt. Wanneer de gebruiker in de laatste node komt (node 5), zal Hasselt de gewenste actie oproepen in de client applicatie.

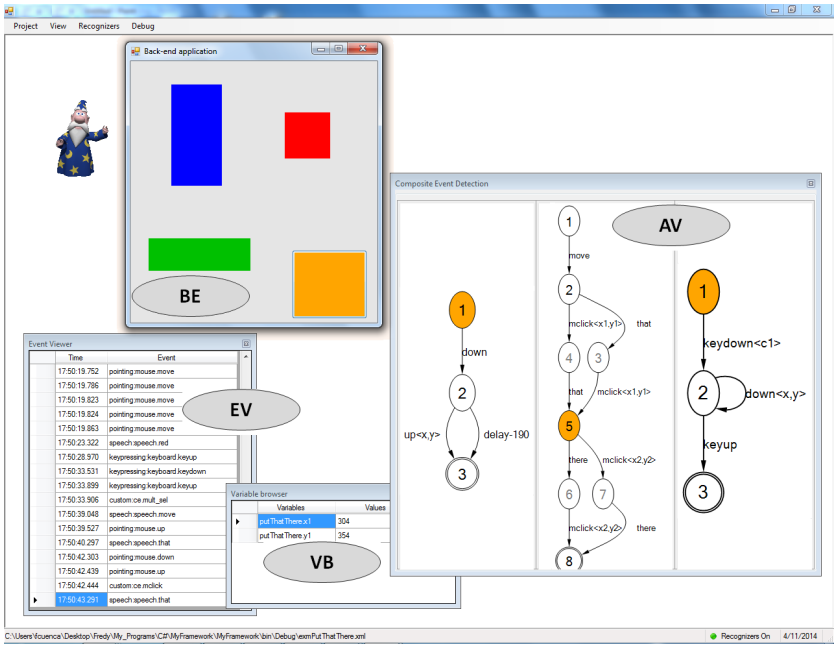
Bovendien bevat SRDL functionaliteit om guard aan events toe te voegen. Hierdoor kan gespecifiëerd worden dat een event enkel moet worden afgevuurd wanneer aan bepaalde voorwaarden voldaan is.

Zoals eerder beschreven, bevat Hasselt nog een derde DSL, namelijk de Human Machine Dialog Description Language (HMD2L). Waar CEDL en SRDL op een laag niveau de events en de acties definiëren, laat HMD2L toe om de volledige flow van het systeem te beschrijven: De gebruiker kan op een visuele manier aangeven in welke volgorde dialogen elkaar kunnen opvolgen. Op die manier kan het verloop van het "gesprek"met het systeem worden gemodelleerd. Om dit te realiseren implementeert HMD2L een *extended* finite state machine. Dit is een FSM waarbij condities kunnen worden toegekend aan bepaalde overgangen. Ook kunnen state variabelen worden bijgehouden. Een voorbeeld van een HMD2L diagram wordt weergegeven in figuur 2.12.



Figuur 2.12: Deze figuur weergeeft een extended finite state machine gedefiniëerd aan de hand van HMD2L [14]. De recthoekige haakjes geven condities aan. Gekrulde haakjes geven manipulatie van de state variabelen weer. Dit diagram beschrijft een gesprek waarbij een gebruiker objecten kan creëren, ze kan verplaatsen en vervolgens weer kan verwijderen. Zoals de conditie [N ¿ 3] aangeeft, zal de gebruiker pas objecten kunnen verplaatsen nadat hij 4 objecten heeft aangemaakt. Op dit moment zal ook de composite event *removeObjects* kunnen worden afgevuurd.

Hasselt werd ontwikkeld met als doel het vergemakkelijken van het prototypen en testen van multimodale systemen. De complexiteit van het definiëren en configureren van multimodale interacties zorgt ervoor dat goede debugging tools van groot belang zijn. Daarom voorziet Hasselt een aantal debugging tools die worden weergegeven in figuur 2.13.



Figuur 2.13: de verschillende *debugging* tools beschikbaar in Hasselt [14]. **EV:** De Event Viewer geeft een overzicht van de events die afgevuurd werden. **AV:** De Automata View geeft een overzicht van de verschillende interacties en de state waarin ze zich bevinden. **VB:** De Variable Viewer geeft een overzicht van de variabelen van een event. **BE:** De back-end applicatie wordt door Hasselt weergegeven in een apart scherm.

Zoals het gebruik van FSMs impliceert, is Hasselt een state-based toolkit. Om die reden is de *scope* van Hasselt gelijk aan die van HephaisTK, namelijk {Input Recognizer, Fusion Manager, Dialog Manager}. Hoewel deze scope hetzelfde is als die van HephaisTK, voorziet Hasselt functionaliteit die het eenvoudiger maakt om multimodale interacties te implementeren. Zo geeft het gebruik van 3 DSLs veel controle over elk niveau van abstractie. Daarboven zijn de plugins makkelijker te definiëren aan de hand van enkele klassen. Ten slotte voorziet Hasselt debugging tools die waardevolle informatie kunnen geven en de implementatie van een eventuele uitbreiding kunnen vergemakkelijken.

Conclusie

In deze sectie werd een overzicht gegeven van de verschillende categorieën waartoe een UIMS kan behoren. Ook werden de voor- en nadelen van deze categorieën besproken aan de hand van een aantal concrete UIMS. Om het prototypen van mens-robot te vereenvoudigen, moeten we bij het kiezen van een UIMS rekening houden met het doelpubliek van de tool. Het gaat hier vaak om mensen die geen programmeerkennis hebben. Om die reden is het belangrijk dat de tool een goede interface bevat die niet te complex is. Ook het is belangrijk dat de code van de tool beschikbaar is, zodat eventuele nodige aanpassingen gedaan kunnen worden om een nuttige toolkit te bekomen.

Een vereiste van een makkelijk bruikbaar prototyping toolkit is dat een state van het systeem wordt weergegeven. In prototyping is dit belangrijk, aangezien zo achterhaald kan worden of alle interacties naar behoren werken en zijn gedefiniëerd. Aangezien de scope van een flow-based toolkit geen dialog manager bevat, worden flow-based toolkits reeds uitgesloten. Hoewel token-based toolkits de breedste scope hebben, moet de complexiteit van zulke tools in achting worden genomen. Zo zal het opstellen van een complexe interactie configuratie vereisen van een groot aantal transities. Aangezien mens-robot interactie multidisciplinair is, kan verondersteld worden dat een onderzoeker deel uitmaakt van een team. Wanneer deze complexe diagrammen geïnterpreteerd moeten worden door mede-onderzoekers, kunnen de mogelijke voordelen teniet worden gedaan door de verhoogde complexiteit. Om die reden worden ook token-based toolkits uitgesloten.

State-based toolkits bieden een gulden middenweg voor de implementatie van de tool. Ze bevatten een dialog manager die de huidige staat van een systeem kan weergeven, weliswaar op een begrijpbare manier. Zoals hierboven vermeld, is toegang tot de source code van de originele toolkit ook van belang. Een vergelijking van de besproken state-based toolkits toont aan dat Hasselt de beste kandidaat is om op verder te bouwen om de doelstellingen te behalen: Buiten het toevoegen van *debugging* tools, laat Hasselt het eenvoudig implementeren van plugins toe. Bovendien is de source doe van Hasselt volledig beschikbaar en kan de hulp ingeroepen worden van onderzoekers die Hasselt ontwikkeld hebben. aan dat een kandidaat goed geschikt is om onze applicatie op te bouwen.

2.4 Robot Programming Frameworks

Robot-instructies werden in het verleden vaak gecodeerd aan de hand van gepatenteerde talen en frameworks. Hierdoor was de interoperabiliteit tussen de verschillende soorten robots vaak onbestaand of heel moeilijk te verkrijgen. Om dit probleem op te lossen, zijn over de jaren heen een aantal frameworks ontwikkeld. Elk van deze frameworks mikt op het standaardiseren van de robot-industrie op een gedistribueerde manier.

In deze sectie zal een overzicht worden gegeven van een aantal van deze frameworks. Ook wordt gekeken welke voordelen de verschillende frameworks kunnen bieden voor het behalen van de gestelde doelstellingen.

2.4.1 Player Project

Player is een open-source gedistribueerd robot framework dat ontwikkeld werd om de algemene workflow met robots te vereenvoudigen. Player hecht tevens grote waarde aan het hergebruiken van zogenaamde *drivers*, die een interface naar sensoren of robots implementeren. Een gebruiker kan bijvoorbeeld een *driver* downloaden voor een bepaalde sensor, waarna hij de sensor op een hoog niveau kan aanspreken. Player werd ontwikkeld met het oog op het beheren van meerdere robots in een netwerk.

Player implementeerde oorspronkelijk een client/server architectuur waarbij communicatie gebeurde via het TCP/IP protocol. Deze oplossing is echter niet schaalbaar genoeg, en vereiste dus een meer generieke oplossing. In versie 2.0 van Player werd gekozen om het framework op te splitsen in twee delen: de *core* en transport layer. Hierdoor kunnen gebruikers zelf kiezen welke transport protocol ze gebruiken om informatie door te geven, hoewel collet et al. denken dat TCP/IP nog altijd het meest gebruikt zal worden [11].

Player 2.0 gebruikt een publish/subscribe systeem, waarbij elke server een binnenkomende queue heeft. Wanneer een server een bericht van een bepaald type published, zal elke server die voor dat type bericht gesubscribed is, het bericht ontvangen in zijn queue. De servers kunnen tevens rechtstreeks communiceren met elkaar wanneer om bijvoorbeeld services of berekeningen uit te voeren.

Buiten het Player robot framework werden twee simulatieomgevingen ontwikkeld: Gazebo en Stage. Hierbij is Gazebo een 3D simulatie, waar Stage slechts 2D is. Deze simulatieomgevingen gebruiken heuristieken om de in-

teractie van robots met de omgeving rondom hen te benaderen. Hierdoor kunnen een redelijk groot aantal robots gesimuleerd worden op middelmatige hardware, hoewel de simulaties niet volledig realistisch kan zijn. De simulatieomgevingen laten echter toe om toekomstige robots te prototypen vooraleer ze effectief gebouwd worden.

Player wordt goed ondersteund op alle POSIX besturingssystemen. Hoewel volledige ondersteuning voor Windows een van de doelstellingen is, is deze momenteel slechts gelimiteerd. Daarbij komt dat C# niet ondersteund wordt voor het schrijven de *drivers*. Dit maakt dat het werkend krijgen van een simpele robot op Windows niet eenvoudig is. Hoewel de simulatieomgevingen van Stage en Gazebo veelbelovend zijn voor het prototypen van de implementatie van deze thesis, lijkt het erop dat het opzetten van de volledige Player/Stage omgeving niet eenvoudig zal zijn onder Windows.

2.4.2 OROCOS

Het Open Robot Control Software of OROCOS is een robot framework dat ontwikkeld werd om een aantal problemen van robot software tegen te gaan. Een gebrek aan open-source robot software zorgden ervoor dat interoperabiliteit tussen verschillende robots en hun software pakketten nagenoeg onmogelijk was. Zo werden robots geleverd met binaire programmacode die vaak enkel in één situatie bruikbaar was. Dit zorgde ervoor dat het ontwikkelen en onderzoeken van complexe robots trager ging dan zou mogen.

Om dit op te lossen, werd OROCOS ontwikkeld als een open-source gedistribueerd framework. In de design-fase werden een aantal lange-termijn doelstellingen opgesteld. Hoewel deze fanatiek zijn, geven ze wel een goed beeld van wat de visie is van OROCOS:

- Voorzien in alle robot control software noden. Hiermee word specifiek de software bedoeld, en ziet men OROCOS dus als het framework waarmee alle verschillende aspecten van een robot kunnen worden aangestuurd.
- Gedistribueerde nodes zijn beschikbaar via LAN en WAN.
- OROCOS zal geïmplementeerd en beschikbaar zijn voor alle operating systems en *off-the-shelf* hardware

Hierboven specifieert OROCOS nog een doel met betrekking tot de ontwikkeling van de verschillende onderdelen van het framework. Er wordt hierbij uitgegaan van 4 doelgroepen binnen het framework. Elk van deze doelgroepen zal het framework op zijn eigen manier gebruiken, en het is dus belangrijk dat voldaan wordt aan hun noden:

- Framework Builders: Deze ontwikkelaars zijn verantwoordelijk voor het ontwikkelen van het OROCOS framework. Zij krijgen de taak om de juiste architectuur te implementeren om een bruikbaar component-gebaseerde oplossing te bekomen.
- Component Builders: Deze ontwikkelaars zijn verantwoordelijk voor het schrijven van functionaliteit, deze te bundelen in componenten en vrij te geven aan de andere gebruikers binnen de community.
- Application Builders: Deze ontwikkelaars zijn verantwoordelijk voor het leveren van werkende robot systemen. Ze doen dit door het OROCOS framework te configureren naar hun doelpubliek, en vervolgens componenten samen te voegen om een bepaalde set van functionaliteit te bekomen.
- End Users: De eindgebruikers van het framework zijn de mensen die de robots en hun software gebruiken om dagdagelijks taken uit te voeren. De robot systemen die ze gebruiken worden geleverd door de *Application Builders*

OROCOS ziet er veelbelovend uit met zijn architectuur die verantwoordelijkheid legt bij verschillende doelgroepen. Ook de flexibiliteit van de component-gebaseerde aanpak kan naar de toekomst toe zeer handig zijn. Met de doelstellingen van deze thesis in het achterhoofd, lijkt de scope van OROCOS echter iets te breed. Het is namelijk niet nodig dat een volledig robot systeem wordt uitgewerkt. Daarom wordt gekozen om verder te zoeken naar andere kandidaten als robot platform om deze thesis mee uit te werken.

