UNA ARQUITECTURA BASADA EN MULTI-AGENTES PARA LA PLANIFICACIÓN DEL AGARRE ESTABLE EN UNA MANO ROBÓTICA

ANEXOS

EDUAR DORADO DAZA
EDWAR DARIO AÑASCO PARRA

Director:
Ing. Jaime Díaz Chávez

UNIVERSIDAD DEL CAUCA
FACULTAD DE INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES
INGENIERÍA EN AUTOMÁTICA INDUSTRIAL
POPAYÁN
2010

TABLA DE CONTENIDO

ANEXO A: METODOLOGÍAS PARA EL DESARROLLO DE SISTEMAS MULTI-AGENTE *1*

A.1 *VOWEL ENGINEERING* **1**

A.2 *MaSE* **2**

A.3 *GAIA* **3**

A.4 *MESSAGE* **4**

A.5 *MAS-CommonKADS* **5**

ANEXO B: DISEÑO DE UNA ARQUITECTURA BASADA EN MULTI-AGENTES PARA LA PLANIFICACIÓN DE AGARRE ESTABLE BAJO LA METODOLOGÍA PROMETHEUS *6*

B.1 ESPECIFICACIÓN DEL SISTEMA **6**

B.1.1 Diagrama De Objetivos 6

B.1.2 Funcionalidades 7

B.1.3 Descriptores De Las Funcionalidades 7

B.1.4 Especificación De Los Escenarios 11

B.1.5 Percepciones, Acciones Y Datos 15

B.2 DISEÑO DE LA ARQUITECTURA **19**

B.2.1 Diagrama De Visión General Del Sistema 19

B.2.2 Descriptores De Agentes 19

B.2.3 Desarrollo De Los Descriptores De Los Protocolos 23

B.2.4 Desarrollo De Los Descriptores De Los Mensajes Entre Agentes 30

B.3 DISEÑO DETALLADO **33**

B.3.1 Agente Indice 34

B.3.2 Agente Pulgar 48

B.3.3 Agente Medio 53

B.3.4 Agente Anular 56

ANEXO C: SIMULADOR GraspIt! *60*

C.1 DESCRIPCIÓN GENERAL DEL SOFTWARE **60**

C.2 MANUAL DE INSTALACIÓN DE *GraspIt!* **61**

C.2.1 Configuración De Variables De Entorno En Windows 61

C.2.2 Instalacion De Qt 62

C.2.3 Instalacion De Coin 64

C.2.4 Instalacion De SoQT 64

C.2.5 Instalacion De Lapack 65

C.2.6 Instalación De *GraspIt!* 65

C.3 MANUAL DE USO DE *GraspIt!* 66

C.3.1 La Ventana Principal Y Los Controles. 67
C.3.2 El Modo Cámara 67
C.3.3 El Modo Interacción 68
C.3.4 Ejemplo De Agarre 69
C.3.5 Un Ejemplo De Dinámica 70
C.3.6 Los Menus 70
C.3.7 Tipos De Datos Principales Y Archivos De Datos 72
C.3.8 Detección De Colision Y Contacto 82
C.3.9 Implementacion Del Software 88
C.3.10 Medidas De Calidad Del Agarre 88
C.3.11 Dos Medidas De Calidad Bajo El Espacio *Wrench* Del Agarre 89
C.3.12 Conceptos Generales Y Tipos De Planificadores 90
C.3.13 Acoplamiento De Las Articulaciones Y Manos Sub-Actuadas 106
C.3.14 Interfaz Con Matlab 107

C.4 INSERCIÓN DE UNA MANO ROBÓTICA EN *GraspIt!* 109

C.5 CREAR UN AMBIENTE DE SIMULACIÓN CON UNA MANO Y UN OBJETO 116

C.6 GENERACIÓN DE MOVIMIENTO EN EL MODO ESTÁTICO Y DINÁMICO DESDE MATLAB 118

C.7 PRUEBAS EN *GraspIt!* PARA PROBAR LA EFICACIA DEL PARALELISMO ENTRE SUPERFICIES. 127

ANEXO D: APLICACIÓN DE FUERZAS EN EL AGARRE 131

LISTA DE FIGURAS

Figura A.1. Arquitectura de capas de Vowel Engineering 1
Figura A.2. Modelos en la metodología GAIA y sus relaciones en el proceso GAIA 3
Figura A.3. Meta-modelos que se especifican en la metodología MESSAGE 5
Figura B.2.1 Diagrama de visión general del sistema para la planificación de agarre en una mano robótica. 19
Figura B.2.3 Diagrama de interacción de acuerdo al escenario de configuración. 26
Figura B.2.4. Diagrama de interacción de acuerdo al escenario de adaptación por cierre. 26
Figura B.2.51. Diagrama de interacción de acuerdo con el escenario de adaptación por alineación. 27
Figura B.2.6. Diagrama de interacción de acuerdo con el escenario de contacto. 27
Figura B.2.7 Protocolo para la flexión. 28
Figura B.2.8 Protocolo para la aducción-abducción. 29
Figura B.3.1 Diagrama de visión general del agente índice. 34
Figura B.3.2. Diagrama de visión general de la capacidad de flexión_extensión_ind 36
Figura B.3.3. Diagrama de visión general de la capacidad de aducción_abduccion_ind 37
Figura B.3.4. Ilustración de algunas notaciones para la especificación de los procesos. 38
Figura B.3.5. Diagrama de proceso de configuración del agente índice. 39
Figura B.3.6 Diagrama de proceso para la flexión-extensión del agente índice. 40
Figura B.3.7. Diagrama de proceso para la aducción-abducción del agente índice 40
Figura B.3.8. Diagrama de visión general del agente pulgar 48
Figura B.3.9. Diagrama de visión general de la capacidad de flexión_extensión_pul 48
Figura B.3.10. Diagrama de visión general de la capacidad de oposicion_pul 49
Figura B.3.13 Diagrama de visión general del agente medio 53
Figura B.3.14 Diagrama de visión general de la capacidad de flexion_extension_med. 54
Figura B.3.15. Diagrama de visión general de la capacidad de aducción_abduccion_med. 54
Figura B.3.16. Diagrama del proceso de configuración del agente medio 55
Figura B.3.17. Proceso de flexion_extension del agente medio 55
Figura B.3.18. Proceso de aducción_abducción del agente medio 56
Figura B.3.19. Diagrama de visión general del agente anular. 56
Figura B.3.20. Diagrama de visión general de la capacidad de flexion_extension_anu. 57
Figura B.3.21. Diagrama de visión general de la capacidad de aducción_abduccion_anu. 57
Figura B.3.22. Diagrama del proceso de configuración del agente anular 58
Figura B.3.23. Proceso de flexion_extension del agente anular 58
Figura B.3.24. Proceso de aducción_abducción del agente anular. 59
Figura C.3.1 El mundo cargado desde el archivo dlr_*flask*.wld 66
Figura C.3.2. Tipos de volúmenes envolventes. 83

Figura C.3.3. Volúmenes envolventes generados por el barrido de una esfera. 83
Figura C.3.4. Jerarquía de Volúmenes Envolventes 84
Figura C.3.5. Ejemplo de un árbol de prueba de volumen envolvente (BVTT) 85
Figura C.3.6. Detección de colisión y proceso de detección del contacto 86
Figura C.3.7. Algunas de las posibles regiones de contacto 87
Figura C.3.8. Modelo de un vaso de café y su representación en primitivas 91
Figura C.3.10. Ubicación de los puntos de contacto deseados para las manos: DLR, Robonaut y la humana. 94
Figura C.4.1. Diferentes configuraciones de d0 114
Figura C.4.2. Falanges inferior y media del modelo de la mano del grupo con las respectivas configuraciones de las juntas. 114
Figura C.5.1. Robot *HumanHand* que viene en la distribución de *GraspIt!* 116
Figura C.5.2. Robot ***HumanHand*** traslapándose con el objeto ***flask*** que viene con la distribución de *GraspIt!* 117
Figura C.5.3. Posible posición del Matraz, con respecto a la mano, en el mundo de simulación 118
Figura C.6.1. Posición inicial de la mano 124
Figura C.6.2. Agarre de una copa de vidrio con la mano del grupo 125
Figura C.6.3. Reconstrucción de las fuerzas de contacto y posiciones para un cubo agarrado en *GraspIt!* 126
Figura C.7.2. Diferentes configuraciones de agarre de un vaso de café. 129
Figura C.7.3. Diferentes configuraciones de agarre de un icosaedro. 129
Figura D.1. Mano robotica agarrando un matraz 131
Figura D.2. Mano robotica agarrando una esfera. 132
Figura D.3. Mano robótica agarrando un libro 133
Figura D.4. Mano robótica agarrando una copa 134
Figura D.5 Mano robótica agarrando un icosaedro 135

LISTA DE TABLAS

Tabla 1. Escenario de configuración 11
Tabla 2. Escenario de adaptación por cierre 12
Tabla 3. Escenario de adaptación por alineación 13
Tabla 4. Escenario de contacto 14
Tabla 5. Agentes asociados a cada una de las funcionalidades según el escenario de configuración**.** 23
Tabla 6. Agentes asociados a cada una de las funcionalidades según el escenario de adaptación por cierre**.** 24
Tabla 7. Agentes asociados a cada una de las funcionalidades según el escenario de adaptación por alineación**.** 25
Tabla 8. Agentes asociados a cada una de las funcionalidades según el escenario de contacto. 25
Tabla 9. Capacidades de cada agente con sus respectivos objetivos 35
Tabla 10. Registro del primer expeimento. 132
Tabla 11. Registro del segundo expeimento. 133
Tabla 12. Registro del tercer expeimento. 134
Tabla 13. Registro del cuarto expeimento. 135
Tabla 14. Registro del quinto expeimento. 136

ANEXO A: METODOLOGÍAS PARA EL DESARROLLO DE SISTEMAS MULTI-AGENTE

A.1 *VOWEL ENGINEERING*

Se trata de la metodología seguida en el grupo *MAGMA*. El término vowel engineering viene de que el sistema final depende de la ordenación y agrupamiento de cuatro vocales A (por agentes), E (por entorno), I (por interacciones) y O (por organización). Los modelos de agente abarcan desde simples autómatas hasta complejos sistemas basados en conocimiento. La forma de ver las interacciones van desde modelos físicos (propagación de onda en el medio físico) hasta los actos del habla (speech acts). Las organizaciones van desde aquellas inspiradas en modelos biológicos hasta las gobernadas por leyes sociales basadas en modelos sociológicos. El propósito de Vowel engineering es lograr librerías de componentes que den soluciones al diseño de cada uno de estos aspectos, para que posteriormente, el diseñador seleccione un modelo de agente, un modelo de entorno, un modelo de interacciones y modelos de organización a instanciar [1]. En la figura A.1 se observa la arquitectura en capas de esta metodología

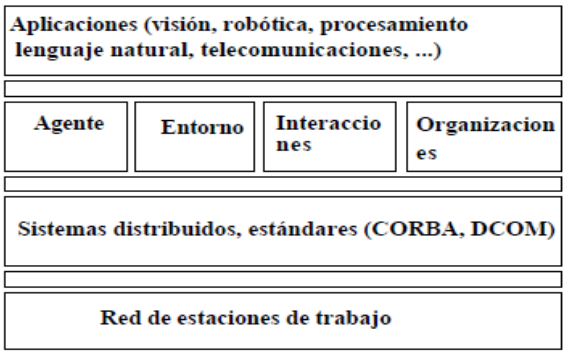


Figura A.1. Arquitectura de capas de *Vowel Engineering* [1].

Como ejemplo, propone para aspectos de interacción un lenguaje para la descripción de protocolos de interacción basado en procesos de comunicación síncronos o asíncrono donde la semántica es muy similar a la de los actos del habla. La representación en sí se hace mediante redes de transición en las que los arcos se corresponden con los mensajes intercambiados y los estados reflejan la situación global (por ejemplo. no hay posibilidad de que un estado se refiera al estado de un agente concreto). Una de las más recientes publicaciones de Vowel Engineering propone la implementación mediante la plataforma Volcano. La plataforma utiliza el ensamblaje de componentes utilizando lenguajes de composición arquitecturas, concretamente *UniCon.* El desarrollo consiste en ensamblar componentes que pueden ser desarrolladas ad-hoc o proceder de una librería. Cada componente pertenece a una categoría concreta de las cuatro consideradas. Pese a que en la literatura se indica que existe un entorno de desarrollo

basado en estas ideas, este no es público. De todas formas, el trabajo desarrollado dentro de *MAGMA* con esta metodología es reseñable debido a su variedad en los dominios de aplicación (Sistemas de información geográfica, Robocup, simulaciones de mercados o agentes en tiempo real). *Vowel Engineering* ha sido una de las primeras metodologías en modelar sistemas utilizando diferentes vistas. Aunque es prometedor, el trabajo en *Vowel Engineering* está incompleto ya que no termina de estabilizarse con herramientas de soporte.

A.2 *MaSE*

MaSE (*Multi-agent systems Software Engineering*) se concibe como una abstracción del paradigma orientado a objetos donde los agentes son especializaciones de objetos. En lugar de simples objetos, con métodos que pueden invocarse desde otros objetos, los agentes se coordinan unos con otros vía conversaciones y actúan pro activamente para alcanzar metas individuales y del sistema. En MaSE los agentes son simplemente una abstracción conveniente, que puede o no poseer inteligencia. En este sentido, los componentes inteligentes y no inteligentes se gestionan igualmente dentro del mismo armazón. Dado el enfoque inicial, los agentes se ven como especializaciones de objetos. De hecho, el sistema se construye sobre tecnología orientada a objetos y su aplicación a la especificación y diseño de sistemas multi-agente [1].