2.4.3 Robot Operating System

De complexiteit en kwantiteit van beschikbare robots zijn over de jaren heen blijven stijgen. Dit heeft ervoor gezorgd dat het vakdomein heterogeen is geworden. Zo werden, voor onderzoeksdoeleinden, talloze robot-frameworks

ontwikkeld die bepaalde aspecten van robot-programmatie eenvoudiger maken. Hierdoor is het schrijven van herbruikbare code voor robots niet triviaal. Net als zovele frameworks, is ook ROS ontstaan uit een onderzoeksachtergrond. Bij het ontwikkelen van ROS werden volgende doelstellingen voorop geplaatst:

Peer-to-Peer

Wanneer robots verbonden zijn in een heterogeen netwerk, kan een centrale server problemen veroorzaken: Wanneer grote hoeveelheden data, bijvoorbeeld van een compute-node die grafische gegevens verwerkt, via de centrale server verdeeld moet worden, zal dit een negatieve invloed hebben op de throughput van het netwerk. Aangezien robots vaak data delen over interne subnets, is een peer-to-peer oplossing gewenster. Hierdoor zullen de reeds trage Wifi-verbindingen niet onnodig worden belast. Er moet echter wel een manier worden voorzien om peers met elkaar te laten communiceren. Hiertoe wordt een master-node opgezet die verder in deze sectie zal worden beschreven.

Tools-based & Thin

In plaats van een monolithische structuur te ontwikkelen waarbij driver logica verwikkeld is met de middleware, werd de functionaliteit van ROS bewust in kleinere componenten opgedeeld. Deze onderdelen implementeren elk een klein onderdeel van de volledige functionaliteit. Dit, gecombineerd met het gebruik van CMake, laat toe dat ROS enkel de nodige modules samenvoegt tot een uitvoerbaar bestand. Deze granulariteit laat het eenvoudig ontwikkelen van bijkomende modules toe en houdt de uitvoerbare bestanden tot een minimum.

Multi-lingual

Voorkeuren voor bepaalde programmeertalen zijn uiteenlopend. Hierdoor werd ROS ontwikkeld als *language neutral* platform. Om dit te bereiken beperkt de implementatie van ROS zich tot de *messaging layer*. Hierdoor word ROS al officiëel ondersteund in 4 talen: C++, Python, Octave en LISP. Buiten deze officiële implementaties worden ook wrappers, zoals de ROS.NET wrapper voor C#, ontwikkeld door derden. Om het implementeren van de ROS functionaliteit zo eenvoudig en cross-language mogelijk te houden, maakt ROS gebruik van de Interface Definition Language (IDL).

Deze wordt gebruikt om ROS berichten te omschrijven op een taal-neutrale manier. De implementaties van ROS moet dus ook functionaliteit voorzien om deze IDL om te vormen naar objecten binnen de gekozen programmeertaal.

Free and open-source

Zoals al eerder werd beschreven, kan functionaliteit door ontwikkelaars worden toegevoegd. ROS is bovendien volledig open-source. Hierdoor kan het platform snel groeien en uitgebreid worden voor talloze verschillende hard- en software.

De architectuur van ROS bevindt zich op de *messaging layer*. De verschillende onderdelen van ROS lijken dan ook hard op van binnen een message-passing patroon. Zo wordt de functionaliteit van ROS omschreven door 4 belangrijke onderdelen: *Nodes, Messages, Services en Topics*. Hierbij zijn Nodes de processen die binnen een ROS architectuur berekeningen doen. Alle *Nodes* en hun verbindingen met elkaar in een ROS architectuur kunnen worden weergegeven in een graaf. Deze *Nodes* communiceren met elkaar door het doorgeven van *Messages*. Deze zijn getypeerde berichten die primitieve types kunnen bevatten alsook andere *Messages* en arrays. *Messages* kunnen tevens arbitrair genest worden.

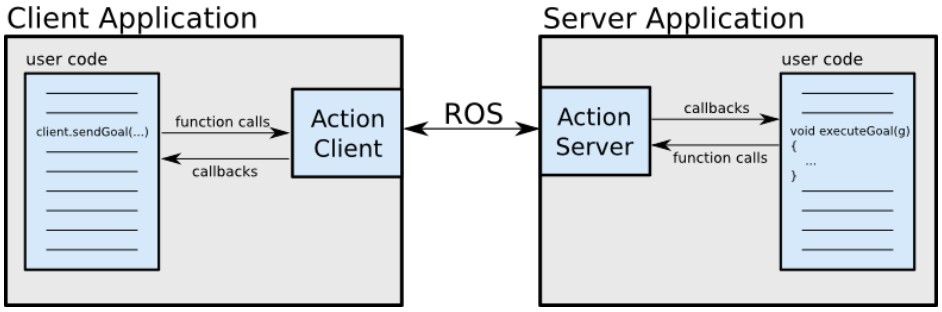
Messages worden doorgegeven aan andere *Nodes* door ze te publiceren op een bepaalde *Topic*. *Nodes* die geïnteresseerd zijn in de informatie kunne zich dan abonneren op de gewenste *Topic*. Ten slotte zijn *Services* synchrone interacties die op commando kunnen worden uitgevoerd. Deze omvat twee speciale *Messages*, een voor de request en een voor het resultaat van de service.

Actionlib

ROS is een protocol dat handig is om rechtstreeks te communiceren met allerlei robot-software, mits deze het ROS protocol ondersteund. In een complex robot-systeem zullen er gevallen zijn waarbij tijds-intensieve acties moeten worden uitgevoerd. ROS biedt hier ondersteuning voor aan de hand van *services*, maar deze zijn bedoeld voor korte-termijn acties en bieden geen optie tot feedback of annulering van acties.

De actionlib library bevat laat toe om zogenaamde *action servers* te programmeren. Deze action servers ontvangen *goals*, waardoor ze een bepaalde actie zullen uitvoeren. De action servers hebben de mogelijkheid om bepaalde speciale ROS berichten uit te sturen, waarmee de huidige staat van de actie kan worden uitgelezen. Een actie die aan het lopen is op een *action server* kan tevens geannuleerd worden.

Buiten de *action server*, levert actionlib ook een *action client*, die het versturen van *goals*, het luisteren naar feedback en het annuleren van acties vergemakkelijkt. Een high-level overzicht van de manier waarop de actionlib client en server communiceren wordt weergegeven in figuur 2.14.



Figuur 2.14: Een action client en een action server communiceren via het ROS protocol. Een action client verstuurt een *goal* naar een action server, die de actie uitvoert. De action server houdt de client op de hoogte van de vooruitgang via *feedback* berichten. Een client kan de actie op elk moment onderbreken door middel van een *cancel* bericht. Wanneer de actie is uitgevoerd, stuurt de server een *result* bericht naar de client.

een actie wordt gedefiniëerd in een *àction* bestand, en bestaat uit drie onderdelen:

Goal Het Goal bericht bevat informatie met betrekking tot het doel van de actie. Voor een beweging kan dit bijvoorbeeld de bestemde positie zijn. Deze informatie kan dan door de action server worden uitgelezen om de nodige acties te ondernemen.

Feedback Het Feedback bericht is een optioneel bericht dat door een action server kan worden uitgestuurd om feedback te geven aan de client. Bij een bewegings-actie kan bijvoorbeeld de huidige snelheid periodiek worden teruggestuurd.

Result Het Result bericht wordt gebruikt wanneer een actie beïndigd wordt. Dit kan zijn omwille van het correct uitvoeren van de actie of het manueel beëindigen ervan. Het Result bericht wordt precies 1 keer uitgestuurd. Een Result bericht wordt meestal gebruikt om bepaalde data terug te sturen die voortkomen uit de actie. Bij een beweging kan bijvoorbeeld de eindpositie worden teruggestuurd.

Simple Action Server

Een *Simple Action Server* is een gesimplificeerde versie van een action server die slechts 1 doel tegelijk verwerkt. De regels die deze server hanteert zijn als volgt:

- Er kan maar 1 doel actief zijn op elk moment
- Binnenkomende doelen annuleren bestaande doelen met een lagere timestamp en worden uitgevoerd
- Een speciaal annuleringsdoel kan alle doelen met een lagere timestamp annuleren.
- Wanneer een nieuw doel geaccepteerd wordt, zullen alle oude doelen geannuleerd worden en zal hun status veranderd worden. Dit impliceert dus ook dat een Result bericht verstuurd wordt die de annulering van de doelen aangeeft.

Deze simpele action server kan gebruikt worden om een simpele eerste versie te maken voor een robot. Zo kan de basis functionaliteit getest worden zonder het beheer van alle concurrente doelen.

Conclusie

In deze sectie werd gekeken naar een aantal robot frameworks. Het idee achter de besproken frameworks is hetzelfde: Het vergemakkelijken van de ontwikkeling van software voor robots. Elk framework doet dit ook op een gedistribueerde manier, zodat de community gebruik kan maken van componenten of modules die door andere gebruikers werden ontwikkeld.

De frameworks verschillen echter op het toepassingsgebied. Waar OROCOS ontwikkeld werd voor de ondersteuning van instructies met bepaalde realtime vereisten, richten Player en ROS zich meer op een multi-robot scenario.

De complexiteit van de ontwikkeling en setup ligt dan ook lager bij deze twee laatste.

Ten slotte biedt ROS betere ondersteuning voor het Windows platform, waar veel van de UIMS in worden ondersteund. Zo werd een ROS wrapper gemaakt in C#, waardoor ROS gebruikt kan worden in een .NET omgeving. Daarboven is ROS zeer licht, makkelijk te installeren en tevens een van de meestgebruikte robot software. Om deze redenen werd beslist om ROS te gebruiken als robot framework voor de tool.

2.5 Conclusie

Het doel van deze thesis is het uitbreiden van de Hasselt UIMS (zie sectie 2.3.5) zodat mens-robot interacties eenvoudig te definiëren zijn. Als communicatie protocol werd gekozen om ROS te gebruiken. Dit protocol wordt voor een groot aantal robots gebruikt om snel een interface op te stellen waarmee deze kunnen worden aangestuurd. Daarboven is er geen speciale hardware nodig om ROS te ondersteunen. Door de grote beschikbaarheid van het protocol, is dit tevens een ideale keuze om de functionaliteit van de tool te kunnen testen op reeds bestaande implementaties voor robots.

Om een nuttige interactie met een robot te verkrijgen is meer nodig dan het simpelweg uitwisselen van informatie. Vaak zal een taak toegekend worden aan een robot. De robot zal de taak uitvoeren en, indien nodig, feedback over de voortgang rapporteren. Daarom is het belangrijk dat de tool die ontwikkeld wordt hier rekening mee houdt. Om de implementatie van deze functionaliteit te ondersteunen, kan voor ROS gebruik gemaakt worden van de actionlib laag (sectie 2.4.3). Deze laat toe om actie-resultaat interacties op te stellen via het ROS protocol.

De functionaliteit van de tool is slechts één deel van het verhaal. Om het vakgebied van mens-robot interactie te ondersteunen, is het belangrijk dat de interface van een robot eenvoudig kan worden aangepast. Hierdoor kunnen onderzoekers een interface definiëren, deze testen, en vervolgens snel aanpassen wanneer er iets fout zit. Hoewel een werkend prototype van deze functionaliteit reeds geïmplementeerd werd voor de Hasselt UIMS, is deze implementatie niet makkelijk aan te passen of uit te breiden. Zo moeten een aantal C# klassen worden geschreven die de nodige events (van en naar de robot) genereren. Dit impliceert ook dat deze klassen opnieuw gecompileerd moeten worden wanneer veranderingen nodig zijn.

Aangezien mens-robot een multidisciplinair vakgebied is, zal een tool die dit vakgebied ondersteund eenvoudig in het gebruik moeten zijn. Hierdoor zullen ook onderzoekers zonder technische achtergrond snel de interfaces van de robots kunnen definiëren en gebruiken.

In de volgende sectie worden de doelstellingen van deze thesis opgesteld en nader toegelicht aan de hand van de net besproken punten.

3 Doelstelling

Hoofdstuk 2 gaf een overzicht van de huidige technologiën binnen multimodale interactie. Hieruit werd op basis van de vergaarde informatie een technologiën uitgekozen om op verder te werken. In deze sectie zullen deze keuzes en de doelstellingen voor deze thesis besproken worden.

3.1 Ondersteuning voor ROS

Zoals vermeld in sectie 2.5, werd reeds een prototype voor het ROS protocol geïmplementeerd voor de Hasselt UIMS. Deze schiet echter te kort wanneer wijzigingen moeten worden aangebracht. Zo moet voor elk event een functie worden gedefiniëerd die beheert wat er gebeurt als een event wordt gedetecteerd of moet worden afgevuurd. Deze oplossing is echter niet generiek genoeg, aangezien zelfs simpele robots zoals Sphero al tientallen *topics* vrijgeven. Om dit probleem op te lossen, worden voor de tool volgende vereisten opgesteld om ROS te ondersteunen:

- (1) De gebruiker moet eenvoudig de interface van een robot kunnen definiëren en gebruiken in de Hasselt UIMS. Hierbij worden zowel input als output van en naar de robot ondersteund.
- (2) Functionaliteit moet kunnen worden aangepast of toegevoegd zonder hercompilatie van de code. Dit impliceert dat de interface van een robot *at runtime* kan worden toegevoegd aan de Hasselt UIMS.
- (3) De functionaliteit van de tool moet kunnen worden overgedragen naar een ander toestel zodat deze eenvoudig gedeeld en gebruikt kan worden door een groot aantal personen.