El análisis en *MaSE* consta de tres pasos: capturar los objetivos, aplicar los casos de uso y refinar roles. El diseño consta de cuatro pasos: crear clases de agentes, construir conversaciones, ensamblar clases de agentes y diseño del sistema. La mayoría de estos pasos se ejecutan dentro de la herramienta que soporta *MaSE*, *AgentTool.* Como productos de estas etapas, *MaSE* espera: diagramas de secuencia para especificar interacciones, diagramas de estados para representar procesos internos a las tareas y modelar interacciones, descomposición del sistema (agente) en subsistemas (componentes del agente) e interconexión de los mismos (definición de la arquitectura del agente). Estos elementos son característicos del *UML* La integración de estos diagramas en el proceso de desarrollo parece demasiado simple. Al final, la metodología podría traducirse como tome la herramienta de soporte y rellene los diferentes apartados. Esto supone ignorar que, como en el modelo *BDI*, se tienen dependencias entre los diagramas propuestos (como entre los diagramas de secuencia y las conversaciones) y que no es tan sencillo el saber qué máquinas de estados definen la ejecución de una tarea en el contexto de una interacción.

La herramienta de soporte permite generar código *auto*máticamente a partir de la especificación. La generación de código es independiente del lenguaje de programación utilizado, ya que se realiza recorriendo las estructuras de datos generadas en la especificación y generando texto como salida. No obstante, el proceso de generación de código es mejorable, ya que el código de los componentes a generar está entremezclado.

A.3 *GAIA*

GAIA es una metodología para el diseño de sistemas basados en agentes cuyo objetivo es obtener un sistema que maximice alguna medida de calidad global (no se llega a detallar cuál). *GAIA* pretende ayudar al analista a ir sistemáticamente desde unos requisitos iníciales a un diseño que, según los *auto*res, esté lo suficientemente detallado como para ser implementado directamente [1]. *En GAIA* se entiende que el objetivo del análisis es conseguir comprender el sistema y su estructura sin referenciar ningún aspecto de implementación. Esto se consigue a través de la idea de organización. Una organización en *GAIA* es una colección de roles, los cuales mantienen ciertas relaciones con otros y toman parte en patrones institucionalizados de interacción con otros roles. Los roles agrupan cuatro aspectos: Responsabilidades del agente, los recursos que se le permite utilizar, las tareas asociadas e interacciones. *GAIA* propone trabajar inicialmente con un análisis a alto nivel. En este análisis se usan dos modelos, el modelo de roles para identificar los roles clave en el sistema junto con sus propiedades definitorias y el modelo de interacciones que define las interacciones mediante una referencia a un modelo institucionalizado de intercambio de mensajes, como el *FIPA-Request*. Tras esta etapa, se entraría en lo que *GAIA* considera diseño a alto nivel. El objetivo de este diseño es generar tres modelos: el modelo de agentes que define los tipos de agente que existen, cuántas instancias de cada tipo y qué papeles juega cada agente, el modelo de servicios que identifica los servicios (funciones del agente) asociados a cada rol, y un Modelo de conocidos, que define los enlaces de comunicaciones que existen entre los agentes. A partir de aquí, los *auto*res de *GAIA* proponen aplicar técnicas clásicas de diseño orientado a objetos. Sin embargo, *GAIA* declara que queda fuera de su ámbito. Esta metodología sólo busca especificar cómo una sociedad de agentes colabora para alcanzar los objetivos del sistema, y qué se Requiere de cada uno para lograr esto último. En la figura A.2 se muestra los modelos de la metodología *GAIA*.

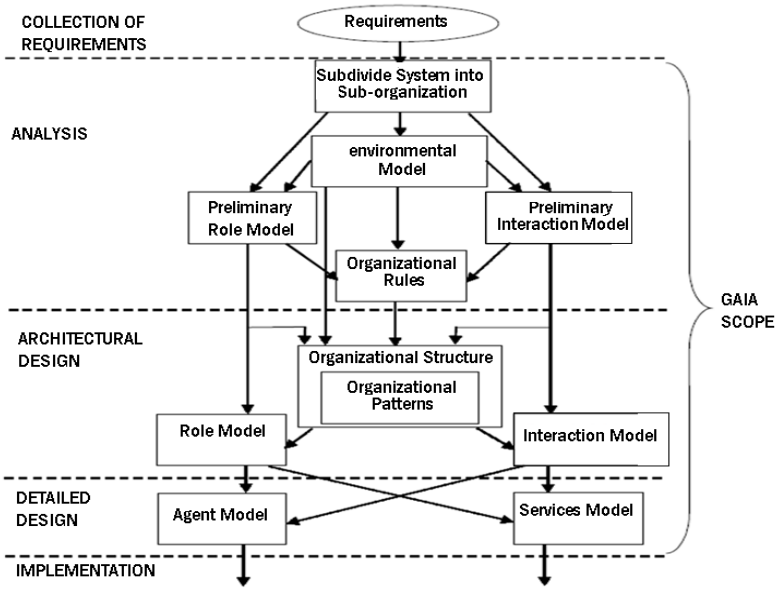


Figura A.2. Modelos en la metodología *GAIA* y sus relaciones en el proceso *GAIA* [3]

A.4 *MESSAGE*

MESSAGE trata de integrar resultados de las metodologías anteriores. Propone el análisis y diseño del sistema multi-agente (SMA) desde cinco puntos de vista para capturar los diferentes aspectos de un SMA: el de Organización, que captura la estructura global del sistema; el de Tareas/Objetivos, que determina qué hace el SMA y sus agentes constituyentes en términos de los objetivos que persiguen y las tareas implicadas en el proceso; el de Agente, que contiene una descripción detallada y extensa de cada agente y rol dentro del SMA; el de Dominio que actúa como repositorio de información (para entidades y relaciones) concernientes al dominio del problema; y el de Interacción, que trata las interacciones a distintos niveles de abstracción. Estos elementos están presentes en los dos modelos fundamentales que propone *MESSAGE*: el modelo de análisis y el modelo de diseño. El modelo de análisis se limita a generar modelos a partir de los meta-modelos [1]. El modelo de diseño no llegó a concretarse completamente.

Se decidió que el propósito del diseño sería producir entidades computacionales que representen el SMA descrito en el análisis. Por ello, cada artefacto producido en el análisis debería transformarse en una entidad computacional o varias cuyo comportamiento fuera el que se esperaba en el análisis. Esto significa que las entidades del análisis se deberían traducir a subsistemas, interfaces, clases, signaturas de operaciones, algoritmos, objetos, diagramas de objetos y otros. *MESSAGE* aporta mejoras en cuanto a conceptos de ingeniería respecto de las alternativas existentes, entre ellas el desarrollo dentro de un paradigma de ingeniería del software (el Proceso Racional Unificado), aportación de métodos para la traducción de entidades de análisis a entidades de diseño y guías para la generación de los modelos. Sin embargo, los objetivos de *MESSAGE* no se completaron totalmente.

La integración con el Proceso Racional Unificado no fue total, ya que las actividades definidas no se adecuaban a las necesidades reales y no se indicó cómo encajaban dentro de este proceso. Además, faltó trabajo en el estudio de las interdependencias entre los distintos modelos propuestos.

A favor de *MESSAGE* hay que destacar que ha sido la primera metodología en utilizar una herramienta para soporte del proceso de especificación de SMA de forma visual, como en UML.

En cuanto a la implementación, *MESSAGE* provee guías en cuanto a posibles arquitecturas y componentes a utilizar en esta etapa. Basándose en estas guías y los modelos de análisis y diseño, se realizó manualmente la implementación, lo cual hizo que se detectaran incorrecciones en las definiciones iníciales de los modelos. Esta experiencia es la base de la crítica realizada con anterioridad a *ZEUS.* En la siguiente figura se muestra los meta-modelos de la metodología *MESSAGE*.

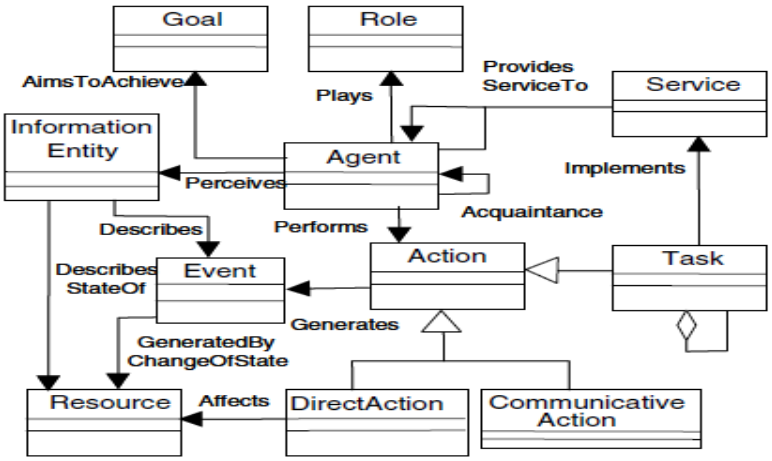


Figura A.3. Meta-modelos que se especifican en la metodología *MESSAGE* [2]

A.5 *MAS-CommonKADS*

Esta metodología extiende *CommonKADS* aplicando ideas de metodologías orientadas a objetos para su aplicación a la producción de SMA. La metodología *CommonKADS* gira alrededor del modelo de experiencia y está pensada para desarrollar sistemas expertos que interactúen con el usuario. De hecho considera sólo dos agentes básicos: el usuario y el sistema. *MASCommonKADS* extiende los modelos de *CommonKADS* para tener en cuenta la posibilidad de que dos o más componentes del sistema interactúen. *MAS-CommonKADS* ha sido la primera en plantear un desarrollo de SMA integrado con un ciclo de vida de software, concretamente el espiral dirigido por riesgos. Propone siete modelos para la definición del sistema: agente, tareas, experiencia, coordinación, comunicación, organización y diseño. Cada modelo presenta referencias a la teoría sobre la que se basa. El modelo en sí parte de una descripción gráfica luego se complementa con explicaciones en lenguaje natural de cada elemento [1]. Según la web de *MAS-CommonKADS* existen herramientas de soporte que han sido desarrolladas. Estas herramientas aparecen bajo el epígrafe *MAST* (*MultiAgent Systems Tool*). Sin embargo, la única que se puede descargar no es una herramienta para apoyo al análisis y diseño, sino un conjunto de arquitecturas y frameworks de agentes. En la siguiente figura se muestra los modelos de *MAS-CommonKADS*.

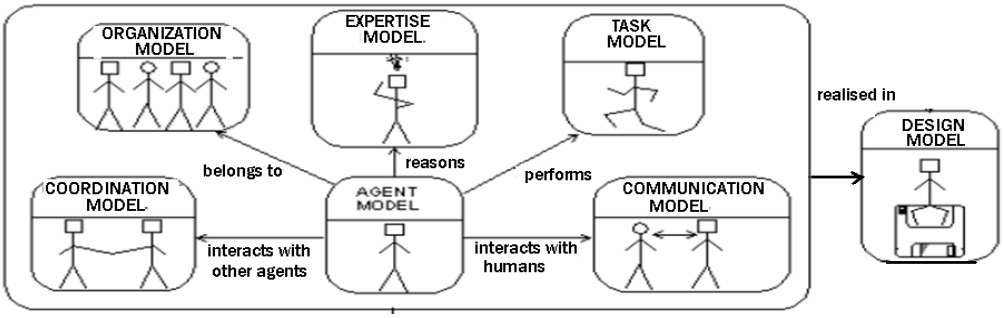


Figura A.4. modelos de *MAS-CommonKADS*. [4]

ANEXO B: DISEÑO DE UNA ARQUITECTURA BASADA EN MULTI-AGENTES PARA LA PLANIFICACIÓN DE AGARRE ESTABLE BAJO LA METODOLOGÍA PROMETHEUS

B.1 ESPECIFICACIÓN DEL SISTEMA

B.1.1 Diagrama De Objetivos

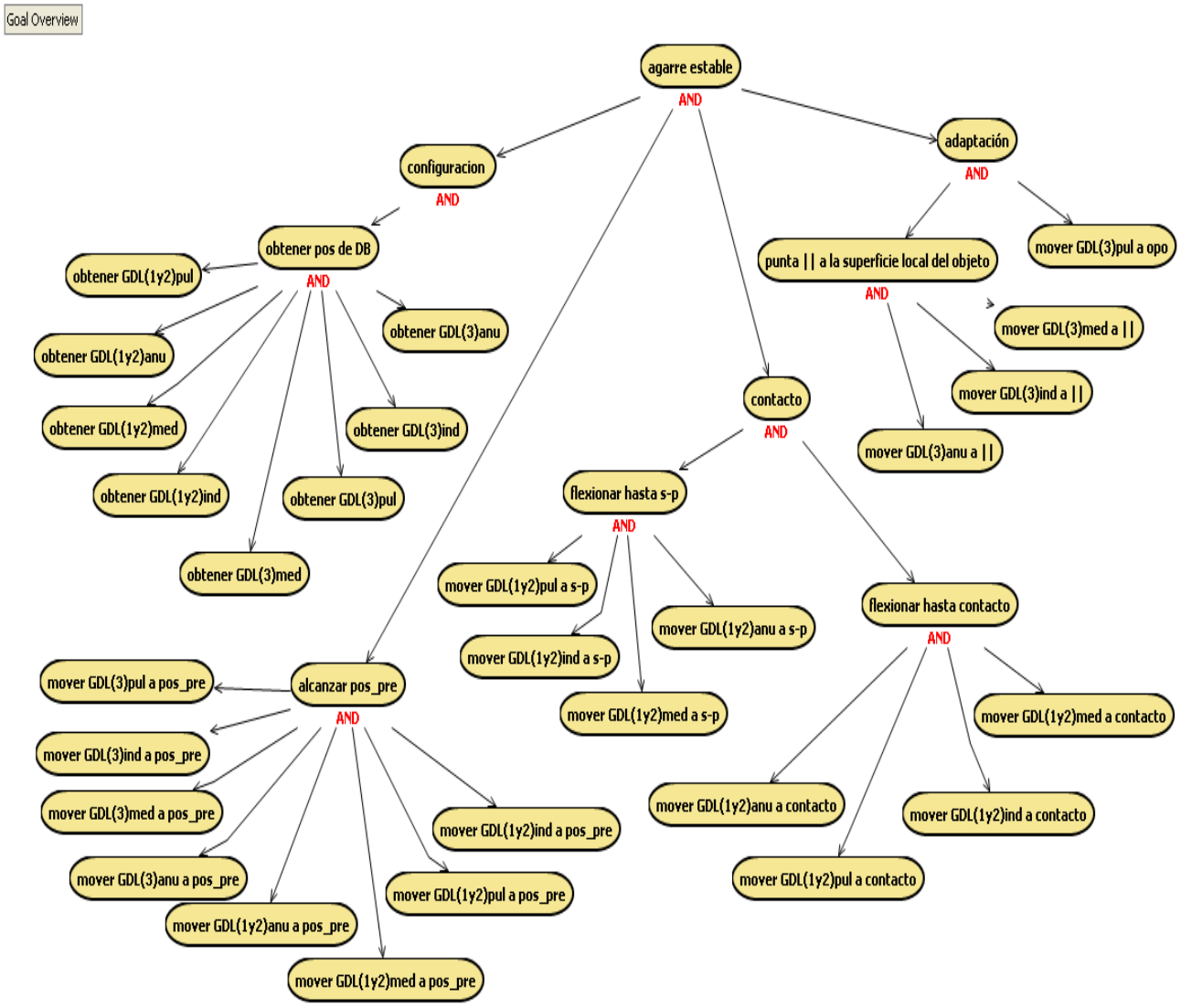


Figura B.1.1. Diagrama de Objetivos para el sistema de planificación.

B.1.2 Funcionalidades

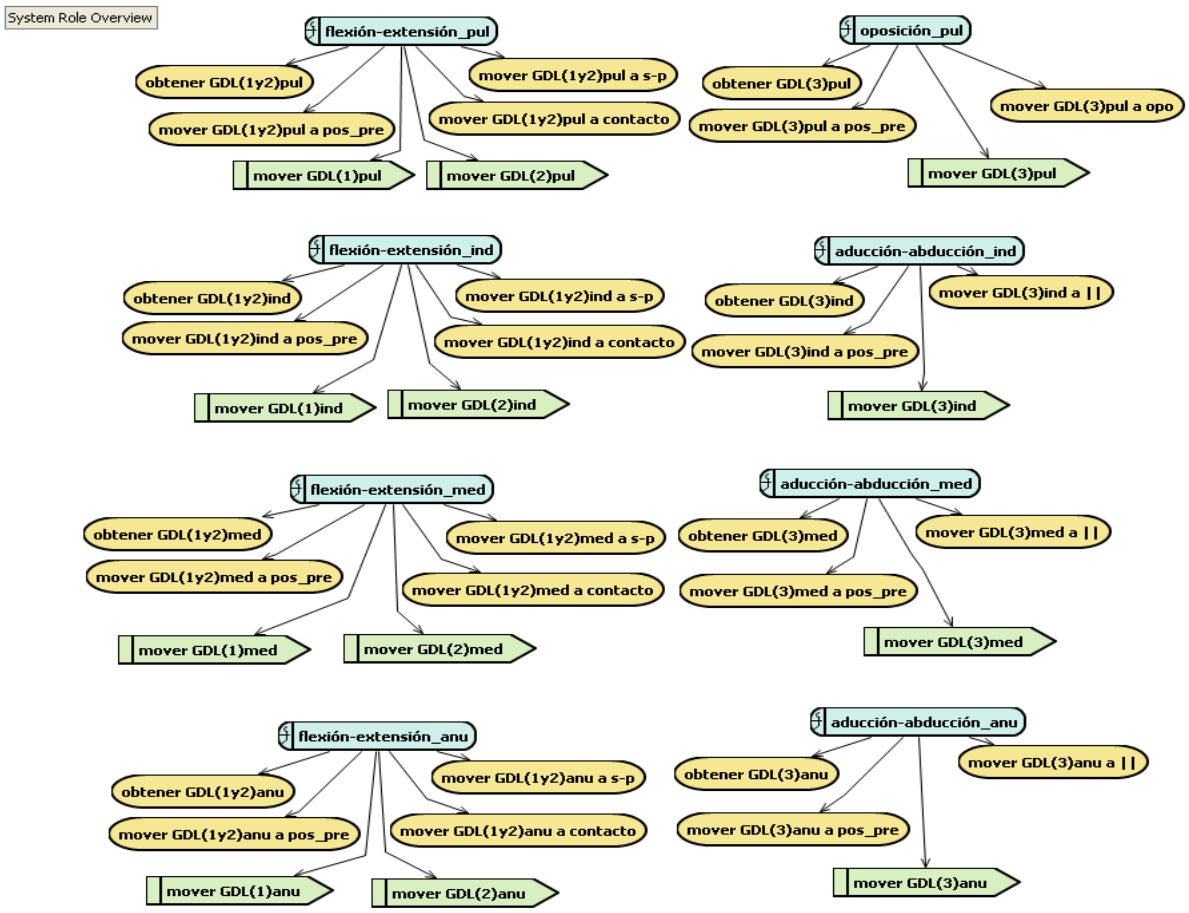


Figura B.1.2. Diagrama de funcionalidades para el sistema de planificación del agarre.

B.1.3 Descriptores De Las Funcionalidades

1. Funcionalidad De Flexión-Extensión_ind

Nombre: **Flexión-Extensión_ind**

Descripción: Esta funcionalidad es la encargada de realizar y regular el movimiento de flexión-extensión del dedo índice. Con dos propósitos, alcanzar la postura de pre-agarre y que el dedo toque el objeto al mismo tiempo que los demás.

Objetivos: Esta funcionalidad logra los siguientes objetivos: obtener GDL(1y2)ind, mover GDL(1y2)ind a pos_pre, mover GDL(1y2)ind a s-p, mover GDL(1y2)ind a contacto.
Acciones: Mover a la posición deseada las articulaciones encargadas de flexión-extensión del índice (mover GDL(1)ind; mover GDL(2)ind).
Accionadores (trigger): *i*) Llegada del valor del tipo de agarre; *ii*) Distancia del dedo al objeto menor o igual al set-point.
Información usada: posición de las articulaciones de flexión-extensión que forman la postura de pre-agarre del dedo índice.
Información producida: Distancia del dedo índice al objeto**.**

Comentario del descriptor:

La información producida por esta funcionalidad es de vital importancia ya que es utilizada por lo demás dedos para tocar el objeto al mismo tiempo. Como se detallará mas adelante esta información es tomada por el dedo pulgar el cual, entre otras cosas, es el encargado de actualizar el set-point

2. Funcionalidad De Flexión-Extensión_med

Nombre: **Flexión-Extensión_med**

Descripción: Esta funcionalidad es la encargada de realizar y regular el movimiento de flexión-extensión del dedo medio. Con dos propósitos, alcanzar la postura de pre-agarre y que el dedo toque el objeto al mismo tiempo que los demás.
Objetivos: Esta funcionalidad logra los siguientes objetivos: obtener GDL(1y2)med, mover GDL(1y2)med a pos_pre, mover GDL(1y2)med a s-p, mover GDL(1y2)med a contacto.
Acciones: Mover a la posición deseada las articulaciones encargadas de flexión-extensión del medio (mover GDL(1)med; mover GDL(2)med).
Accionadores (trigger): *i*) Llegada del valor del tipo de agarre; *ii*) Distancia del dedo al objeto menor o igual al set-point.
Información usada: posición de las articulaciones de flexión-extensión que forman la postura de pre-agarre del dedo medio.
Información producida: Distancia del dedo medio al objeto**.**

3. Funcionalidad De Flexión-Extensión_anu

Nombre: **Flexión-Extensión_anu**

Descripción: Esta funcionalidad es la encargada de realizar y regular el movimiento de flexión-extensión del dedo anular. Con dos propósitos, alcanzar la postura de pre-agarre y que el dedo toque el objeto al mismo tiempo que los demás.

Objetivos: Esta funcionalidad logra los siguientes objetivos: obtener GDL(1y2)anu, mover GDL(1y2)anu a pos_pre, mover GDL(1y2)anu a s-p, mover GDL(1y2)anu a contacto.

Acciones: Mover a la posición deseada las articulaciones encargadas de flexión-extensión del anular (mover GDL(1)anu; mover GDL(2)anu).
Accionadores (trigger): *i*) Llegada del valor del tipo de agarre; *ii*) Distancia del dedo al objeto menor o igual al set-point.
Información usada: posición de las articulaciones de flexión-extensión que forman la postura de pre-agarre del dedo anular.
Información producida: Distancia del dedo anular al objeto**.**

4. Funcionalidad De Flexión-Extensión_Pul

Nombre: **Flexión-Extensión_pul**
Descripción: Esta funcionalidad es la encargada de realizar y regular el movimiento de flexión-extensión del dedo pulgar. Con dos propósitos, alcanzar la postura de pre-agarre y que el dedo toque el objeto al mismo tiempo que los demás.
Objetivos: Esta funcionalidad logra los siguientes objetivos: obtener GDL(1y2)pul, mover GDL(1y2)pul a pos_pre, mover GDL(1y2)pul a s-p, mover GDL(1y2)pul a contacto.
Acciones: Mover a la posición deseada las articulaciones encargadas de flexión-extensión del pulgar (mover GDL(1)pul; mover GDL(2)pul).
Accionadores (trigger): *i*) Llegada del valor del tipo de agarre; *ii*) Distancia del dedo al objeto menor o igual al set-point.
Información usada: posición de las articulaciones de flexión-extensión que forman la postura de pre-agarre del dedo pulgar.
Información producida: Distancia del dedo pulgar al objeto**.**

5. Funcionalidad De Aducción-Abducción_Ind

Nombre: Aducción-Abducción_ind

Descripción: Esta funcionalidad es la encargada de realizar y regular el movimiento de aducción-abducción del dedo índice. Con tres propósitos: alcanzar la postura de pre-agarre; lograr que la superficie de su punta sea paralela a superficie local del objeto; y por último, evitar una colisión con el dedo medio.
Objetivos: Esta funcionalidad logra los siguientes objetivos: obtener GDL(3)ind; mover GDL(3)ind a pos_pre ; mover GDL(3)ind a ||
Accion: Mover a posición la articulación encargada de la aducción-abducción del índice (mover GDL(3)ind).
Accionadores (trigger): *i*) Llegada del valor del tipo de agarre; *ii*) distancia del dedo al objeto menor o igual a una referencia específica (3.5 cm). Esta distancia corresponde al momento cuando entran a funcionar los sensores de proximidad ópticos. No confundir con el Set-point.

Información usada: *i)*posición de la articulación de aducción-abducción que forma la postura de pre-agarre del dedo índice; *ii*) posición de la articulación de aducción-abducción del dedo medio.
Información producida: posición de la articulación de aducción-abducción del dedo índice.

Comentario del descriptor:

La información sobre la posición de la articulación de aducción-abducción es usada por los demás dedos para que no ocurra una colisión entre ellos al momento de cumplir con los objetivos de esta funcionalidad.

6. Funcionalidad De Aducción-Abducción_Med

Nombre: Aducción-Abducción_med

Descripción: Esta funcionalidad es la encargada de realizar y regular el movimiento de aducción-abducción del dedo medio. Con tres propósitos: alcanzar la postura de pre-agarre; lograr que la superficie de su punta sea paralela a superficie local del objeto; y por último, evitar una colisión con los dedos índice y anular.
Objetivos: Esta funcionalidad logra los siguientes objetivos: obtener GDL(3)med; mover GDL(3)med a pos_pre ; mover GDL(3)med a ||
Accion: Mover a posición la articulación encargada de la aducción-abducción del medio (mover GDL(3)med).
Accionadores (trigger): *i*) llegada del valor del tipo de agarre; *ii*) distancia del dedo al objeto menor o igual a una referencia específica (3.5 cm). Esta distancia corresponde al momento cuando entran a funcionar los sensores de proximidad ópticos. No confundir con el Set-point.
Información usada: *i)* posición de la articulación de aducción-abducción que forma la postura de pre-agarre del dedo medio; *ii*) posición de la articulación de aducción-abducción de los dedos índice y anular.
Información producida: posición de la articulación de aducción-abducción del dedo medio.

7. Funcionalidad De Aducción-Abducción_Anu

Nombre: **Aducción-Abducción_anu**

Descripción: Esta funcionalidad es la encargada de realizar y regular el movimiento de aducción-abducción del dedo anular. Con tres propósitos: alcanzar la postura de pre-agarre; lograr que la superficie de su punta sea paralela a superficie local del objeto; y por último, evitar una colisión con el dedo medio.