3.2 Ondersteuning voor actionlib

Het ondersteunen van actionlib is een belangrijk onderdeel van deze thesis. Robots zullen vaak acties uitvoeren die enige tijd duren. Ondersteuning voor het configureren van zulke acties zal de bruikbaarheid van de tool verhogen. Om een *action* te definiëren, moeten 2 implementaties worden gemaakt: De *action client* en de *action server*. De *action client* verstuurt actiedoelen naar

de *action server*, die de acties uitvoert en rapporteert over hun voortgang. In de context van robots komt de *action server* overeen met de robot die de actie uitvoert en de *action client* met de software die de robot delegeert. Daar de *action client* een implementatie vereist in een van de ondersteunde talen en moet integreren in Hasselt, zal ook hiervoor een generieke oplossing moeten worden bedacht. We kunnen de vereisten voor actionlib dus als volgt opsommen:

- (4) De gebruiker moet op een eenvoudige manier *action clients* aan Hasselt kunnen toevoegen.

Hierboven zijn ook (2) en (3) van toepassing op de ondersteuning van actionlib.

3.3 Ease of Use

Zoals (1), (2) en (4) impliceren, zal het algemene doel van de tool zijn om onderzoek naar mens-robot interacties te vergemakkelijken. De tool zal dus een manier moeten voorzien om zowel action clients als ROS events te definiëren. Voor ease of use worden volgende doelstellingen opgesteld:

- (5) De tool moet een interface voorzien om ROS events te definiëren en te configureren.
- (6) De tool moet een interface voorzien om acties te definiëren en te configureren.
- (7) De tool moet een manier voorzien om de action clients eenvoudig te gebruiken in Hasselt.

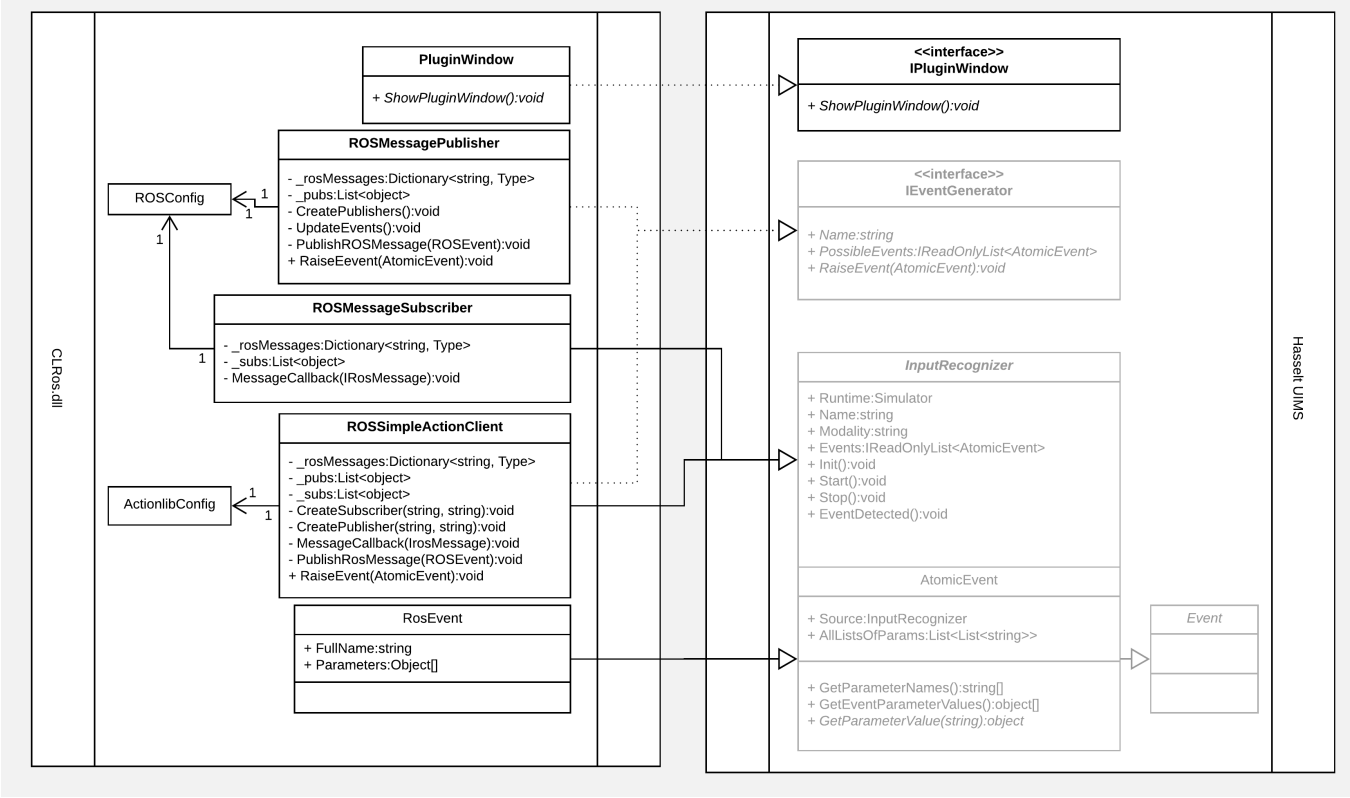
4 Implementatie

In deze sectie zat de implementatie van de Hasselt uitbreiding besproken worden. Allereerst wordt een overzicht gegeven van de architectuur aan de hand van een UML. Hierna wordt verder ingegaan op de manier waarop ondersteuning voor ROS en Actionlib werden toegevoegd.

4.1 Architectuur

De architectuur van de tool is gebaseerd op die van de plugins voor Hasselt: Hasselt stelt namelijk een klasse en een interface ter beschikking die geïmplementeerd kunnen worden om plugins te definiëren. Deze zijn de *InputRecognizer* klasse en de *IEventGenerator* interface. De *InputRecognizer* klasse wordt gebruikt om inputs om te vormen naar events binnen Hasselt. De *IEventGenerator* interface definiëert de functies die nodig zijn om events uit Hasselt op te vangen en acties te ondernemen. Een overzicht van de relatie van de tool tot de architectuur van Hasselt wordt weergegeven in figuur 4.1. Hasselt maakt gebruik van een interface en een klasse omdat *multiple inheritance* niet ondersteund wordt in C#.

Om het gebruik van de tool eenvoudig te maken, moet tevens een manier voorzien worden om een configuratie-scherm te weergeven. Hiervoor word door Hasselt echter geen ondersteuning geboden. Daarom werd besloten om een interface aan Hasselt toe te voegen. De *IPluginWindow* interface is een simpele interface die een hook blootstelt om een WPF dialoog te weergeven. De Hasselt Interface werd tevens uitgebreid met een menu-item waarmee de configuratieschermen van de plugins kunnen worden opgeroepen (zie figuur 5.1). Op deze manier kunnen ontwikkelaars van een Plugin op een gestandaardiseerde manier configuratie van de Plugin toelaten.



Figuur 4.1: De tool maakt gebruik van klassen en interfaces die gedefiniëerd worden door Hasselt. Er werd echter beslist om nog een interface aan Hasselt toe te voegen. De IPluginWindow laat plugins toe om een configuratiescherm te voorzien die automatisch in Hasselt wordt ingeladen. De elementen in het zwart werden door de tool toegevoegd.

Zoals wordt weergegeven in figuur 4.1, is de tool die ontwikkeld werd een plugin voor Hasselt. Aangezien Actionlib een uitbreiding is van ROS, werd gekozen om Actionlib en ROS in eenzelfde plugin te implementeren. Hierdoor kon de gezamenlijke ROS setup hergebruikt worden.

In de rest van deze sectie zal besproken worden hoe de implementatie van de ROS en Actionlib plugin in zijn werk is gegaan. Zo wordt de nodige setup besproken, alsook de problemen die overkomen moesten worden en de geïmplementeerde oplossingen.

4.2 ROS

Aangezien Actionlib een uitbreiding is van ROS, was het logisch om te beginnen met het ondersteunen van ROS. Allereerst werd geprobeerd een werkende ROS configuratie te verkrijgen. Op het moment van de implementatie was de ROS implementatie voor Windows in beta. De volledige functionaliteit kon dus niet gegarandeerd worden. Hoewel een .NET implementatie van ROS gevonden werd, bleek deze niet volledig te werken: hoewel deze toeliet een ROS node op te starten, was het niet mogelijk om een master node op te zetten. Aangezien een master node vereist is om de communicatie tussen de afzonderlijke nodes te beheren, werd gekozen om een virtuele machine aan te maken. Deze virtuele machine draait ubuntu 16.04 en wordt vooral gebruikt om de master node op te draaien. De linux installatie van ROS laat tevens toe eenvoudig custom ROS-berichten te sturen. Deze functionaliteit werd dan ook meerdere malen gebruikt om snel testen uit te voeren op.

In de rest van deze sectie zullen we spreken over een *inkomend event* wanneer een ROS bericht via de *InputRecognizer* binnenkomt en zo een state verandering triggert in Hasselt. Een *uitgaand event* komt voor wanneer een state verandering ertoe leidt dat een ROS bericht uitgestuurd wordt.

4.2.1 Eerste implementatie

Allereerst werd gekozen om twee eenvoudige ROS applicaties te schrijven, de *talker* en *listener*. Deze applicaties publishen en subscriben naar een topic, respectievelijk. Aan de hand van deze eerste werkende implementatie kon dan een werkende statische implementatie van een plugin worden gemaakt. Deze vuurde een aantal events in Hasselt af, gebaseerd op de tekst die ontvangen werd door de listener. Er werd ook een publisher gemaakt die tekst kon versturen op een topic, afhankelijk van het event dat manueel

geïmplementeerd werd.

Door deze simpele plugin te implementeren werd al snel tot de conclusie gekomen dat de .NET implementatie van ROS niet gedefiniëerd was met het doel van deze thesis in ogen. Zo wordt voor elk berichttype een klasse gegenereerd. Een *string* wordt bijvoorbeeld omvat door de *Messages.std_msgs.String* klasse. Om een publisher of subscriber van een bepaalde type aan te maken, moet een generieke functie opgeroepen worden met het gewenste type. Zo wordt voor het maken van een subscriber de functie *node.subscribe¡Messages.std_msgs.String¿()* opgeroepen. Hoewel deze manier generiek is, laat het niet toe om eenvoudig de types van een subscriber of publisher aan te geven *at-runtime*. Om dit op te lossen, kan echter gebruik gemaakt worden van reflectie om subscribers en publisher te genereren aan de hand van een messagetype, zoals later in deze sectie wordt besproken.

Nadat de statische plugin volledig geïntegreerd was met Hasselt werden de eerste stappen gezet richting een dynamische plugin. Allereerst werd er een configuratiescherm ontwikkelt. Hierbij werd rekening gehouden met de vooropgestelde doelstellingen. Zo werden dialogen voorzien waarin allerlei parameters van een event kan worden aangepast. Hierbij werd ook geprobeerd de dialogen zo gebruiksvriendelijk mogelijk te maken. Zo werd bijvoorbeeld bij het instellen van het berichttype een zoekfunctie geïmplementeerd. Aangezien de lijst van standaard berichttypes reeds honderden types bevat, maakt dit het makkelijk om het juiste type te vinden. Een gedetailleerde omschrijving van de verschillende configuratieschermen wordt besproken in sectie 6.

4.2.2 Dynamische publisher en subscriber

Voor het maken van een dynamische publisher en subscriber werd eerst onderzocht welke vereisten deze zouden hebben. Het implementeren van de statische plugin heeft hier een goed overzicht van gegeven. In deze sectie zal een overzicht worden gegevens van de problemen die zullen moeten worden opgelost. Vervolgens worden de gevonden oplossingen besproken. Om de plugin te testen, werd gekozen om een Sphero-robot te gebruiken. De Sphero-robot is een eenvoudige robot die bestuurd kan worden via ROS berichten. Zo kan een richting en snelheid worden aangegeven, een draairichting worden ingesteld en kunnen de interne LED's op een kleur worden ingesteld. Een volledige use-case met de robot wordt besproken in sectie 6.

De problemen waarvoor een oplossing gevonden moet worden zijn als volgt:

(P1) Sommige berichttypes bevatten parameters die niet altijd gebruikt worden. Zo bestaat er een algemeen type dat een beweging aanduid. Dit type, "Twist lx ly lz ax ay az", bevat parameters die zowel een lineaire als een angulaire beweging kunnen aangeven. Wanneer bijvoorbeeld een event "Vooruit"wordt gedefiniëerd, zal de angulaire beweging altijd nul zijn. De publisher moet deze waarden echter meegeven als zijnde nul. Er moet dus een manier voorzien worden waardoor enkel die parameters moeten worden meegegeven die betrekking hebben tot de situatie. Hierdoor kan het topic "Twist 0 0 1 0 0 0"gesimplificeerd worden tot "Beweeg 0 0 1öf zelfs "Vooruit 1". Dit probleem is enkel van toepassing op uitgaande events.

(P2) Buiten overbodige parameters zullen events ook parameters bevatten die altijd dezelfde waarde hebben. Er moet dus een manier voorzien worden om deze waarden een standaard waarde mee te geven. Als we het voorbeeld uit P1 nemen, kan "Vooruit 1"Gesimplificeerd worden tot "Vooruit", terwijl "Vooruit -1"dan "Achteruit"zou worden. Dit probleem is enkel van toepassing op uitgaande events.