Objetivos: Esta funcionalidad logra los siguientes objetivos: obtener GDL(3)anu; mover GDL(3)anu a pos_pre ; mover GDL(3)anu a ||

Accion: Mover a posición la articulación encargada de la aducción-abducción del anular (mover GDL(3)anu).
Accionadores (trigger): *i*) llegada del valor del tipo de agarre; *ii*) distancia del dedo al objeto menor o igual a una referencia específica (3.5 cm). Esta distancia corresponde al momento cuando entran a funcionar los sensores de proximidad ópticos. No confundir con el Set-point.
Información usada: *i)* posición de la articulación de aducción-abducción que forma la postura de pre-agarre del dedo anular; *ii*) posición de la articulación de aducción-abducción del dedo medio.
Información producida: posición de la articulación de aducción-abducción del dedo anular.

8. Funcionalidad De Oposición_Pul

Nombre: Oposición_pul

Descripción: Esta funcionalidad es la encargada de oponer el pulgar al dedo anular.
Objetivos: obtener GDL(3)pul; mover GDL(3)pul a pos_pre ; mover GDL(3)pul a opo.
Accion: Mover a posición la articulación encargada de la oposición del pulgar (mover GDL(3)pul).
Accionadores (trigger): *i*) Llegada del valor del tipo de agarre; *ii*) movimiento de la articulación de aducción-abducción del dedo anular.
Información usada: *i*) posición de la articulación de oposición para formar la postura de pre-agarre; *i*) posición de la articulación de aducción-abducción del dedo anular.
Información producida: no hay.

B.1.4 Especificación De Los Escenarios

Tabla 1. Escenario de configuración

Nombre: Configuración
Descripción: En este escenario, de acuerdo al valor del tipo de agarre, cada dedo se encarga de alcanzar su postura de pre-agarre.
Accionador (Trigger): llegada del valor del tipo de agarre.
Pasos:

#	Tipo de paso	Paso	Funcionalidad	Datos: * Usados + Producidos
1	Percepción	Valor del tipo de agarre	Aducción-Abducción y/o oposición	
2	Percepción	posición GDL(1y2)	Flexión-Extensión	
3	Percepción	posición GDL(3)	Aducción-Abducción	
4	Objetivo	obtener GDL(1y2)	Flexión-Extensión	***BD-dedo.**
5	Objetivo	obtener GDL(3)	Aducción-Abducción y/o Oposición	***** BD-dedo.**
6	Objetivo	mover GDL(1y2) a pos_pre	Flexión-Extensión	
7	Objetivo	mover GDL(3) a pos_pre	Aducción-Abducción y/o oposición	
8	Acción	mover GDL(1)	Flexión-Extensión	
9	Acción	mover GDL(2)	Flexión-Extensión	
10	Acción	mover GDL(3)	Aducción-Abducción	

Alternativas: no hay.

Notación usada:
BD-dedo: Dato que contiene la posición articular del dedo en consideración, que conforma la la postura de pre-agarre. El nombre de la base de datos consultada varía de acuerdo al dedo que lo la contenga. Por el ejemplo, para el dedo índice su nombre es: **BD-índice**

Tabla 2. Escenario de adaptación por cierre

Nombre: adaptación por cierre

Descripción: En este escenario se describen los pasos necesarios para que todos los dedos lleguen al set-point.

Accionador (Trigger): Señal de que se ha alcanzado la Postura de pre-agarre.

Pasos:

#	Tipo de paso	Paso	Funcionalidad	Datos: * Usados + Producidos
1	Percepción	posición GDL(1y2)	Flexión-Extensión	
2	Percepción	distancia	Flexión-Extensión,	

			Aducción-Abducción y/o Oposición_pul	
3	Objetivo	actualizar s-p	Flexión-Extensión	***s-p, d_ind, d_med, d_anu, d_pul.** **+ s-p**
4	Objetivo	mover GDL(1y2) a s-p	Flexión-Extensión	***** s-p, d_ind, d_med ,d_anu, d_pul** **+ s-p**
5	Acción	mover GDL(1)	Flexión-extensión	
6	Acción	mover GDL(2)	Flexión-extensión	
7	Escenario	contacto		

Alternativas:
La ejecución de todos los pasos de este escenario están limitados por el segundo paso. Si la distancia desde la falange proximal hasta el objeto ya es aproximadamente igual al set-point, entonces solo se ejecutan los pasos del 1 al 5 hasta que la punta del dedo alcance el set-point. Por otro lado si la distancia desde la punta del dedo al objeto es igual al set-point, entonces no se ejecuta ningún paso. La única manera en la cual se pueden ejecutar lo pasos del 1 al 6 es que ni la falange proximal ni la punta de los dedos hayan alcanzado el set-point.
Después de que la punta de los dedos alcance el set-point se ejecuta el escenario de contacto (paso 7).
Notacion usada:
d_ind: Distancia hasta el objeto desde la falange proximal y desde la punta del dedo índice.
d_med: Distancia hasta el objeto desde la falange proximal y desde la punta del dedo medio.
d_anu: Distancia hasta el objeto desde la falange proximal y desde la punta del dedo anular.
d_pul: Distancia hasta el objeto desde la punta del dedo pulgar.

Tabla 3. Escenario de adaptación por alineación

Nombre: adaptación por alineación

Descripción: En este escenario el movimiento del dedo conduce a que la superficie de su punta sea lo más paralela posible a la pose local del objeto.

Accionador (Trigger): Distancia de la punta del dedo al objeto en el rango 0.5cm<=distancia<=3.5cm.

Pasos:

#	Tipo de paso	paso	Funcionalidad	Datos: * Usados + Producidos
1	Percepción	posición GDL(3)	Aducción-abducción Oposición_pul	
2	Percepción	pose: orientación de la superficie local del objeto	Aducción-abducción	
3	Percepción	distancia	Aducción-abducción Oposición	
4	Objetivo	mover GDL(3) a || y/o mover GDL(3)pul a opo.	Aducción-abducción Oposición_pul	***posición GDL(3)ind; posición GDL(3)med; y posición GDL(3)anu.** **+ posición GDL(3)ind; posición GDL(3)med; posición GDL(3)anu.**
5	Acción	Mover GDL(3)	Aducción-abducción Oposición_pul	
Alternativas: los pasos del 1 a 5 se repiten cíclicamente hasta que la distancia al objeto desde la punta del dedo o desde la falange proximal sea menor o igual a 0.5cm. El objetivo mover GDL(3)pul a opo (paso 4), solo le concierne al dedo pulgar y depende de los movimientos de aducción-abducción del dedo anular. **Notacion usada:** **posición GDL(3):** Posición actual de la articulación en la cual se llevan a cabo los movimientos de aducción-abducción del GDL(3). En el caso del dedo pulgar es la posición actual de la articulación que realiza el movimiento de oposición.				

Tabla 4. Escenario de contacto

Nombre: Contacto

Descripcion: Luego que todos los dedos han alcanzado el set-point, este escenario describe los pasos que se deben llevar a cabo para que todos contacten el objeto.

Accionador (Trigger): Distancia de la punta de todos los dedos al objeto aproximadamente

igual al set-point. **Pasos:**				
#	**Tipo de paso**	**paso**	**Funcionalidad**	**Datos:** *** Usados** **+ Producidos**
1	Percepción	posición GDL(1y2)	Flexión-Extensión	
2	Percepción	distancia	Flexión-Extensión	
3	Objetivo	mover GDL(1y2) a contacto	Flexión-Extensión	*** d_ind;d_med;d_anu;d_pul**
4	Acción	mover GDL(1)	Flexión-Extensión	
5	Acción	mover GDL(2)	Flexión-Extensión	
Alternativas: el paso 5 se ejecuta únicamente si la distancia desde la falange proximal al objeto es diferentes de cero. Para los demás casos los pasos 1 al 5 se ejecutan cíclicamente hasta que la distancia al objeto desde la punta de los dedos y desde la falange proximal sea igual a cero.				

B.1.5 Percepciones, Acciones Y Datos

1. Percepciones

A continuación se explican las percepciones para el agente índice. Los demás agentes tienen las mismas percepciones por lo tanto no es necesario detallarlas.

- **Valor del tipo de Agarre**

  Nombre: tipo de agarre
  Descripción: información que indica el tipo de agarre que desea realizar el paciente.
  Información llevada: Valor entero de un byte. A cada bit se le asigna un tipo de agarre [*26*].
  Actualización de la información: Actualiza el valor que representa el tipo de agarre que desea realizar el paciente.
  Recurso: electrodos.
  Procesamiento: el agente recibe esta percepción directamente y el análisis de su información se hace internamente.
  Agentes: pulgar, índice, medio y anular.
  Frecuencia esperada: solo se espera que llegue una vez.

- **Pose local del objeto vista por el dedo índice**

  Nombre: pose_ind.
  Descripción: Informa sobre la orientación de la superficie local del objeto que es vista por la punta del dedo índice.
  Información llevada: orientación en el eje X y eje Z del plano que representa la superficie local que esta en frente de la punta del dedo índice.
  Actualización de la información: pose local del objeto.
  Recurso: sensores ópticos
  Procesamiento: la percepción es recibida directamente por el agente. Pero la lectura de los sensores ópticos ha sido antes procesada para informar a los agentes sobre la pose local del objeto.
  Agentes: índice.

- **Distancia al objeto vista por el dedo índice**

  Nombre: d_ind.
  Descripción: Se informa sobre la distancia hasta el objeto desde la punta del dedo índice y desde su falange proximal.
  Información llevada: distancia en mm de la punta y falange del dedo índice al objeto.
  Actualización de la información: distancia del dedo índice al objeto.
  Recurso: sensores ópticos.
  Procesamiento: la percepción es recibida directamente por el agente. Pero la lectura de los sensores ópticos ha sido antes procesada para informar al agente sobre la distancia.
  Agentes que responden: índice.

- **Posición de los grados de libertad que permiten los movimientos de Flexión-Extensión del dedo índice.**

  Nombre: posición GDL(1y2)ind
  Descripción: Obtiene las posición actual de los grados de libertad que permiten los movimientos de flexión-extensión del dedo índice.
  Información llevada: ángulos correspondientes a GDL(1) y GDL(2).
  Conocimiento actualizado: posición actual de GDL(1) y GDL(2).
  Recurso: Sensores de posición en cada articulación del dedo índice.
  Procesamiento: A la lectura de los sensores simplemente se les hace una adecuación deseñal para librarla de ruido, se podría decir que se usa la lectura directamente del sensor sin necesidad de procesamientos complejos.
  Agentes que responden: índice.

- **Posición de los grados de libertad que permiten el movimiento de aducción-abducción del dedo índice.**

Nombre: posición GDL(3)ind.
Descripción: Obtiene la posición actual de GDL(3).
Información llevada: ángulo correspondiente a GDL(3).
conocimiento actualizado: posición actual de GDL(3).
Recurso: Sensor de posición para GDL(3).
Procesamiento: A la lectura del sensor simplemente se le hace una adecuación de señal para librarla de ruido, se podría decir que se usa la lectura directamente del sensor sin necesidad de procesamientos complejos.
Agentes que responden: índice.

2. Acciones

Así como las percepciones, las acciones también pueden ser complejas, requiriendo un rigoroso diseño y desarrollo fuera del paradigma de agentes, especialmente, cuando se usan actuadores físicos, como en el caso de una mano robótica.

Los descriptores que se muestran a continuación contienen toda la información necesaria de las acciones que realiza el agente índice sobre el ambiente. No se ha tenido en cuenta la acción mover GDL(2)ind. Ya que la información que contiene su descriptor es básicamente la misma a la contenida en el descriptor de la acción mover GDL(1)ind.

- **Mover GDL(1) para contribuir al proceso de flexión del dedo índice.**

  Nombre: mover GDL(1)ind.
  Descripción: mover GDL(1) para contribuir al proceso de flexión del dedo índice.
  Parámetros: Aceleración y velocidad angular de las articulaciones.
  Temporalidad: Duracional (puede tomar algunos ms).
  Detección de fallas: Una falla ocurre cuando la posición de GDL(1) no es alcanzada en un determinado tiempo o nunca es alcanzada.
  Cambio parcial: desplazamiento de la posición de GDL(1) hasta el instante en el cual ocurrió la falla.
  Efectos secundarios: reducción de la potencia de la batería.

- **Mover GDL(3) para permitir los movimientos de aducción-abducción del dedo índice.**

  Nombre: mover GDL(3)ind.
  Descripción: mover GDL(3) para permitir los movimientos de aducción-abducción del dedo índice.
  Parámetros: aceleración y velocidad angular de las articulaciones,
  Temporalidad: Duracional (puede tomar algunos ms)
  Detección de fallas: Una falla ocurre cuando la posición articular no es alcanzada en un determinado tiempo o nunca es alcanzada.

Cambio parcial: desplazamiento de la articulación hasta el instante en el cual ocurrió la falla.
Efectos secundarios: reducción de la potencia de la batería.

3. Datos

En esta sección se desarrollan los descriptores de las bases de datos que fueron encontradas en la fase de especificación del sistema. A continuación solo se desarrolla el descriptor de la base de datos del agente índice y pulgar. Los descriptores de las bases de datos de los agentes: anular y medio contienen básicamente la misma información del descriptor de la base de datos del agente índice por lo tanto no se datallan.

Nombre: BD dedo índice.

Descripción: En esta base de datos se guardan tres tipos de datos: i) información sobre la posición que deben alcanzar GDL(1y2) y GDL(3) para formar la postura de pre-agarre del dedo índice. ii) ángulos máximos y mínimos de GDL(1y2) y GDL(3) (limites articulares). iii) posición actual de GDL(3).
Tipos de datos: flotante.
Campos o aspectos incluidos: los tres tipos de datos que se describieron anteriormente.
Producido por: agente índice.
Usado por: el agente medio y el agente índice.
Persistente (*este ítem detalla si la información en la base de datos esta siendo constantemente modificada o es persistente, es decir, constante):* Los 2 primeros tipos de datos que se guardan en la base de datos son persistentes pero el último no lo es.
Inicialización: ninguna.
Cuando se usa: cuando llega el valor del tipo de agarre y cuando él o un agente cercano intenta realizar movimientos de aducción-abducción.

Nombre: BD dedo pulgar.

Descripción: En esta base de datos se guardan dos tipos de datos; i) información sobre la posición que deben alcanzar GDL(1y2) y GDL(3) del dedo pulgar para formar la postura de pre-agarre. ii) ángulos máximos y mínimos de GDL(1y2) y GDL(3) (limites articulares).
Tipos de datos: flotante.
Campos o aspectos incluidos: los dos tipos de datos que se describieron anteriormente
Producido por: agente pulgar.
Usado por: agente pulgar.
Persistente: si.
Inicialización: ninguna.
Cuando se usa: cuando llega el valor del tipo de agarre y cuando el agente pulgar necesita reconfigurar GDL(3).

B.2 DISEÑO DE LA ARQUITECTURA

B.2.1 Diagrama De Visión General Del Sistema

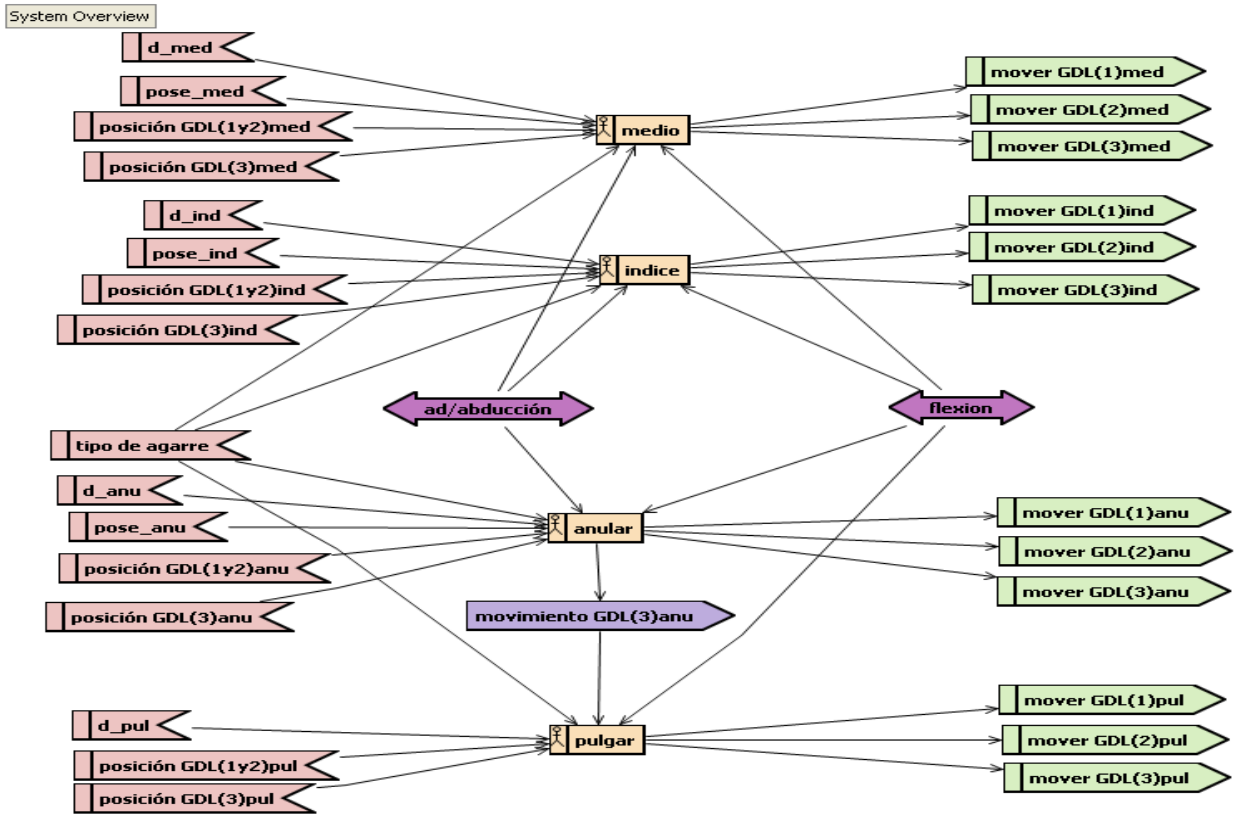


Figura B.2.1 Diagrama de visión general del sistema para la planificación de agarre en una mano robótica.

B.2.2 Descriptores De Agentes

1. Descriptor Del Agente Pulgar

Nombre: Pulgar

Descripción: Realiza los movimientos de Flexión-Extensión y oposición del dedo pulgar.
Cardinalidad: uno
Tiempo de vida: Desde la llegada del valor que identifica el tipo de agarre hasta el contacto con el objeto.

Inicialización: Lee de la *BD dedo pulgar* la posición de sus articulaciones para formar su postura de pre-agarre.
Desaparición: Al tomar contacto con el objeto, el agente pulgar debemantener posición de sus articulaciones.
Funcionalidades incluidas: Flexión-Extensión_pul y Oposición_pul.
Datos usados: *BD dedo pulgar, BD dedo índice, BD dedo medio y BD dedo anular.*
Datos producidos: no produce.
Objetivos:

- obtener GDL(1y2)pul;
- mover GDL(1y2)pul a pos_pre;
- mover GDL(1y2)pul a s-p
- mover GDL(1y2)pul a contacto
- obtener GDL(3)pul
- mover GDL(3)pul a pos_pre
- mover GDL(3)pul a ||

Percepciones a las que reacciona:
- valor del tipo de agarre
- d_pul
- posición GDL(1y2)pul.
- posición GDL(3)pul

Acciones que realiza:
- mover GDL(1)pul.
- mover GDL(2)pul.
- mover GDL(3)pul.

Protocolos de interacción: *flexión*

2. Descriptor Del Agente Medio

Nombre: Medio
Descripción: Realiza los movimientos de Aducción-Abducción y Flexión-Extensión del dedo medio.
Cardinalidad: uno
Tiempo de vida: Desde la llegada del valor que identifica el tipo de agarre hasta el contacto con el objeto.
Inicialización: Lee de la *BD dedo medio* la posición de sus articulaciones para formar su postura de pre-agarre.
Desaparición: Al tomar contacto con el objeto, el agente medio debemantener posición de sus articulaciones.
Funcionalidades incluidas: Flexión-Extensión_med, Aducción-Abducción_med.

Datos usados: *BD dedo medio, BD dedo índice y BD dedo anular*. Los dos últimos datos son usados para evitar la colision con los dedos anular o índice cuando realiza movimientos de aducción-abducción.
Datos producidos: *BD dedo medio.*

Objetivos:

- obtener GDL(1y2)med
- mover GDL(1y2)med a pos_pre
- mover GDL(1y2)med a s-p
- mover art(1y2)med a contacto
- obtener GDL(3)med
- mover GDL(3)med a pos_pre
- mover GDL(3)med a ||

Percepciones a las que reacciona:

- d_med.
- posición GDL(1y2)med.
- posición GDL(3)med.
- Pose_med.

Acciones:

- mover GDL(1)med.
- mover GDL(1)med.
- mover GDL(3)med

Protocolos de interacción: *flexión y ad/abducción.*

3. Descriptor Del Agente Indice

Nombre: Indice
Descripción: Realiza los movimientos de Aducción-Abducción y Flexión-Extensión del dedo índice.
Cardinalidad: uno
Tiempo de vida: Desde la llegada del valor que identifica el tipo de agarre hasta el contacto con el objeto.
Inicialización: Lee de la *BD dedo índice* la posición de sus articulaciones para formar su postura de pre-agarre.
Desaparición: Al tomar contacto con el objeto, el agente índice debemantener posición de sus articulaciones.
Funcionalidades incluidas: Flexión-Extensión_ind, Aducción-Abducción_ind.

Datos usados: *BD dedo medio, BD dedo índice*. El primer dato es usado para evitar la colisión con el dedo medio cuando realiza movimientos de aducción-abducción.
Datos producidos: *BD dedo índice.*

Objetivos:

- obtener GDL(1y2)ind
- mover GDL(1y2)ind a pos_pre
- mover GDL(1y2)ind a s-p
- mover art(1y2)ind a contacto
- obtener GDL(3)ind
- mover GDL(3)ind a pos_pre
- mover GDL(3)ind a ||

Percepciones a las que reacciona:

- d_ind.
- posición GDL(1y2)ind.
- posición GDL(3)ind.
- Pose_ind.

Acciones:

- mover GDL(1)ind.
- mover GDL(1)ind.
- mover GDL(3)ind

Protocolos de interacción: *flexión y ad/abducción.*

4. Descriptor Del Agente Anular

Nombre: Anular
Descripción: Realiza los movimientos de Aducción-Abducción y Flexión-Extensión del dedo anular.
Cardinalidad: uno
Tiempo de vida: Desde la llegada del valor que identifica el tipo de agarre hasta el contacto con el objeto.
Inicialización: Lee de la *BD dedo anular* la posición de sus articulaciones para formar su postura de pre-agarre.
Desaparición: Al tomar contacto con el objeto, el agente anular debemantener posición de sus articulaciones.
Funcionalidades incluidas: Flexión-Extensión_anu, Aducción-Abducción_anu.

Datos usados: *BD dedo medio, BD dedo anular*. El primer dato es usado para evitar la colisión con el dedo medio cuando realiza movimientos de aducción-abducción.
Datos producidos: *BD dedo anular.*

Objetivos:

- obtener GDL(1y2)anu
- mover GDL(1y2)anu a pos_pre
- mover GDL(1y2)anu a s-p
- mover art(1y2)anu a contacto
- obtener GDL(3)anu
- mover GDL(3)anu a pos_pre
- mover GDL(3)anu a ||

Percepciones a las que reacciona:

- d_anu.
- posición GDL(1y2)anu.
- posición GDL(3)anu.
- Pose_anu.

Acciones:

- mover GDL(1)anu.
- mover GDL(1)anu.
- mover GDL(3)anu.

Protocolos de interacción: *flexión y ad/abducción.*

B.2.3 Desarrollo De Los Descriptores De Los Protocolos

1. Asociación de las funcionalidades con el agente (s) que las contienen

En las siguientes tablas se asocia cada fucnionalidad con el agente (s) que la contiene. Cada tabla se refiere a un escenario en específico.

Tabla 5. Agentes asociados a cada una de las funcionalidades según el escenario de configuración**.**

	Tipo de	**Paso**	**Funcionalidad**	**Agente**

	paso			
1	Percepción	Valor del tipo de agarre	Aducción-Abducción y/o oposición	índice, medio, anular y pulgar
2	Percepción	posición GDL(1y2)	Flexión-Extensión	índice, medio, anular y pulgar
3	Percepción	posición GDL(3)	Aducción-Abducción	índice, medio, anular y pulgar
4	Objetivo	obtener GDL(1y2)	Flexión-Extensión	índice, medio, anular y pulgar
5	Objetivo	obtener GDL(3)	Aducción-Abducción y/o Oposición	índice, medio, anular y pulgar
6	Objetivo	mover GDL(1y2) a pos_pre	Flexión-Extensión	índice, medio, anular y pulgar
7	Objetivo	mover GDL(3) a pos_pre	Aducción-Abducción y/o oposición	índice, medio, anular y pulgar
8	Acción	mover GDL(1)	Flexión-Extensión	índice, medio, anular y pulgar
9	Acción	mover GDL(2)	Flexión-Extensión	índice, medio, anular y pulgar
10	Acción	mover GDL(3)	Aducción-Abducción	índice, medio, anular y pulgar

Tabla 6. Agentes asociados a cada una de las funcionalidades según el escenario de adaptación por cierre**.**

#	Tipo de paso	Paso	Funcionalidad	Agente
1	Percepción	posición GDL(1y2)	Flexión-Extensión	índice, medio, anular y pulgar
2	Percepción	distancia	Flexión-Extensión, Aducción-Abducción y/o Oposición_pul	índice, medio, anular y pulgar
3	Objetivo	actualizar s-p	Flexión-Extensión	índice, medio, anular y pulgar
4	Objetivo	mover GDL(1y2) a s-p	Flexión-Extensión	índice, medio, anular y pulgar
5	Acción	mover GDL(1)	Flexión-extensión	índice, medio, anular y pulgar
6	Acción	mover GDL(2)	Flexión-extensión	índice, medio, anular y pulgar
7	Escenario	contacto		

Tabla 7. Agentes asociados a cada una de las funcionalidades según el escenario de adaptación por alineación**.**

#	Tipo de paso	paso	Funcionalidad	Agente
1	Percepción	posición GDL(3)	Aducción-abducción Oposición_pul	índice, medio anular y pulgar.
2	Percepción	pose: orientación de la superficie local del objeto	Aducción-abducción	índice, medio y anular
3	Percepción	distancia	Aducción-abducción Oposición_pul	índice, medio anular y pulgar.
4	Objetivo	mover GDL(3) a || y/o mover GDL(3)pul a opo.	Aducción-abducción Oposición_pul	índice, medio anular y pulgar.
5	Acción	Mover GDL(3)	Aducción-abducción Oposición_pul	índice, medio anular y pulgar.