(P3) Hasselt laat toe om condities vast te hangen aan events, zodat deze enkel getriggered worden indien aan de condities voldaan wordt. Dit kan de code echter onleesbaar maken. Wanneer bijvoorbeeld het event "Beweeg 0 0 1"binnenkomt, zal de Hasselt editor code moeten bevatten die kijkt of de derde parameter positief of negatief is om te kijken of het een voorwaartse of achterwaartse beweging is. Om dit op te lossen moeten condities dus op de events zelf kunnen worden gedefiniëerd. Dit probleem is enkel van toepassing op inkomende events

(P4) ROS berichten bevatten geneste types. Zo kan een bericht een ander type ROS bericht bevatten, wat op zijn beurt opnieuw een ROS type kan bevatten. Wanneer deze typen dynamisch moeten kunnen worden opgevuld, zal er iets voorzien moeten worden waarmee waarden kunnen worden toegekend aan het diepste niveau van de hiërarchie van een ROS bericht.

(P5) Als we ervan uitgaan dat een oplossing voor P1 is gevonden, bestaat de mogelijkheid dat de signature van een event ambigu is. Zo zal een event "Beweeg lx ly lz"niet te onderscheiden zijn van het event "Beweeg ax ay az". Er moet dus een manier voorzien worden om duidelijk te maken welk event getriggered moet worden binnen Hasselt.

(P6) Configuraties van Hasselt events moeten kunnen worden opgeslagen zodat ze tussen sessies door blijven bestaan, en ze ook met anderen gedeeld kunnen worden. Hiertoe moet een taal zoals XML of JSON of iets dergelijks gebruikt worden om de configuraties naar de harde schijf *at runtime* weg te schrijven.

Tijdens de implementatie van de tool zijn volgende oplossingen voor de hierboven vermelde problemen geïmplementeerd:

(P1) Om dit probleem op te lossen kan bij het aanmaken van een nieuw ROS Event de groepen parameters worden aangegeven die beschikbaar zijn. Wanneer het event dan in Hasselt wordt ingetypt, worden de verschillende opties weergegeven door Hasselt. Wanneer het event gegenereerd wordt, zullen de parameters doorgegeven worden aan de plugin. Hier wordt gekeken welke parameteroptie overeenkomt met de meegegeven parameters. Deze parameteropties bevatten tevens informatie over de variabelenaam in het ROS berichttype, zodat de juiste waarde aan het juiste veld wordt gekoppeld.

(P2) Om dit probleem op te lossen wordt tevens een dialoog voorzien waarin standaard-waarden voor de verschillende variabelen kunnen worden aangegeven. Hier wordt een mapping bijgehouden van de namen van variabelen en hun waarden. Alle waarden die voorkomen in de lijst van standaardvariabelen worden eerst geïnitialiseerd op hun standaard waarde, waarna ze overschreven worden indien ze aanwezig zijn in het uitgaande event.

(P3) Ook voor dit probleem wordt een dialoog voorzien die toelaat condities te definiëren. Zo kunnen aan een variabele een of meerdere condities worden vastgehangen. Het inkomend event dat deze condities heeft zal enkel worden getriggered in Hasselt indien aan alle condities wordt voldaan. Deze condities houden ook de naam van de variabele bij, zodat eenvoudig gekeken kan worden of inderdaad aan de condities voldaan wordt.

(P4) Wanneer een ROS type wordt toegekend aan een event, zal deze volledig overlopen worden, Wanneer een nieuw type gevonden wordt, zal deze op zijn beurt volledig overlopen worden. Dit blijft doorgaan tot alle primitieve variabelen gevonden zijn. Deze worden dan opgesomd zodat bekend is tot welke sub-variabele deze behoort. Als voorbeeld nemen we het fictief type Locatie, dat de variabele "loc"bevat. Deze variabele is van type Vector3, en

bevat op zijn beurt 3 primitieve variabelen met type float. Alle variabelen van type Locatie worden dan als volgt weergegeven in het configuratiescherm:

- loc.x
- loc.y
- loc.z

Op deze manier kunnen aan alle variabelen in een ROS type een waarde worden toegekend. Een Hasselt event dat als ROS type "loc"heeft kan dan opgeroepen worden als "SendLocation 10 1 10". Deze manier van werken wordt ook gebruikt bij het instellen van condities of standaard waarden.

(P5) Aangezien dit probleem inherent is aan functies met gelijke *signatures*, werd besloten dat een oplossing zoeken voor dit probleem een te grote tijdsinvestering zou zijn, als een oplossing zelfs mogelijk was. Men kan tevens beargumenteren dat twee events *Beweeg lx ly lz* en *Beweeg ax ay az* kunnen worden opgesplitst in *Beweeg lx ly lz* en *Draai ax ay az*, waardoor ze niet langer ambigu zijn.

(P6) Als technologie om de events weg te schrijven werd gekozen om XML te gebruiken. XML wordt goed ondersteund in zo goed als alle programmeertalen en is daarboven ook lees- en aanpasbaar voor mensen. C# bevat daarboven klassen die het wegschrijven en inlezen van een volledige klasse naar en van een XML trivialiseren.

4.3 Actionlib

4.3.1 Eerste implementatie

Net zoals bij ROS, werd er bij de actionlib implementatie eerst een afzonderlijke configuratie gemaakt die buiten Hasselt werkte. Deze testconfiguratie was een tutorial van actionlib zelf, en bestond uit een action server die een fibonacci reeks berekende aan de hand van de input van een action client. Ook hier werd gekeken naar eventuele problemen die zich zouden kunnen voordoen bij het implementeren in de Hasselt plugin. Net zoals bij de ROS implementatie, zijn door deze eerste versie een aantal problemen aan het licht gekomen. De belangrijkste zijn als volgt:

(P1) Er werd besloten om ook voor actionlib gebruik te maken van een Sphero robot. Hier bestond op het moment van de implementatie echter geen action server voor. Er werd echter wel een python library gevonden die de verschillende variabelen van de Sphero-robot kon aanspreken via het ROS protocol. Er moet dus iets voorzien worden zodat deze library aangesproken kan worden via een action server.

(P2) Een action client bevat callback functies die aangeroepen worden bij het terugkrijgen van een *feedback* bericht van de action server. Er moet dus een manier zijn om deze berichten op te vangen zonder de uitvoering van het programma te blokkeren.

(P3) Communicatie tussen een action-client en een action-server gebeurt met speciale ROS berichten. Buiten een bericht dat het doel omvat, bevatten deze ook meta-informatie over het doel. Deze meta-informatie dient op de juiste manier ingevuld te worden.

(P4)* De problemen die bij de ROS implementatie werden tegengekomen, zullen ook bij actionlib terugkomen. Ook voor actionlib moet hier een, eventueel soortgelijke, oplossing worden voorzien.

4.3.2 Dynamische action clients

De eerste stap richting een dynamische action client was het implementeren van een statische client. Hierdoor konden onvoorziene problemen snel ontdekt worden. Omdat reeds een action client en serve geïmplementeerd werd om een fibonacci reeks te berekenen, werd beslist om deze ook te implementeren in Hasselt. De implementatie die gemaakt werd ontving een input,

waarna een doel werd gestuurd naar de fibonacci action server. Deze stuurde op zijn beurt feedback over elk berekend element in de reeks, met een artificële delay van 1 second. De ingebouwde Hasselt spraak functionaliteit werd gebruikt om elk van de nummers op te zeggen. Wanneer de reeks tot het gewenste getal berekend was, werd tevens *done* uitgesproken.

Vervolgens werd een configuratiescherm gemaakt dat een gelijkaardige functionaliteit had als die van de ROS implementatie. Het aanmaken van een actie in dit venster zorgt ervoor dat automatisch alle nodige ROS subscribers en publishers worden opgezet. De drie belangrijkste zijn de Goal publisher, de Feedback subscriber en de Result subscriber. De onderlinge werking tussen dezen wordt besproken in (P2).

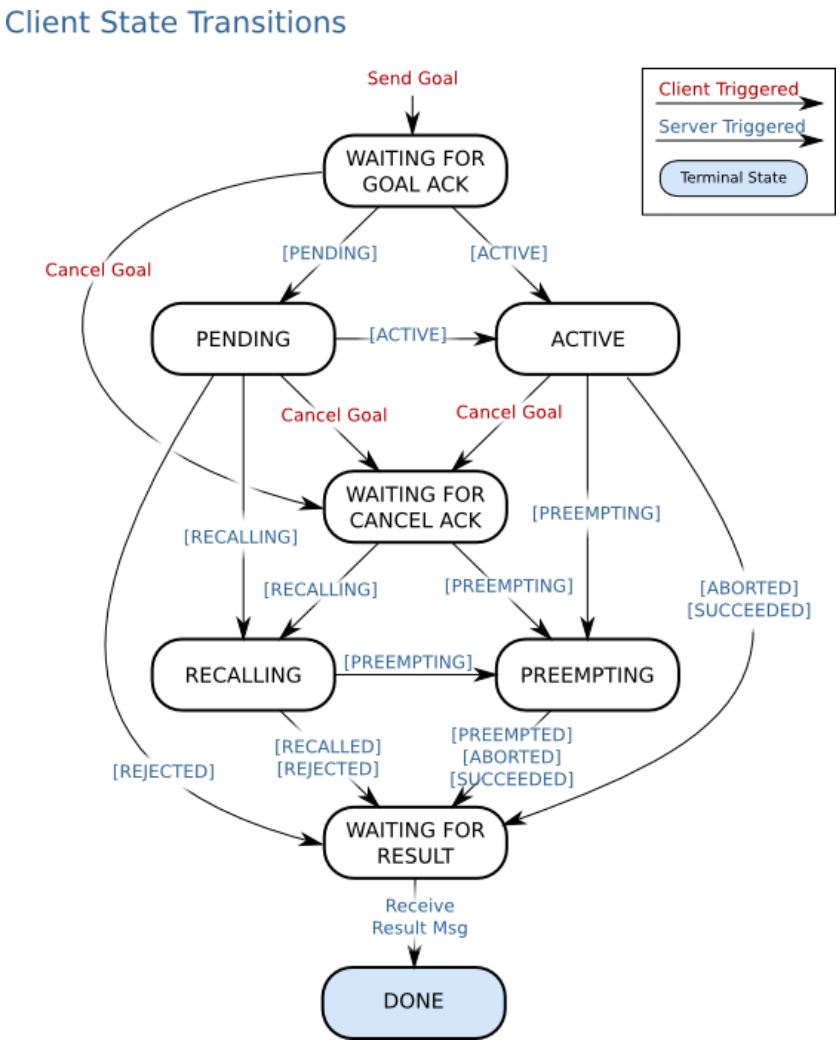
(P1) Om met de Sphero robot te interageren via een action client, werd besloten om zelf een simpele action server te implementeren. Deze action server draait op python en gebruik ROS om de eerder vermelde python library voor Sphero aan te spreken. De simpele action server bevat volgende functionaliteit:

- Roll: Rol in een bepaalde richting volgens het XZ-vlak, over een bepaalde afstand. Sphero laat weten wanneer de afstand is afgelegd en stopt vanzelf. Eventueel kan de actie onderbroken worden.
- ChangeColor: Stel de kleur van de Sphero in.
- Rotate: Laat de Sphero draaien in een bepaalde richting rond zijn Y-as. Sphero blijft draaien tot de actie onderbroken wordt.
- Dance: Rol en draai tegelijk tot de actie onderbroken wordt.

Een uitgewerkte use-case die deze functionaliteit gebruikt wordt besproken in sectie 6.

(P2) De action server is slechts een van de twee componenten die nodig zijn om actionlib te ondersteunen. De action client stuurt doelen naar de server en krijgt periodiek berichten terug met de status van de actie. Origineel werd gepland om de action client tevens in een aparte applicatie te ontwikkelen, en die vervolgens te linken met Hasselt. Als we echter kijken naar een overzicht van een action client volgens de actionlib wiki, dan zien we dat dit een state machine is, zoals weergegeven in figuur 4.2. Aangezien

Hasselt achterliggend finite state machines gebruikt, is het mogelijk dat de state machine van een action client in Hasselt kan worden nagebootst.



Figuur 4.2:

(P3) De action client is normaal gezien verantwoordelijk voor het opvullen van de nodige meta-data. Een naïve client kan de meta-data echter leeg laten, waardoor de action server deze waarden zal opvullen. Door een domme client te maken, zal deze echter geen weet hebben van de lopende acties. In complexere robots kan het zijn dat meerdere acties tegelijkertijd worden uitgevoerd, zoals bijvoorbeeld het bewegen van een robot naar een locatie terwijl de vastgemaakte robot-arm ook een bepaalde actie uitvoert. Om een

specifieke actie te kunnen stoppen is het dus belangrijk dat de action client ID's toekent aan de acties.

Buiten een ID wordt ook de huidige timestamp toegekend aan het bericht, zodat alle acties die binnenkwamen voor een bepaalde tijdstip kunnen worden stopgezet. Dit kan worden gebruikt om alle acties tegelijk te stoppen wanneer de huidige timestamp wordt meegegeven aan een *cancel* bericht.

(P4)* Dezelfde oplossingen werden gebruikt als bij ROS.

4.4 Conclusie

In deze sectie werd de implementatie van ROS en actionlib besproken. Aangezien actionlib gebruik maakt van het ROS protocol, werd de ROS implementatie eerst volledig uitgewerkt. Dit bracht enkele mogelijke problemen aan het licht (4.2.2). Deze problemen werden, indien mogelijk en passend binnen de doelstellingen van deze thesis, zo goed mogelijk opgelost. De oplossing voor deze problemen in ROS konden tevens hergebruikt worden in de implementatie van actionlib.