Tabla 8. Agentes asociados a cada una de las funcionalidades según el escenario de contacto.

#	Tipo de paso	paso	Funcionalidad	Agentes
1	Percepción	posición GDL(1y2)	Flexión-Extensión	índice, medio, anular y pulgar.
2	Percepción	distancia	Flexión-Extensión	índice, medio, anular y pulgar.
3	Objetivo	mover GDL(1y2) a contacto	Flexión-Extensión	índice, medio, anular y pulgar.
4	Acción	mover GDL(1)	Flexión-Extensión	índice, medio, anular y pulgar.
5	Acción	mover GDL(2)	Flexión-Extensión	índice, medio, anular y pulgar.

2. Desarrollo de los diagramas de interacción

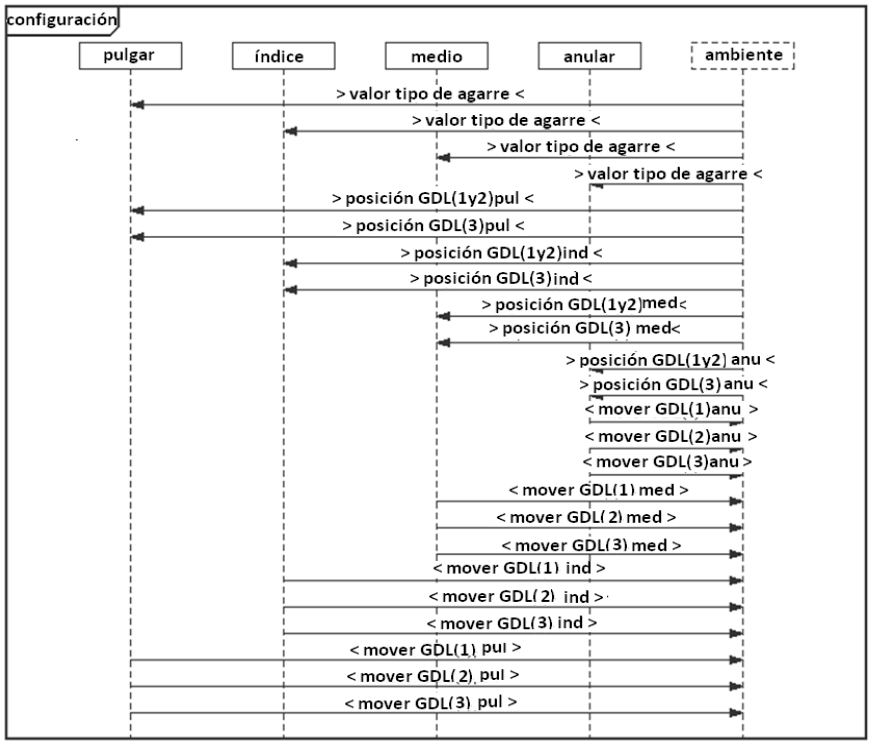


Figura B.2.3 Diagrama de interacción de acuerdo al escenario de configuración.

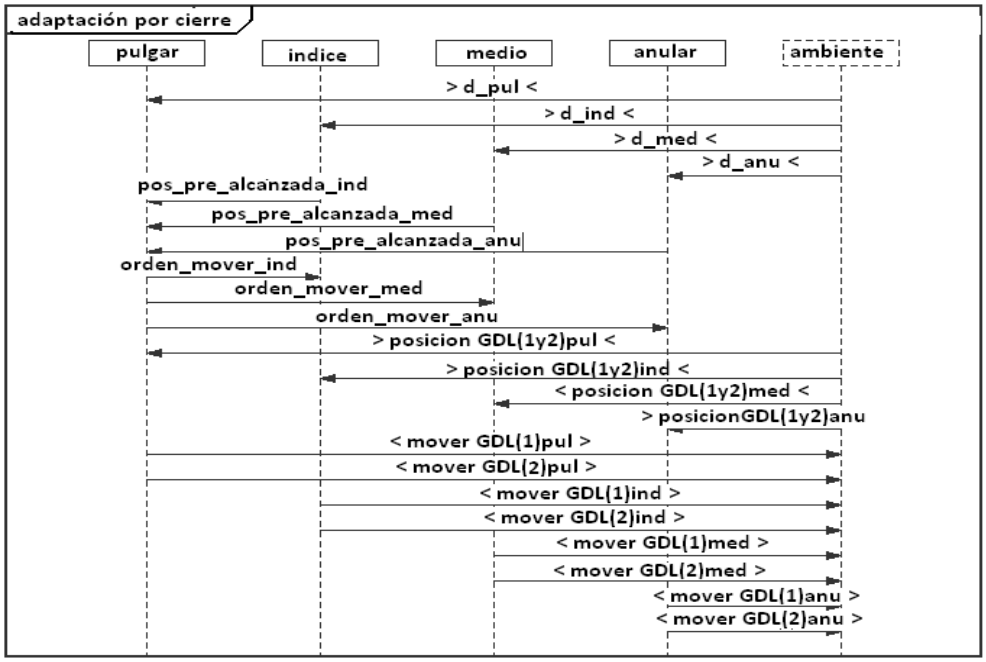


Figura B.2.4. Diagrama de interacción de acuerdo al escenario de adaptación por cierre.

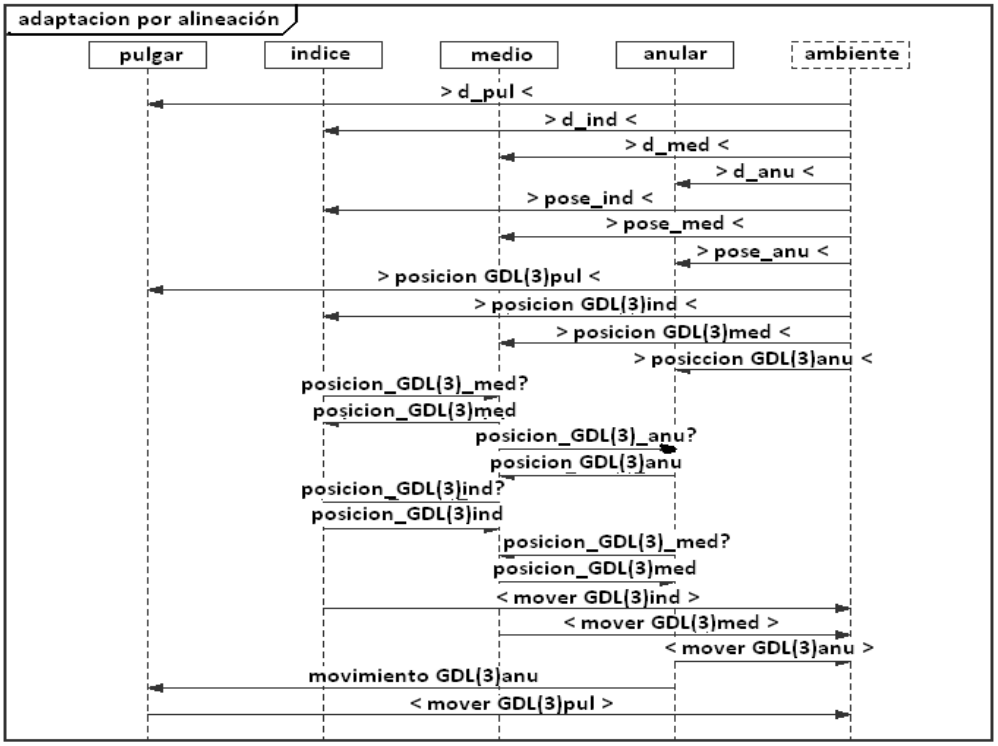


Figura B.2.51. Diagrama de interacción de acuerdo con el escenario de adaptación por alineación.

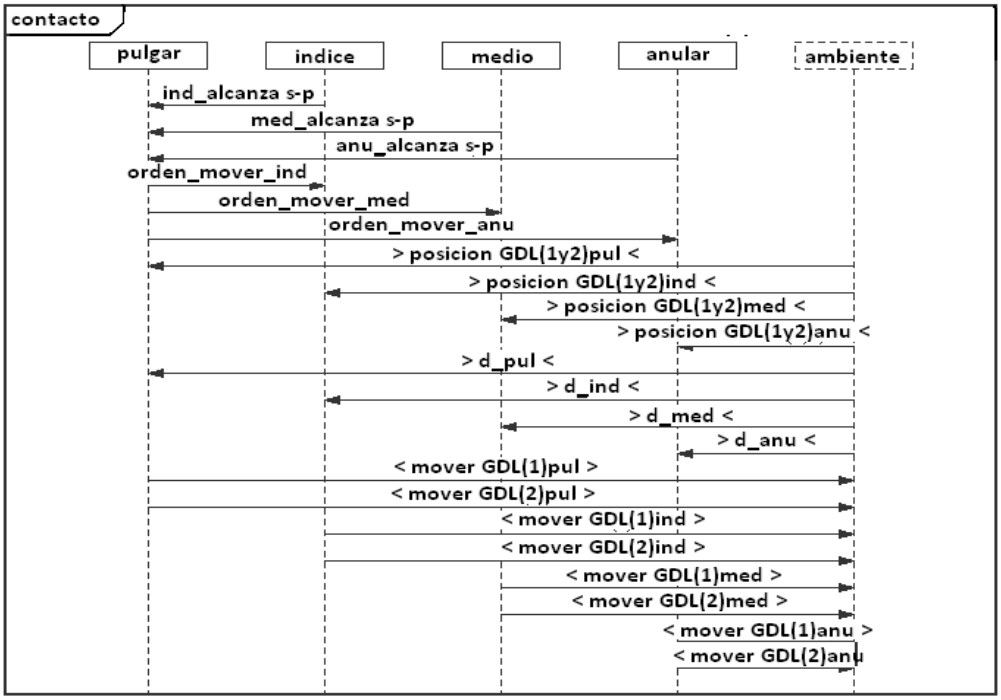


Figura B.2.6. Diagrama de interacción de acuerdo con el escenario de contacto.

3. Descriptores de los protocolos

- **Protocolo para la Flexión**

Nombre: Flexión
Descripción: Interacción entre los agentes en todo el proceso que involucra el movimiento de flexión de los dedos.
Mensajes incluidos: pos_pre_alcanzada_ind; pos_pre_alcanzada_med; pos_pre_alcanzada_anu; orden_mover_ind; orden_mover_med; orden_mover_anu; ind_alcanza s-p; med_alcanza s-p; anu_alcanza s-p
Escenarios: *Adaptación por cierre; Contacto.*
Agentes: Todos.
Notas:

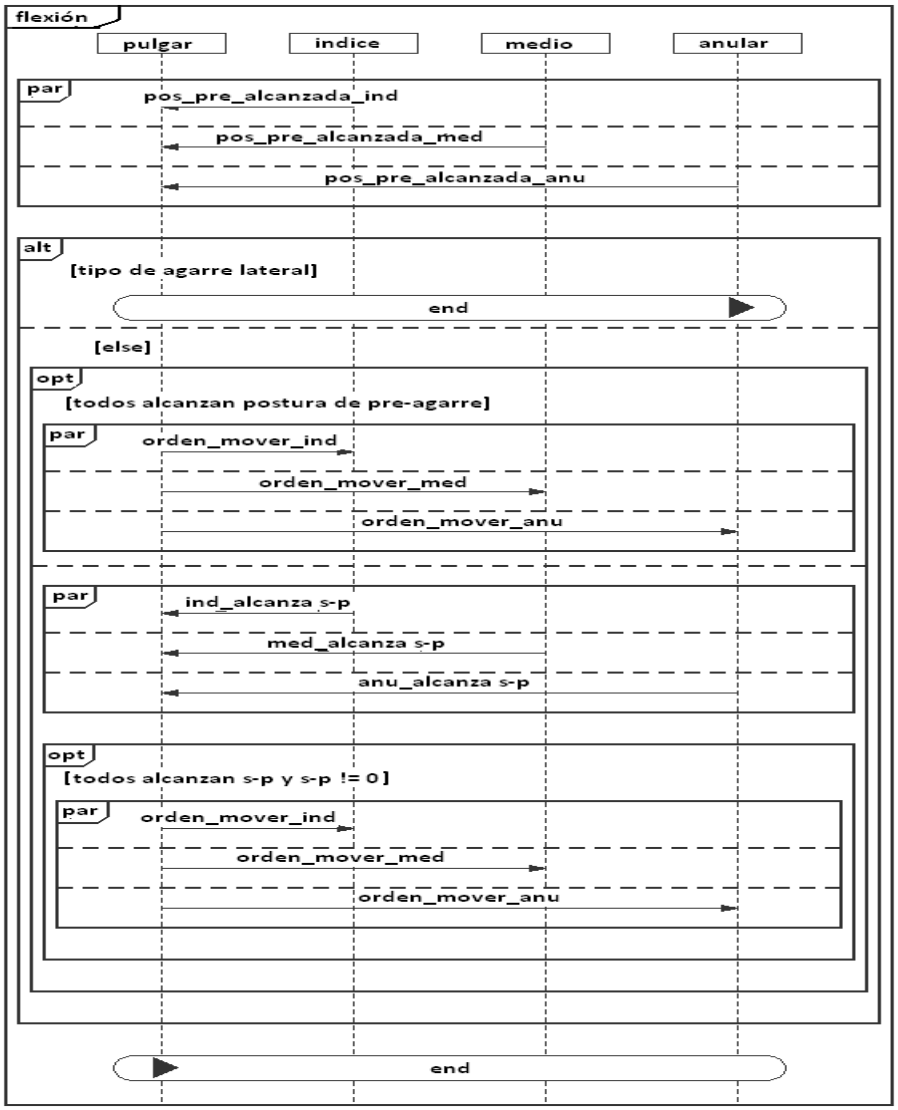


Figura B.2.7 Protocolo para la flexión.

- **Protocolo para la aducción-abducción**

  Nombre: ad/abducción.
  Descripción: Interacción entre los agentes en todo el proceso que involucra los movimientos de aducción-abducción de los dedos.
  Mensajes incluidos: posición_GDL(3)_med?; posición_GDL(3)_med; posición_GDL(3)_ind?; posición_GDL(3)_ind, posición_GDL(3)_anu?, posición_GDL(3)_anu;
  Escenarios: *Adaptación por alineación.*
  Agentes: Todos.
  Notas: los agentes solo reciben información sobre GDL(3), no de todas las posiciones articulares del dedo.

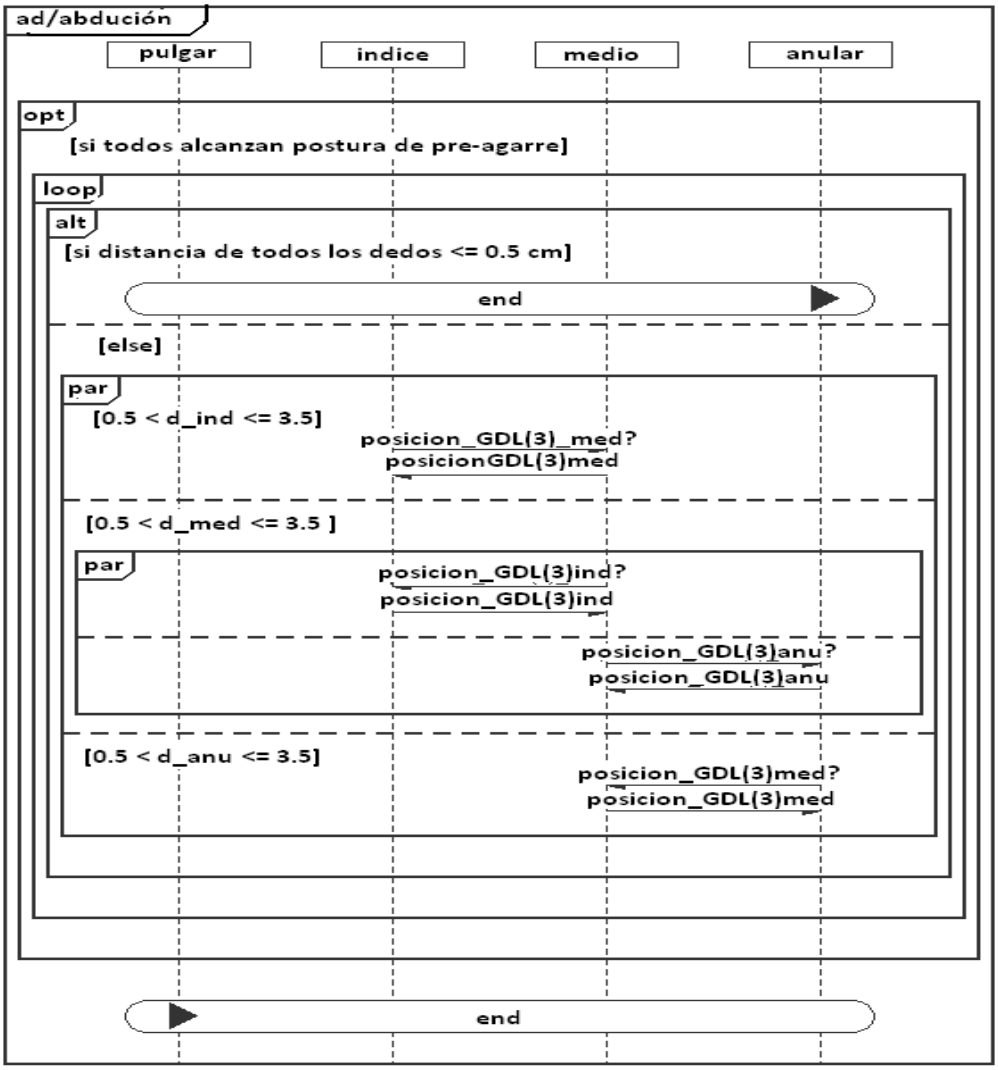


Figura B.2.8 Protocolo para la aducción-abducción.

B.2.4 Desarrollo De Los Descriptores De Los Mensajes Entre Agentes

- **Postura de pre-agarre alcanzada por parte del agente medio**

  Nombre: pos_pre_alcanzada_med
  Descripción: En este mensaje se indica que la postura de pre-agarre del agente medio ya ha sido alcanzada.
  Del agente: medio
  Para el agente: pulgar
  Propósito: indicar que la postura de pre-agarre del agente medio ya ha sido alcanzada.
  Información llevada: distancia hasta el objeto desde la punta o desde la falange del dedo medio. Si esta información esta contenida en el mensaje es por que la distancia del dedo al objeto es menor que el set-point por defecto (1 cm)

- **Orden de movimiento de flexión del agente medio**

  Nombre: orden_mover_med
  Descripción: En este mensaje se indica que el dedo medio puede seguir realizando sus movimientos de flexión.
  Del agente: pulgar
  Para el agente: medio
  Propósito: activar el movimiento de flexión del dedo medio.
  Información llevada: set-point actualizado por parte del agente pulgar. Este set-point es la distancia menor de todas las distancias enviadas en los mensajes pos_pre_alcanzada de cada agente.

- **El agente medio alcanza el set-point**

  Nombre: med_alcanza s-p
  Descripción: En este mensaje se indica que el agente medio alcanzo el set-point.
  Del agente: medio
  Para el agente: pulgar
  Propósito: indicar al agente pulgar que la distancia hasta el objeto es igual a la distancia actualizada o a la distancia establecida por defecto, es decir, igual al set-point.
  Información llevada: ninguna, solo indica.

- **Posición actual de GDL(3) del agente medio?**

  Nombre: posición_GDL(3)_med?
  Descripción: En este mensaje se solicita la posición actual de GDL(3) del agente medio.
  Desde los agentes: índice, anular
  Al agente: medio
  Propósito: que se le envíe al agente que lo solicite la posición de GDL(3) del agente medio.

Información llevada: ninguna, solo es una solicitud.

- **Posición actual de GDL(3) del agente medio**

  Nombre: posición_GDL(3)_med
  Descripción: mensaje en el cual se envía la posición actual de GDL(3) del agente medio.
  Del agente: medio
  Para los agentes: anular, índice (según el agente que solicite)
  Propósito: enviar la posición actual de GDL(3) del agente medio.
  Información llevada valor actual de GDL(3) del agente medio.

- **Postura de pre-agarre alcanzada por parte del agente índice**

  Nombre: pos_pre_alcanzada_ind
  Descripción: En este mensaje se indica que la postura de pre-agarre del agente índice ya ha sido alcanzada.
  Del agente: índice
  Para el agente: pulgar
  Propósito: indicar que la postura de pre-agarre del agente índice ya ha sido alcanzada.
  Información llevada: distancia hasta el objeto desde la punta o desde la falange del dedo índice. Si esta información esta contenida en el mensaje es por que la distancia del dedo al objeto es menor que el set-point por defecto (1 cm)

- **Orden de movimiento de flexión del agente índice**

  Nombre: orden_mover_ind
  Descripción: En este mensaje se indica que el dedo índice puede seguir realizando sus movimientos de flexión.
  Del agente: pulgar
  Para el agente: índice
  Propósito: activar el movimiento de flexión del dedo índice.
  Información llevada: set-point actualizado por parte del agente pulgar. Este set-point es la distancia menor de todas las distancias enviadas en los mensajes pos_pre_alcanzada de cada agente.

- **El agente índice alcanza el set-point**

  Nombre: ind_alcanza s-p
  Descripción: En este mensaje se indica que el agente índice alcanzo el set-point.
  Del agente: índice
  Para el agente: pulgar
  Propósito: indicar al agente pulgar que la distancia hasta el objeto es igual a la distancia actualizada o a la distancia establecida por defecto, es decir, igual al set-point.
  Información llevada: ninguna, solo indica.

- **Posición actual de GDL(3) del agente índice?**

  Nombre: posición_GDL(3)_ind?
  Descripción: En este mensaje se solicita la posición actual de GDL(3) del agente índice.
  Desde el agente: medio
  Al agente: índice
  Propósito: que se le envíe al agente que lo solicite la posición de GDL(3) del agente índice.
  Información llevada: ninguna, solo es una solicitud.

- **Posición actual de GDL(3) del agente índice**

  Nombre: posición_GDL(3)_ind
  Descripción: mensaje en el cual se envía la posición actual de GDL(3) del agente índice.
  Del agente: índice
  Para el agente: medio
  Propósito: enviar la posición actual de GDL(3) del agente índice.
  Información llevada valor actual de GDL(3) del agente índice.

- **Postura de pre-agarre alcanzada por parte del agente anular**

  Nombre: pos_pre_alcanzada_anu
  Descripción: En este mensaje se indica que la postura de pre-agarre del agente anular ya ha sido alcanzada.
  Del agente: anular
  Para el agente: pulgar
  Propósito: indicar que la postura de pre-agarre del agente anular ya ha sido alcanzada.
  Información llevada: distancia hasta el objeto desde la punta o desde la falange del dedo índice. Si esta información esta contenida en el mensaje es por que la distancia del dedo al objeto es menor que el set-point por defecto (1 cm)

- **Orden de movimiento de flexión del agente anular**

  Nombre: orden_mover_anu
  Descripción: En este mensaje se indica que el dedo anular puede seguir realizando sus movimientos de flexión.
  Del agente: pulgar
  Para el agente: anular
  Propósito: activar el movimiento de flexión del dedo anular.
  Información llevada: set-point actualizado por parte del agente pulgar. Este set-point es la distancia menor de todas las distancias enviadas en los mensajes pos_pre_alcanzada de cada agente.

- **El agente anular alcanza el set-point**

  Nombre: anu_alcanza s-p
  Descripción: En este mensaje se indica que el agente anular alcanzo el set-point.
  Del agente: anular
  Para el agente: pulgar
  Propósito: indicar al agente pulgar que la distancia hasta el objeto es igual a la distancia actualizada o a la distancia establecida por defecto, es decir, igual al set-point.
  Información llevada: ninguna, solo indica.

- **Posición actual de GDL(3) del agente anular?**

  Nombre: posición_GDL(3)_anu?
  Descripción: En este mensaje se solicita la posición actual de GDL(3) del agente anular.
  Desde el agente: medio
  Al agente: anular
  Propósito: que se le envíe al agente que lo solicite la posición de GDL(3) del agente anular.
  Información llevada: ninguna, solo es una solicitud.

- **Posición actual de GDL(3) del agente anular**

  Nombre: posición_GDL(3)_anu
  Descripción: mensaje en el cual se envía la posición actual de GDL(3) del agente anular.
  Del agente: anular
  Para el agente: medio
  Propósito: enviar la posición actual de GDL(3) del agente anular.
  Información llevada valor actual de GDL(3) del agente anular.

B.3 DISEÑO DETALLADO

Durante el diseño de la arquitectura, se desarrollaron los protocolos y el diagrama de visión general del sistema. En el diseño detallado, se toman estos artefactos para generar el diagrama general del agente en el cual se deben identificar las capacidades las cuales son el resultado de agrupar funcionalidades que cumplan objetivos comunes. Al interior de las capacidades pueden confinarse otras capacidades o planes que suministran los detalles de cómo el agente reacciona ante eventos para lograr sus objetivos. Además, del diagrama general del agente, en esta fase se deben detallar los procesos que se deben realizar los agentes para lograr sus objetivos de diseño. Estos procesos se realizan a partir de los escenarios, los diagramas de interacción y los protocolos.

En esta fase solo se desarollan los descriptores de los mensajes, procesos, capacidades y planes del agente índice. Los descriptores que corresponden a los otros agentes son

básicamente los mismos a los del agente índice y por eso no son desarollados en este trabajo. La única excepción es el agente pulgar quien en su interior contiene algunas entidades nuevas. Para estas entidades se desarrollan sus respectivos protocolos.

B.3.1 Agente Indice

1. Diagrama de visión general

El diagrama de visión general del agente muestra las relaciones que existen entre las capacidades. Estas relaciones se reflejan por medio de enlaces que describen mensajes, percepciones, acciones y bases de datos. Para desarrollar el diagrama de visión general del agente se usa la herramienta *PDT*. En esta herramienta el usuario simplemente da doble clic en el agente e inmediatamente se despliega una pantalla que muestra el interior del agente. En su interior el agente contiene entidades que habían sido definidas anteriormente y que representan el mundo externo para el agente. Estas entidades externas y las creadas al interior del agente son conectadas por el usuario de manera interactiva en la pantalla generada por la herramienta *PDT*. Las acciones y las percepciones son conectadas a las capacidades de igual forma como se hizo para las funcionalidades. Lo anterior da como resultado el diagrama general del agente
En la simbología determinada por la herramienta *PDT* los mensajes son de color morado; los planes de color azul; las acciones de color verde; las *BD* de color amarillo; las capacidades de color naranja. Las entidades más descoloridas ya habían sido creadas y representan la interacción del agente con su mundo externo.