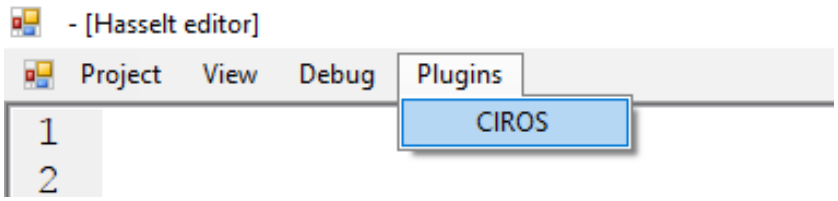
actionlib gebruikt een client-server model waarbij zowel de client als de server finite state machines zijn. Aangezien Hasselt achterliggend FSMs gebruikt, werd besloten om de implementatie van de action client volledig in de Hasselt talen te definiëren. Deze oplossing bleek te werken, en zorgt er tevens voor dat een action client, door middel van de Hasselt UI, eenvoudig kan worden geconfigureerd zonder dat hiervoor hercompilatie nodig is.

5 Functionaliteit

In deze sectie zal de functionaliteit van de geïmplementeerde tool in detail worden besproken. In sectie 6 vindt u een volledig uitgewerkte use-case voor de tool toegepast op een Sphero robot.

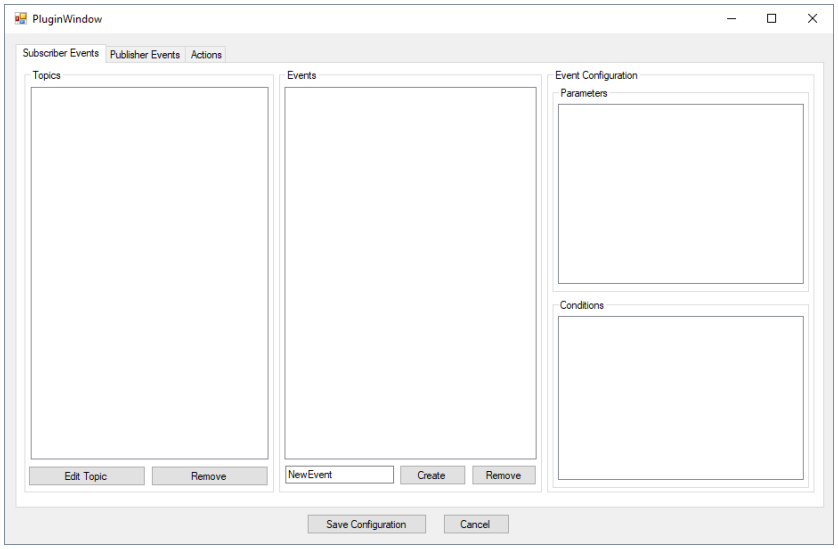
5.1 Plugin Window

Hoewel Hasselt reeds ondersteuning bood voor plugins, werd er geen manier voorzien om deze te configureren. Plugins waren statische dlls die meestal opnieuw gecompileerd moesten worden wanneer aanpassingen gemaakt moesten worden (soms werd gebruikt gemaakt van tekst-bestanden om specifieke data voor een plugin bij te houden). Bij het opstarten van Hasselt valt het op dat de menubalk een nieuw item bevat: *Plugins* (zie figuur 5.1). Dit menu somt alle ingeladen plugins op die een implementatie van een IPluginWindow interface bevatten. Dit laat plugins toe om eenvoudig configuratiemenu's aan de gebruiker bloot te stellen. Hoewel het implementeren van een extra interface extra werk vereist, kan deze, zoals ondervonden tijdens deze masterproef, een groot voordeel voorzien bij het ontwikkelen en testen van multimodale interacties.



Figuur 5.1: Het *Plugin* menu weergeeft alle ingeladen plugins die een configuratiescherm voorzien.

Een overzicht van het configuratiescherm voor de ROS/actionlib tool ziet u in figuur 5.2. In de rest van dit hoofdstuk zal de functionaliteit van de tool verder worden toegelicht.



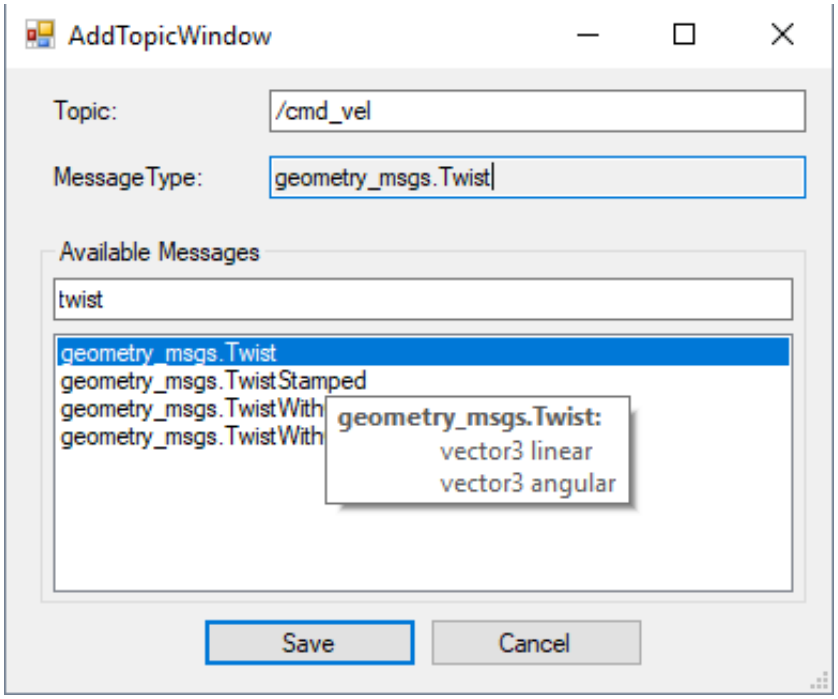
Figuur 5.2: Het configuratiescherm voor de tool bevat 3 tabbladen, respectievelijk om events te genereren gebaseerd aan de hand van inkomende ROS berichten, ROS berichten te genereren aan de hand van een event en action clients te configureren.

5.2 ROS

De eerste twee tabbladen van het configuratiescherm zijn voorzien om ROS berichten om te zetten in Hasselt events en omgekeerd. In deze sectie zullen alle opties van deze twee tabbladen overlopen worden.

5.2.1 Subscriber Events

Een subscriber event is een event dat gegenereerd wordt wanneer een bepaald ROS bericht toekomt. Zoals beschreven in sectie 2.4.3, worden ROS berichten naar een bepaald topic gepubliceerd. In de linker kolom worden alle geconfigureerde topics weergegeven. Om de tool te laten luisteren op een nieuw topic, moet men dubbelklikken in de "Topics"kolom. Een nieuw venster opent zich om een topic te configureren. Een ingevulde versie van dit venster wordt weergegeven in figuur 5.3.



Figuur 5.3: Een ingevulde versie van het topic-venster. Er werd een zoekbalk geïmplementeerd om het zoeken van een berichttype te vergemakkelijken

Een topic bestaat uit een *string* die het topic identificeert en een berichttype die aangeeft welk soort bericht kan worden verwacht. De naam van een ROS topic begint altijd met een /. Deze wordt ook in dit menu geforceerd zodat onverwachte fouten vermeden worden.

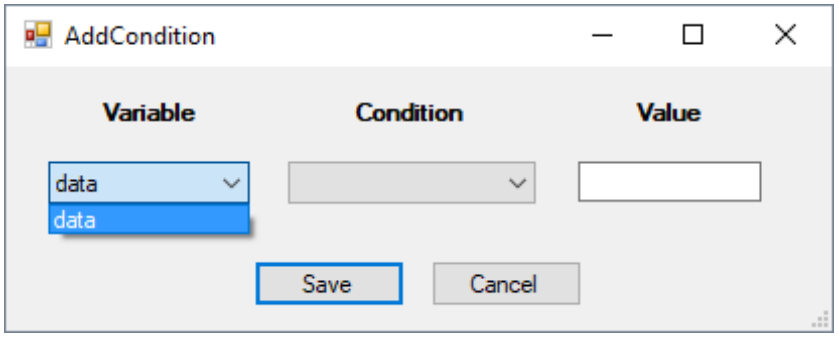
De lijst van standaard berichttypes omvat reeds honderdtal items. Buiten de standaard types kunnen gebruikers zelf *custom* berichttypes genereren. Om het zoeken van een berichttype te vergemakkelijken werd een zoekbalk toegevoegd. Het menu toon daarboven meer informatie over een bepaald type wanneer erover wordt *gehovered*. Deze twee eenvoudige toevoegingen zorgen ervoor dat de tool sneller kan worden geconfigureerd. Wanneer men dubbelklikt op een van de berichttypes wordt deze aan de topic toegekend. Wanneer vervolgens op "save"wordt geklikt zal het topic worden toegevoegd aan de configuratie van de tool. Het topic is nu zichtbaar in het plugin venster (zie figuur 5.2). Er kunnen nu events aan worden toegevoegd.

Een subscriber event beschrijft een event dat binnen de omgeving van Hasselt gegenereerd wordt wanneer aan bepaalde condities wordt voldaan. Wanneer

een event wordt afgevuurd, kan deze tevens informatie doorspelen aan Hasselt. Hierdoor kunnen de ROS events op een flexibele manier gebruikt worden om allerhande situaties af te handelen. Om een nieuw event toe te voegen, kan men dubbelklikken in de lijst van events. Men kan vervolgens een naam kiezen voor het event. Een event kan ook aangemaakt worden door een naam in te vullen in het veld onder de lijst van events en vervolgens op *Create* te klikken (figuur 5.2).

Per subscriber event kan aangeduid worden welke parameters moeten worden overgedragen naar Hasselt. Deze parameters kunnen vervolgens gebruikt worden om andere events af te vuren of bepaalde acties uit te voeren. Om te controleren wanneer een bepaald event moet worden afgevuurd kunnen condities worden toegevoegd. Hoewel Hasselt reeds functionaliteit biedt om events af te vuren onder bepaalde omstandigheden, kunnen deze condities helpen om de Hasselt code overzichtelijker te houden. Ook zorgt dit ervoor dat conditionele events kunnen worden opgeslagen in de configuratie om zo makkelijker hergebruikt te kunnen worden.

Om een conditie toe te voegen kan men dubbelklikken op de (lege) lijst van condities, waarna een speciaal menu opent, zoals wordt weergegeven in figuur 5.2.1. De gebruiker kan in dit menu meerdere condities invoeren op de parameters van het event. Slechts als alle condities slagen, zal het event worden afgevuurd in Hasselt.



Figuur 5.4: Het menu waarin de gebruiker condities kan toevoegen voor een event. In het linkse veld kan de parameter gekozen worden waarvoor een conditie wordt ingesteld. Het middelste veld geeft aan welke operator zal worden toegepast. De beschikbare operators hangen af van het type van de parameter, Ten slotte bevat het laatste veld de waarde waar de operator op wordt toegepast. Het is mogelijk om voor eenzelfde parameter meerder condities te definiëren.

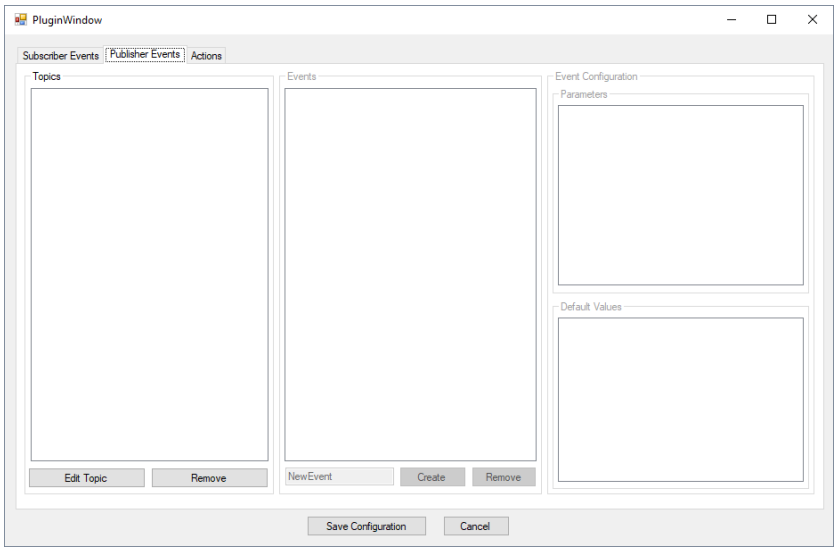
Omdat deze condities een nice-to-have zijn, en dus niet essentieel voor de werking van de tool, werden slechts een klein aantal condities geïmplementeerd. Zo kunnen voor strings de operators {EQUALS, CONTAINS, STARTS_WITH, ENDS_WITH} worden gebruikt. Voor getallen kunnen de operators {EQUALS, GREATER_THAN, LESS_THAN, GREATER_THAN_EQUALS, LESS_THAN_EQUALS } worden gebruikt. Ten slotte worden voor booleaanse waarden de operators {TRUE, FALSE} gedefiniëerd.

5.2.2 Publisher Events

Publisher events kunnen binnen Hasselt afgevuurd worden door een state-verandering. Een publisher event zal een ROS bericht op een bepaald topic versturen. Om te definiëren welk ROS bericht verstuurd moet worden op welk topic, werd hiervoor een interface voorzien.

De interface voor de publisher events gebruikt een interface die gelijkaardig is aan die van de subscriber events: In de linkerkolom kunnen topics aangemaakt worden, terwijl de kolom met Events alle events voor een topic weergeeft. Ten slotte kan per publisher event ook de parameters gekozen worden. Deze parameters definiëren de functie-definitie van het event in Hasselt. In plaats van condities bevat dit venster een lijst met *default* waarden voor de aangeduide parameters. Zoals beschreven in 4, kan dit gebruikt worden om events aan te maken die voor bepaalde parameters altijd eenzelfde waarde

doorsturen, tenzij deze wordt overschreven.

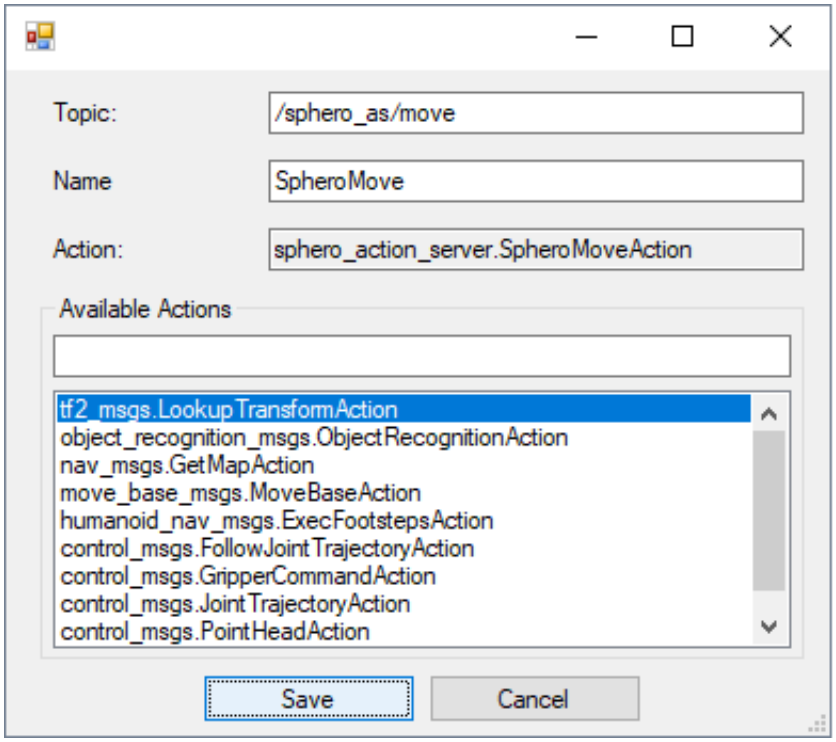


Figuur 5.5: Het menu voor de publisher events. In plaats van condities kunnen in dit venster standaard waarden aan de parameters worden meegegeven. Hierdoor kan de *signature* voor een event verkleint worden indien bepaalde parameters vaak eenzelfde waarde hebben.

Bij het aanmaken van zowel subscriber als publisher events, moet rekening worden gehouden met het kiezen van parameters voor een event. Wanneer verschillende event *signature* worden gedefiniëerd door de gebruiker, moet hij zelf nagaan dat twee *signatures* niet identiek zijn. Indien dit toch het geval is, zal bij het triggeren van een event de eerst-gevonden *signature* gekozen worden, wat kan leiden tot onverwachte resultaten.

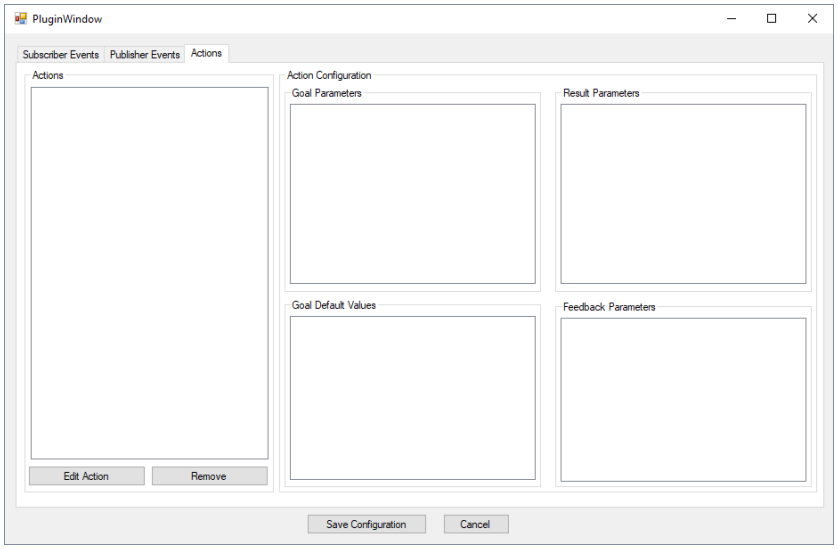
5.3 Actionlib

De laatste tab van het configuratiemenu wordt gebruikt om actions te definiëren. Om te beginnen moet een action worden aangemaakt door in de linkerkolom te dubbelklikken. Een menu verschijnt dat lijkt op die van de ROS events, zoals wordt weergegeven in figuur 5.6. De type berichten die gekozen kunnen worden bevatten echter enkel ROS berichten die overeenkomen met een actionlib *Goal* bericht (zie sectie 2.4.3).



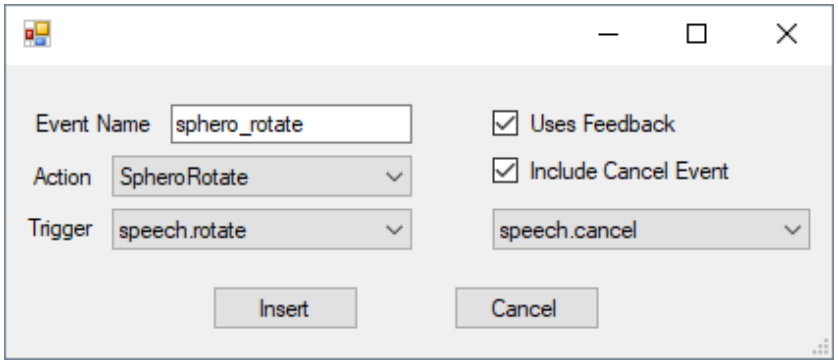
Figuur 5.6: Het menu voor het toevoegen van actie. Een naam moet worden meegegeven zodat de actie in Hasselt kan worden aangeroepen.

Bijkomend moet een naam worden gegeven aan de actie om deze later in Hasselt te kunnen oproepen. Wanneer de action is aangemaakt, kan gekozen worden welke informatie moet worden meegegeven, alsook welke informatie wordt teruggegeven door de action feedback en result berichten. De manier waarop dit gebeurt is hetzelfde als bij de hierboven besproken ROS events. Figuur 5.7 geeft een overzicht van het configuratiemenu voor actions.



Figuur 5.7: Het menu voor de actions.

Zoals beschreven in 4.3, werd gekozen om de action client te implementeren in Hasselt zelf, met behulp van SRDL. Wanneer een actie volledig geconfigureerd is, ligt de verantwoordelijkheid bij de gebruiker om de precieze flow van de client op te stellen. Door de implementatie van de action client in SRDL te verwezenlijken, wordt de flexibiliteit verhoogd. Zo kan een gebruiker afhankelijk van de feedback of resultaten van de action server een event afvuren, de dialoog resetten of zelfs een nieuw doel starten. In sectie 6 wordt een simpele action client gemaakt voor het bewegen van een Sphero robot. Om het voor de gebruiker eenvoudiger te maken, werd Hasselt van een contextmenu voorzien waarmee een action client kan worden aangemaakt. Door het rechtsklikken in de CEDL editor, kan de optie *insert action* worden aangeklikt. Dit opent een menu waarin de action client kan worden geconfigureerd. Een voorbeeld van dit menu wordt weergegeven in figuur 5.8



Figuur 5.8: Het menu voor het toevoegen van de sphero_rotate action client. Er kan een trigger gekozen worden, alsook of de action client een feedback loop heeft en of er automatisch een cancel event moet worden aangemaakt. Dit zorgt ervoor dat de gebruiker zelf weinig kennis moet hebben van hoe een action client precies werkt.

5.4 Genereren van berichttypes

Nieuwe berichttypen kunnen aan de tool worden toegevoegd aan de hand van code binnen ROS.NET. Dit is echter niet erg triviaal, en vereist het uitvoeren van gespecialiseerde code in de rootfolder van de geïmplementeerd tool. Tijdens de implementatie van deze thesis werd hiervoor een oplossing gevonden door de geleverde ROS.NET code aan te passen zodat deze als standalone applicatie kon worden uitgevoerd. Hierdoor kan deze opgeroepen worden binnen de ROS en Actionlib plugins om berichttypen at-runtime toe te voegen.

5.5 Conclusie

Deze sectie geeft een overzicht van de functionaliteit die werd ontwikkeld. Aan de hand van de besproken topics kan een robot in Hasselt worden geïntegreerd. In de volgende sectie wordt een use-case rond de Sphero robot uitgewerkt aan de hand van de functionaliteit besproken in deze sectie.

6 Sphero use-case

Tijdens de ontwikkeling van deze thesis werd gebruik gemaakt van een Sphero robot om de functionaliteit te testen. Er werd tevens een demo gemaakt met deze robot, waarbij de functionaliteit en het gebruik van de ontwikkelde tool worden gedemonstreerd. In deze sectie wordt een overzicht van de implementatie van deze use-case gegeven.

6.1 Overzicht

Sphero is een simpele balvormige robot die vooral voor entertainende doeleinden gebruikt wordt. De bal beweegt zich door zijn omgeving door te rollen. Bovendien bevat hij een LED die ingesteld kan worden. Latere modellen van de robot kunnen tevens geluiden afspelen.

De use-case die in deze sectie wordt toegelicht, zal de Sphero robot een aantal acties laten uitvoeren. Eerst zal de ROS implementatie gebruikt worden om Sphero rechtstreeks te besturen via een toetsenbord. Hierbij zal ook iets voorzien worden om Sphero van kleur te doen veranderen zolang een knop is ingedrukt. Hierna gaan wordt de code uitgebreid om Sphero te besturen via spraak-commando's.

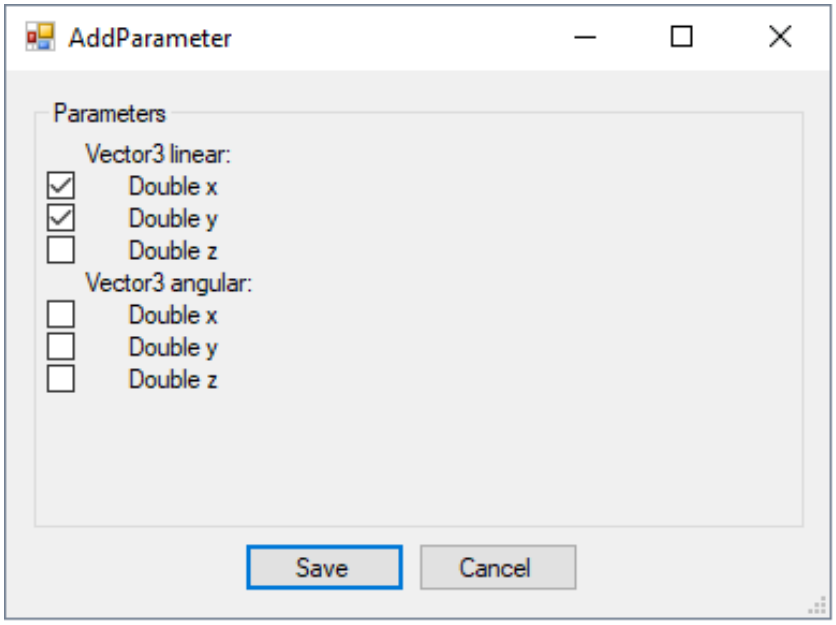
De driver voor de Sphero robot werd geïmplementeerd in Python, en bevat ROS connectiviteit en een simpele action server. De implementatie van deze driver zal niet in deze paper worden besproken. Videos waarin de implementatie van de ROS en actionlib use-case worden getoond, kunnen worden teruggevonden op respectievelijk `https://vimeo.com/269838575` en `https://vimeo.com/269837629`.

6.2 ROS Implementatie

Om de ROS events die de Sphero driver ondersteunt aan te spreken, openen we het *Publisher Events* tab in het configuratiemenu. Vervolgens dubbelklikken we op de topics lijst om twee nieuwe topics toe te voegen: */cmd_vel* om

Sphero te doen rollen, en */set_heading* om Sphero te roteren. Het *cmd_vel topic* accepteert berichten van type *geometry_msgs.Twist*. Dit geven we aan in het topic door te dubbelklikken op het juiste berichttype, zoals wordt weergegeven in figuur 5.3.

Vervolgens worden twee nieuwe events aangemaakt, genaamd Move en Rotate. Het berichttype voor Move bevat twee *Vector3* Subtypes. een voor lineaire beweging, en een voor angulaire beweging. Aangezien het Move event enkel een lineaire beweging met zich meebrengt, worden enkel de lineaire X, en Y variabelen aangevinkt als parameters voor het event. Bij het aanmaken van een parameter binding worden alle beschikbare parameters van het Twist event opgesomd, zoals wordt weergegeven in figuur 6.1.



Figuur 6.1: een overzicht van de parameters voor het Twist event

Nu het Move event is aangemaakt, kan deze worden gebruikt in Hasselt door middel van de syntax $Move < X, Y >$. Allereerst moet een *composite* event worden gedefiniëerd waarmee de input van de gebruiker kan worden gedetecteerd. Hieronder wordt de CEDL code voor dit event weergegeven:

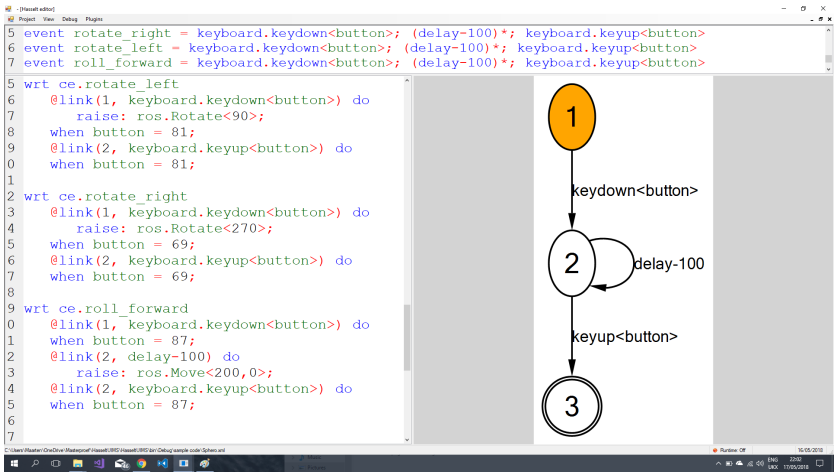
```
18 event roll_forward = keyboard.keydown<button>; (delay-100)*; keyboard.keyup<button>
```

Dit definiëert een event dat, zolang een knop wordt ingehouden, in eenzelfde state blijft. het Move event kan nu worden toegekend aan een node of link binnen deze *composite* event. Figuur 6.2 weergeeft de code die ervoor zorgt dat het Move event elke 100ms wordt afgevuurd zolang de "W"toets wordt ingehouden:

```
91  wrt ce.roll_forward
92      @link(1, keyboard.keydown<button>) do
93      when button = 87;
94      @link(2, delay-100) do
95          raise: ros.Move<200,0>;
96      @link(2, keyboard.keyup<button>) do
97      when button = 87;
98
```

Figuur 6.2: Zolang de "W"toets ingehouden wordt, zal de FSM die het event beschrijft via link 2 in node 2 blijven. Op link 2 wordt het Move event gedefiniëerd, zodat elke 100ms het ROS Move event naar Sphero wordt gestuurd. Wanneer de "W"knop wordt losgelaten, zal de FSM naar zijn eindstaat gaan en zich resetten naar Node 1.

We herhalen de hierboven genoemde stappen om ook rotatie events toe te voegen. De volledige code wordt weergegeven in figuur 6.3. We merken op dat voor deze simpele implementatie gebruik gemaakt werd van slechts 1 event voor het roteren van Sphero, waarbij de rotatierichitng wordt aangeduid in SRDL. Er kon echter geopteerd worden om twee events aan te maken: *RotateLeft* en *RotateRight*, waarbij de parameter een standaard waarde krijgt van respectievelijk 90 en 270 graden.
Alle configuratie nodig om Sphero te doen reageren op het indrukken van het toets is voltooid. Het volstaat nu om de computer waar Hasselt op draait te verbinden met het netwerk van de ROS master node en Sphero. Wanneer het Hasselt project uitgevoerd word, kan Sphero bediend worden via het toetsenbord.



Figuur 6.3: De CEDL code, SRDL code en FSM nodig om een Sphero robot via het toestenbord te besturen. De keyboard inputs kunnen vervangen worden door bijvooorbeeld spraakcommando's, zoals in sectie 6.3 wordt weergegeven.

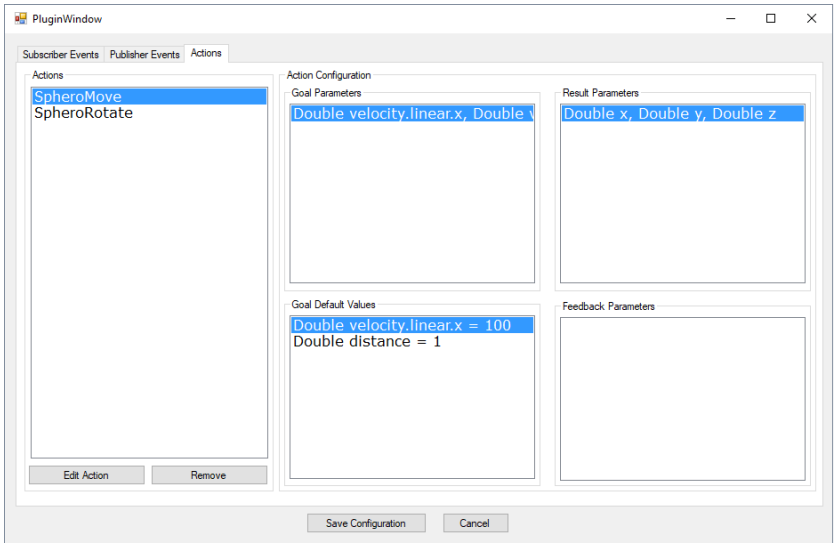
6.3 Actionlib Implementatie

De geïmplenteerde Sphero driver bevat tevens een simpele action server. Deze ondersteunt twee acties: Sphero in een bepaalde richting bewegen over een bepaalde afstand, en het roteren van Sphero tot de actie onderbroken wordt. In deze sectie wordt weergegeven hoe deze acties aangesproken kunnen worden via de ontwikkelde plugin.

Net zoals bij ROS, moeten eerst de ROS topics gedefiniëerd worden waarop de acties worden aangesproken. In tegenstelling tot ROS, kan een topic slechts gebruikt worden voor 1 actie per robot. Om die reden bevat het menu een veld om de naam van de actie te omschrijven. Daarboven wordt de lijst van beschikbare events automatisch gefilterd op *Action* typen, zodat gebruiksfouten worden uitgesloten. Wanneer een naam gekozen wordt, zal de plugin automatisch 3 Hasselt events genereren. Om de Sphero robot te kunnen bewegen en roteren, worden 2 events *SpheroMove* en *SpheroRotate* aangemaakt. Een ingevuld menu voor het SpheroMove event wordt weergegeven in figuur 5.6.

Wanneer de acties zijn aangemaakt, zijn een aantal verdere configuraties mogelijk. Net zoals bij ROS, kunnen de parameters van de actie doel wor-

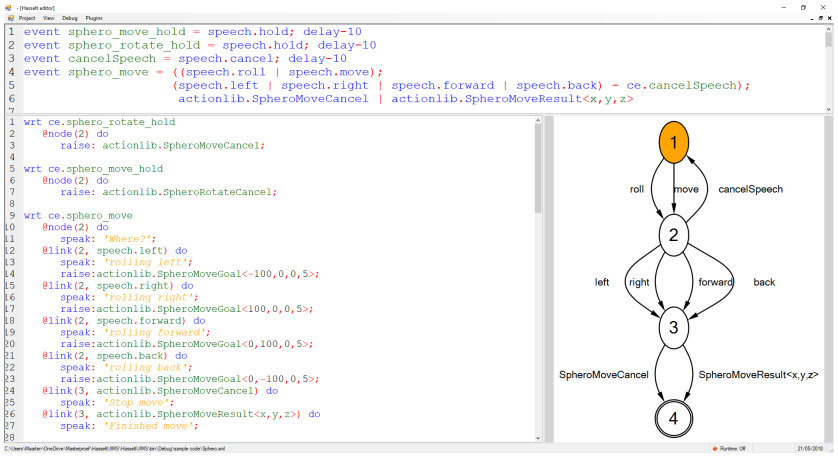
den aangegeven. Hieraan kunnen ook *default* waarden worden meegegeven. Daarboven kunnen de parameters van het *Feedback*- en het *Result*-bericht worden aangegeven. Een voorbeeld configuratie voor de SpheroMove actie wordt weergegeven in figuur 6.4. Er wordt ook een SpheroRotate aangemaakt op dezelfde manier.



Figuur 6.4: De parameters voor het doel bevatten de x, y en z beweging, alsook de afstand die Sphero af zal leggen alvorens vanzelf te stoppen. Als standaard waarden wordt een afstand van 1 ingesteld.

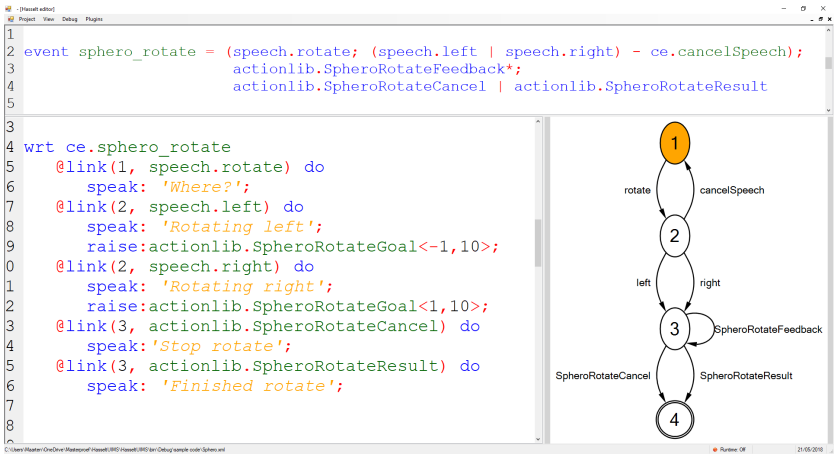
De volgende stap is het implementeren van de action clients in Hasselt zelf. Dit gebeurd aan de hand van het contextmenu dat speciaal werd toegevoegd (zie sectie 5.3). Hierdoor worden *composite events* aangemaakt die dezelfde flow als een simpele action client implementeren. Het event voor de SpheroMove action client wordt weergegeven in figuur 6.5. Dit event zal eerst luisteren naar een *Roll* of *Move* spraakcommando. Vervolgens kan een richting worden aangegeven aan de hand van *Left, Right, Forward en Back*. Eventueel kan de gebruiker het event stopzetten door *Cancel* te zeggen. Wanneer echter een richting wordt gekozen, zal het gekozen *Goal* bericht worden opgeroepen met de gewenste parameters. Vervolgens kan de actie op elk moment stopgezet worden door het spraakcommando *Stop* uit te voeren, waardoor het SpheroMoveCancel event wordt opgeroepen en de FSM zijn einde bereikt. Een andere mogelijkheid is dat de Sphero de gewenste afstand heeft afgelegd, waardoor een SpheroMoveResult bericht binnenkomt met informa-

tie over de huidige positie van Sphero. Dit zal tevens de FSM eindigen, waardoor Sphero zal stoppen.



Figuur 6.5: De SRDL code en FSM voor de SpheroMove actie. Zoals wordt weergegeven, zal afhankelijk van de uitgesproken richting, het SpheroMove-Goal event worden opgeroepen met een bepaalde richting. Op het moment wordt de afstand die Sphero aflegt meegegeven aan het SpheroMove event. Dit zou tevens kunnen worden opgenomen in de standaard waarden van het doel. Het is echter handig bij het testen om deze eenvoudig in SRDL editor aan te kunnen passen.

Het feedback-aspect van een action server werd niet opgenomen in de SpheroMove actie. Deze werd echter wel opgenomen in de SpheroRotate actie. Deze wordt weergegeven in figuur 6.6. Het toevoegen van een feedback loop kunnen andere Hasselt events worden opgeroepen als reactie op feedback van de SpheroRotate actie. Zo kan de rotatie automatisch worden stopgezet wanneer een bepaalde kijkrichting werd bereikt.



Figuur 6.6: De SRDL code en FSM voor de SpheroRotate actie. In deze action client wordt een feedback loop geïmplementeerd.

6.4 Conclusie

In deze sectie werd een use-case uitgewerkt met de Sphero robot. Aan de hand van deze use-case kan gekeken worden of doelstellingen opgesteld in sectie 3 gehaald werden. Hieronder wordt weergegeven of bepaalde doelstellingen al dan niet werden gehaald.

(1), (5), (6) De interface voor een robot kan eenvoudig geconfigureerd worden via de interface van de tool. Hierbij kunnen zowel inkomende als uitgaande berichten worden aangemaakt. Ook kunnen actions geconfigureerd worden. Er werden tevens bijkomende constraints toegevoegd, waardoor een interactie nog specifieker kan worden gedefiniëerd. Hierbij worden zowel ROS als actionlib ondersteund door de interface.

(2) Het aanpassen van de configuratie gebeurt net zoals het aanmaken via de interface van de tool. Deze werd volledig dynamisch gemaakt, waardoor geen programmatie en dus ook geen compilatie vereist is. De interface voor de robot kan worden aangepast zonder Hasselt opnieuw op te starten.

(3) De configuratie van de ROS en actionlib interfaces worden weggeschreven naar een XML bestand. Er is voorlopig echter nog geen manier om aan te geven welke configuratie gebruikt moet worden in een Hasselt project. Deze doelstelling word dus slechts gedeeltelijk ondersteund.

(4), (7) Action clients kunnen worden toegevoegd aan Hasselt door middel van een context menu. Dit menu zorgt ervoor dat de gebruiker geen kennis moet hebben van de structuur van een action client. Bovendien zorgt dit ervoor dat langdurige acties kunnen worden uitgevoerd door robots, en dat Hasselt events kunnen worden afgevuurd aan de hand van de vooruitgang van de actie.

7 Conclusie

Het doel van deze thesis was het vereenvoudigen van het prototypen van mens-robot interactie. De inherente multi-modaliteit van menselijke interactie is hierbij een belangrijk aspect. Hierdoor werd gekozen om een robot software framework te integreren met een reeds bestaand framework voor multimodale interactie. Om een multimodaal framework te kiezen werd een vergelijking gemaakt aan de hand van hun *scope*, zoals gedefiniëerd door Cuenca e.a. Door deze vergelijking te maken werd gekozen om voor een state-based framework te opteren. Deze frameworks bieden een goede set van functionaliteit en voldoen aan de doelstelling om een gebruiksvriendelijke tool te maken. Het gekozen framework is Hasselt, waarvan de source code beschikbaar is. Dit boodt een maximale integratie. Als robot software framework zelf werd gekozen voor het Robot Operating System (ROS). ROS is een *message-passing* framework, waarbij berichten van en naar robots worden verstuurd via een master node. Dit zorgt ervoor dat koppeling tussen de robots laag ligt, maar ook dat het implementeren en uitbreiden van een ROS netwerk eenvoudig is.

Er werd vervolgens een plugin geschreven voor Hasselt die toeliet een Sphero robot te controleren aan de hand van ROS berichten. Deze berichten worden verzonden als een Hasselt event wordt afgevuurd. Ook het omgekeerde is waar: Bepaalde ROS berichten kunnen Hasselt events afvuren, die vervolgens gebruikt kunnen worden in een meer complexe *composite* event. Robots verrichten echter vaak taken, en ROS op zich is hier niet erg geschikt voor: Het terugkrijgen van feedback van een taak wordt niet ondersteund. Om die reden werd gekozen om de plugin uit te breiden om ook actionlib te ondersteunen. actionlib is een onderdeel van ROS dat toelaat om acties uit te voeren, en gedurende de uitvoering van de actie feedback te krijgen over zijn status. Aan de hand van Sphero werd een gelijkaardig scenario opgesteld, maar dan met actionlib. Hierbij beweegt of draait de Sphero een bepaalde afstand, of tot de beweging wordt stopgezet.

Het integreren van ROS en actionlib met een multimodaal framework heeft als resultaat dat een multimodale interactie met een robot op een eenvoudige manier kan worden geïmplementeerd. Niet enkel de implementatie, maar ook het testen en aanpassen ervan wordt vereenvoudigd. Daarboven kunnen mo-

daliteiten eenvoudig worden toegevoegd aan een interactie door middel van het framework zelf. De scenario's met Sphero die beschreven werden maakten gebruik van het toetsenbord en spraakcommando's. Het toevoegen van een extra modaliteit, zoals bijvoorbeeld *gestures*, is nu slechts kwestie van een *Gesture* plugin toe te voegen en deze inputs te verbinden met de ROS en actionlib plugin.

De configuratiebestanden voor de geïmplementeerde plugin worden tevens als XML bestanden opgeslagen. Hierdoor wordt het delen van configuratie voor specifieke robots eenvoudig. Dit kan de ontwikkeltijd van multimodale systemen nogmaals vereenvoudigen. Ook is deze feature handig voor ontwikkelaars die meerdere computers hebben: Hasselt en de plugins kunnen op een cloud storage platform geplaatst worden om zo overal bruikbaar te zijn.

8 Future Work

Naar de toekomst toe zijn er een aantal verbeteringen die geïmplementeerd kunnen worden om de gebruiksvriendelijkheid van de tool te verbeteren. Deze worden hieronder opgesomd:
Genereren van bericht typen: Tijdens de ontwikkeling werd de bestaande code van ROS.NET uitgebreid zodat berichttypes eenvoudig toegevoegd konden worden door het uitvoeren van een programma. Deze scande automatisch zijn root folder om zo alle definities van berichttypes op te sporen en deze te compileren zodat ze bruikbaar waren in de plugin. Deze manier van werken was voldoende, aangezien het toevoegen van berichttypes voor de implementatie slechts een aantal keer moest gebeuren. In een echte onderzoeksomgeving zal dit echter vaker voorvallen. Hierdoor kan het handig zijn om deze functionaliteit te voorzien in de plugin, zodat geen externe programma's uitgevoerd dienen te worden.

Condities: De huidige implementatie voor de condities van de events is redelijk mininaal. Hierdoor is de gebruiker beperkt tot slechts simpele condities, zonder bijvoorbeeld *OR* operaties. Een mogelijke verbetering is het uitbreiden van de plugin met een meta-taal om condities te definiëren.

Action Clients: Momenteel worden enkel simpele action clients ondersteund. Dit zijn action clients waarbij slechts 1 actie tegelijk kan draaien op de server. Dit zorgt ervoor dat acties sequentieel moeten worden uitgevoerd, in plaats van parallel. Een mogelijke verbetering is het onderzoeken van ondersteuning voor action clients die meer dan 1 taak kunnen afhandelen.

[1] Yamine Ait Ameur en Nadjet Kamel. „A generic formal specification of fusion of modalities in a multimodal HCI". In: *Building the Information Society*. Springer, 2004, p. 415–420.

[2] Ronald C Arkin. *Behavior-based robotics*. MIT press, 1998.

[3] Minoru Asada e.a. „Cognitive developmental robotics as a new paradigm for the design of humanoid robots". In: *Robotics and Autonomous Systems* 37.2 (2001), p. 185–193.

[4] Meera M Blattner en Ephraim P Glinert. „Multimodal integration". In: *IEEE multimedia* 3.4 (1996), p. 14–24.

[5] Richard A Bolt. *"Put-that-there": Voice and gesture at the graphics interface*. Deel 14. 3. ACM, 1980.

[6] Oliver Brock e.a. „A framework for learning and control in intelligent humanoid robots". In: *International Journal of Humanoid Robotics* 2.03 (2005), p. 301–336.

[7] Rodney Brooks. „A robust layered control system for a mobile robot". In: *IEEE journal on robotics and automation* 2.1 (1986), p. 14–23.

[8] Rodney Allen Brooks. *Cambrian intelligence: The early history of the new AI*. Deel 44. Mit Press Cambridge, MA, 1999.

[9] Harry Bunt, Robbert-Jan Beun en Tijn Borghuis. *Multimodal human-computer communication: systems, techniques, and experiments*. Deel 1374. Springer Science & Business Media, 1998.

[10] Philip R Cohen e.a. „Quickset: Multimodal interaction for distributed applications". In: *Proceedings of the fifth ACM international conference on Multimedia*. ACM. 1997, p. 31–40.

[11] Toby HJ Collett, Bruce A MacDonald en Brian P Gerkey. „Player 2.0: Toward a practical robot programming framework". In: *Proceedings of the Australasian conference on robotics and automation (ACRA 2005)*. Citeseer Citeseer. 2005, p. 145.

[12] Jacob W Crandall e.a. „Validating human-robot interaction schemes in multitasking environments". In: *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 35.4 (2005), p. 438–449.

[13] Fredy Cuenca e.a. „Graphical toolkits for rapid prototyping of multimodal systems: A survey". In: *Interacting with Computers* 27.4 (2014), p. 470–488.

[14] Fredy Cuenca e.a. „Hasselt UIMS: A Tool for Describing Multimodal Interactions with Composite Events". In: *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. EICS '15. Duisburg, Germany: ACM, 2015, p. 226–229. ISBN: 978-1-4503-3646-8. DOI: 10.1145/2774225.2775437. URL: http://doi.acm.org/10.1145/2774225.2775437.

[15] Joris De Schutter e.a. „Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty". In: *The International Journal of Robotics Research* 26.5 (2007), p. 433–455.

[16] Wilm Decré, Herman Bruyninckx en Joris De Schutter. „Extending the iTaSC constraint-based robot task specification framework to time-independent trajectories and user-configurable task horizons". In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, p. 1941–1948.

[17] Wilm Decré e.a. „Extending iTaSC to support inequality constraints and non-instantaneous task specification". In: *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. IEEE. 2009, p. 964–971.

[18] Bruno Dumas, Denis Lalanne en Rolf Ingold. „HephaisTK: a toolkit for rapid prototyping of multimodal interfaces". In: *Proceedings of the 2009 international conference on Multimodal interfaces*. ACM. 2009, p. 231–232.

[19] Jerry Alan Fails en Dan R Olsen Jr. „Interactive machine learning". In: *Proceedings of the 8th international conference on Intelligent user interfaces*. ACM. 2003, p. 39–45.

[20] Jodi Forlizzi en Carl DiSalvo. „Service robots in the domestic environment: a study of the roomba vacuum in the home". In: *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*. ACM. 2006, p. 258–265.

[21] Michael A. Goodrich en Alan C. Schultz. „Human-robot Interaction: A Survey". In: *Found. Trends Hum.-Comput. Interact.* 1.3 (jan 2007), p. 203–275. ISSN: 1551-3955. DOI: 10.1561/1100000005. URL: http://dx.doi.org/10.1561/1100000005.

[22] MA Hearst e.a. „Mixed-initiative interaction: Trends and controversies". In: *IEEE Intelligent Systems* 14.5 (1999), p. 14–23.

[23] Takayuki Kanda e.a. „Interactive robots as social partners and peer tutors for children: A field trial". In: *Human-computer interaction* 19.1 (2004), p. 61–84.

[24] David B Koons, Carlton J Sparrell en Kristinn Rr Thorisson. „Integrating simultaneous input from speech, gaze, and hand gestures". In: *MIT Press: Menlo Park, CA* (1993), p. 257–276.

[25] Christopher A Miller en Raja Parasuraman. „Beyond levels of automation: An architecture for more flexible human-automation collaboration". In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*. Deel 47. 1. SAGE Publications. 2003, p. 182–186.

[26] Robin Murphy. *Introduction to AI robotics*. MIT press, 2000.

[27] Robin R Murphy. „National Science Foundation summer field institute for rescue robots for research and response (R4)". In: *AI Magazine* 25.2 (2004), p. 133.

[28] Jeannette G Neal e.a. „Natural language with integrated deictic and graphic gestures". In: *Proceedings of the workshop on Speech and Natural Language*. Association for Computational Linguistics. 1989, p. 410–423.

[29] Monica N Nicolescu en Maja J Mataric. „Learning and interacting in human-robot domains". In: *IEEE Transactions on Systems, man, and Cybernetics-part A: Systems and Humans* 31.5 (2001), p. 419–430.

[30] Nils J Nilsson. *Shakey the robot*. Tech. rap. DTIC Document, 1984.

[31] Illah R Nourbakhsh, Clayton Kunz en Thomas Willeke. „The mobot museum robot installations: A five year experiment". In: *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*. Deel 4. IEEE. 2003, p. 3636–3641.

[32] Sharon Oviatt. „Ten myths of multimodal interaction". In: *Communications of the ACM* 42.11 (1999), p. 74–81.

[33] Sharon Oviatt, Rebecca Lunsford en Rachel Coulston. „Individual differences in multimodal integration patterns: What are they and why do they exist?" In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 2005, p. 241–249.

[34] Sharon Oviatt e.a. „Designing the user interface for multimodal speech and pen-based gesture applications: state-of-the-art systems and future research directions". In: *Human-computer interaction* 15.4 (2000), p. 263–322.

[35] Leah M Reeves e.a. „Guidelines for multimodal user interface design". In: *Communications of the ACM* 47.1 (2004), p. 57–59.

[36] Deb K Roy. „Learning visually grounded words and syntax for a scene description task". In: *Computer speech & language* 16.3 (2002), p. 353–385.

[37] Joe Saunders, Chrystopher L Nehaniv en Kerstin Dautenhahn. „Teaching robots by moulding behavior and scaffolding the environment". In: *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*. ACM. 2006, p. 118–125.

[38] Marcos Serrano e.a. „The Openinterface Framework: A Tool for Multimodal Interaction." In: *CHI '08 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '08. Florence, Italy: ACM, 2008, p. 3501–3506. ISBN: 978-1-60558-012-8. DOI: 10.1145/1358628.1358881. URL: http://doi.acm.org/10.1145/1358628.1358881.

[39] Thomas B Sheridan en William L Verplank. *Human and computer control of undersea teleoperators*. Tech. rap. DTIC Document, 1978.

[40] Takanori Shibata en Kazuo Tanie. „Influence of a priori knowledge in subjective interpretation and evaluation by short-term interaction with mental commit robot". In: *Intelligent Robots and Systems, 2000.(IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*. Deel 1. IEEE. 2000, p. 169–174.

[41] Candace L Sidner e.a. „Where to look: a study of human-robot engagement". In: *Proceedings of the 9th international conference on Intelligent user interfaces*. ACM. 2004, p. 78–84.

[42] Ousmane Sy e.a. „Petshop: a case tool for the petri net based specification and prototyping of corba systems". In: *Petri Nets 2000* (2000), p. 78.

[43] Sebastian Thrun e.a. „MINERVA: A second-generation museum tour-guide robot". In: *Robotics and automation, 1999. Proceedings. 1999 IEEE international conference on*. Deel 3. IEEE. 1999.

[44] G. Trafton. „Experimental Design in HRI". In: *Tutorial at HRI2007* (2007). Available from: http://hri2007.org/Tutorials.htm.

[45] Jan Van den Bergh e.a. „Toward specifying Human-Robot Collaboration with composite events". In: *Robot and Human Interactive Communication (RO-MAN), 2016 25th IEEE International Symposium on*. IEEE. 2016, p. 896–901.

[46] Benfang Xiao, Cynthia Girand en Sharon L Oviatt. „Multimodal integration patterns in children." In: *INTERSPEECH*. 2002.

[47] Benfang Xiao e.a. „Modeling multimodal integration patterns and performance in seniors: toward adaptive processing of individual differences". In: *Proceedings of the 5th international conference on multimodal interfaces*. ACM. 2003, p. 265–272.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Tool support for programming human-robot interaction

Richting: **master in de informatica-multimedia**
Jaar: **2018**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Martens, Maarten

Datum: **15/06/2018**