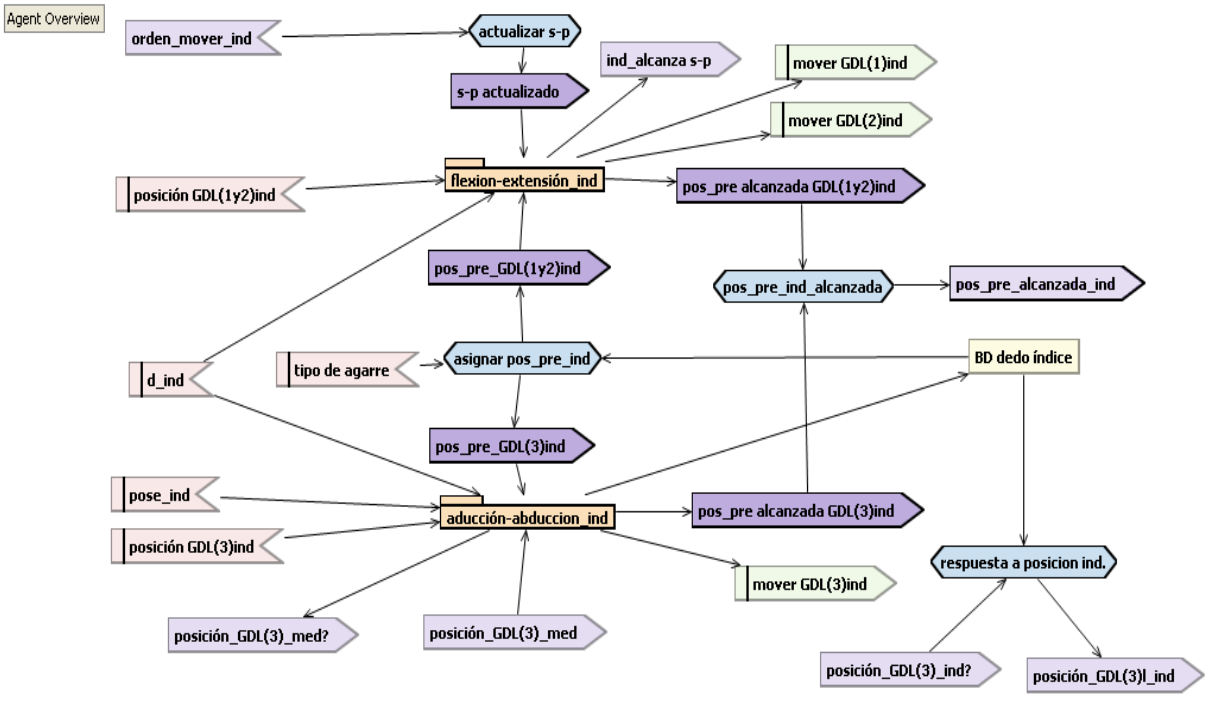


Figura B.3.1 Diagrama de visión general del agente índice.

2. Capacidades

En el diseño de la arquitectura se determinó que funcionalidades deberían de pertenecer a cada agente. La lista de estas funcionalidades es un punto de partida para definir las capacidades. Debido a la alta cohesión quetiene cada funcionalidad, en este trabajo se optó por hacer corresponder cada una de ellas con una capacidad. Por ejemplo, las funcionalidades del agente índice: flexión-extensión y aducción-abducción resultan en dos capacidades con el mismo nombre. En la tabla 9 se muestran las capacidades resultantes para cada agente.

Tabla 9. Capacidades de cada agente con sus respectivos objetivos

AGENTE	CAPACIDADES	OBJETIVOS
pulgar	flexión-extensión_pul	mover GDL(1y2)pul a s-p ; mover GDL(1y2)pul a contacto; mover GDL(1y2)pul a pos_pre; obtener GDL(1y2)pul.
	oposición_pul	mover GDL(3)pul a opo; mover GDL(3)pul a pos_pre; obtener GDL(3)pul.
medio	flexión-extensión_med	mover GDL(1y2)med a s-p ; mover GDL(1y2)med a contacto; mover GDL(1y2)med a pos_pre; obtener GDL(1y2)med
	aducción-abducción_med	mover GDL(3)med a II; mover GDL(3)med a pos_pre; obtener GDL(3)med.
índice	flexión-extensión_ind	mover GDL(1y2)ind a s-p; mover GDL(1y2)ind a contacto; mover GDL(1y2)ind a pos_pre; obtener GDL(1y2)ind.
	aducción-abducción_ind	mover GDL(3)ind a II; mover GDL(3)ind a pos_pre; obtener GDL(3)ind.
anular	flexión-extensión_anu	mover GDL(1y2)anu a s-p; mover GDL(1y2)anu a contacto; mover GDL(1y2)anu a pos_pre; obtener GDL(1y2)anu.
	aducción-abducción_anu	mover GDL(3)anu a II; mover GDL(3)anu a pos_pre; obtener GDL(3)anu.

2.1 Capacidad Flexión_Extensión_Ind

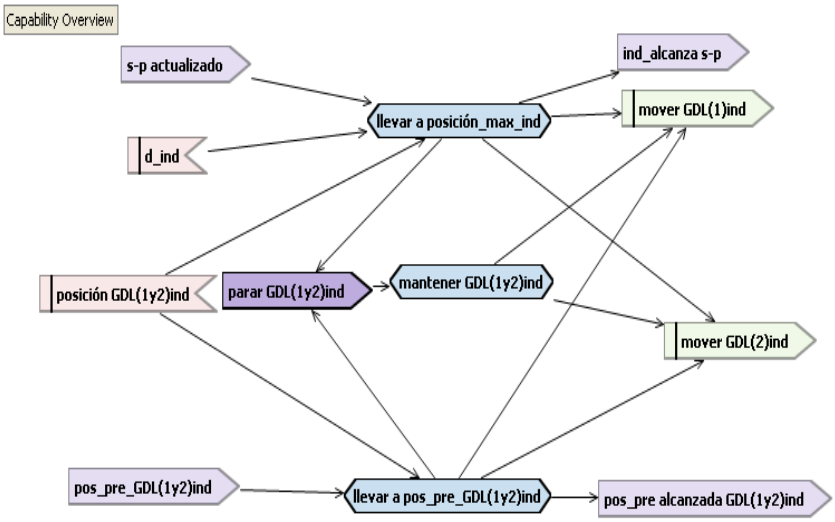


Figura B.3.2. Diagrama de visión general de la capacidad de flexión_extensión_ind

Nombre: flexión-extensión_ind.

Descripción: Esta capacidad calcula la nueva posición de GDL(1y2), es decir, de los grados de libertad que realizan los movimientos de flexión-extensión del dedo índice asegurando que la distancia de la punta del dedo al objeto sea la misma a la de los otros dedos.
Objetivos: mover GDL(1y2)ind a s-p; mover GDL(1y2)ind a contacto; mover GDL(1y2)ind a pos_pre; obtener GDL(1y2)ind.
Procesos: *configuración y flexión_ind*
Protocolos: *flexión.*
Mensajes de salida: ind_alcanza s-p; pos_pre alcanzada GDL(1y2)ind;
Mensajes de entrada: s-p actualizado pos_pre_GDL(1y2)ind.
Mensajes internos: parar GDL(1y2)ind
Percepciones: d_ind, posición GDL(1y2)ind.
Acciones: mover GDL(1)ind; mover GDL(2)ind
Capacidades incluidas: ninguna
Datos Usados (Importados): BD dedo índice.
Datos Producidos (Exportados):ninguno.
Datos Internos: ninguno.
Capacidades anidadas: ninguna.
Planes incluidos: llevar a posición max_ind; mantener GDL(1y2)ind; llevar a pos_pre_GDL(1y2)ind.
Notas:

2.2 Capacidad Aduccion_Abduccion_Ind

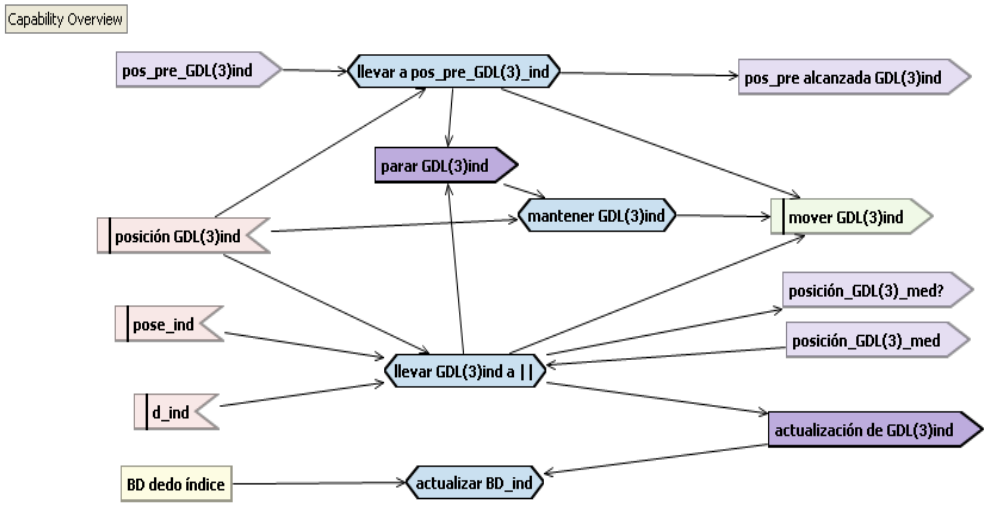


Figura B.3.3. Diagrama de visión general de la capacidad de aducción_abduccion_ind

Nombre: aducción-abducción_ind.

Descripción: Esta capacidad calcula la nueva posición de GDL(3), es decir,la posición del grado de libertad encargado de los movimientos de aducción y abducción del dedo índice, colocando la punta del dedo lo más paralela posible a la pose local del objeto y evitando que el dedo colisione con otros dedos.
Objetivos: mover GDL(3)ind a II; mover GDL(3)ind a pos_pre; obtener GDL(3)ind.
Procesos: configuración yaducción_abducción_ind.
Protocolos: ab/aducción.
Mensajes de salida: pos_pre alcanzada GDL(3)ind; posición GDL(3)med?;
Mensajes de entrada:pos_pre_GDL(3)ind;
Mensajes internos: parar GDL(3)ind.
Percepciones: pose_ind, posición GDL(3)ind; d_ind
Acciones: mover GDL(3)ind.
Capacidades incluidas: ninguna.
Datos Usados (Importados): BD dedo índice.
Datos Producidos (Exportados): BD dedo índice.
Datos Internos: ninguno.
Capacidades anidadas: ninguna.
Planes incluidos: llevar a pos pos_pre_GDL(3)ind; mantener GDL(3)ind; llevar GDL(3)ind a ||; actualizar BD_ind
Notas:

3. Procesos

Durante la etapa de especificación del sistema, se especificaron los escenarios, los cuales son procesos de alto nivel. Estos procesos fueron refinados y especificados completamente usando los diagramas de interacción y los protocolos en la etapa del diseño de la arquitectura. En el diseño detallado, también se necesita de la ayuda de algún mecanismo para especificar más detalladamente este proceso. El mecanismo usado por la metodología Prometheuses una variante de los diagramas de actividad UML. Estos diagramas se desarrollan a partir de los protocolos, los escenarios y los objetivos de cada agente y describen de manera aún más detallada la interacción entre los agentes por medio de mensajes.

En el fase del diseño de la arquitectura de identificaron básicamente dos protocolos. En la fase de diseño detallado se tomaran en cuenta estos protocolos para hacer una especificación de éstos mas detallada. En la figura B.3.4 se ilustran algunas notaciones para la especificación de los procesos.

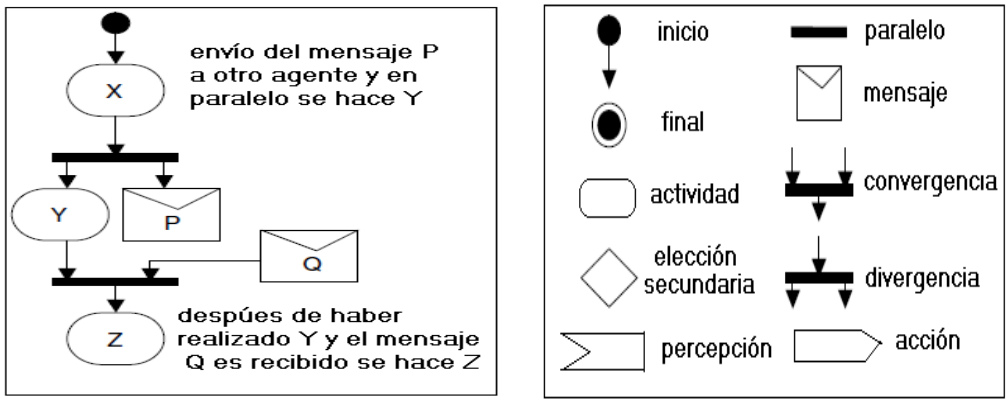


Figura B.3.4. Ilustración de algunas notaciones para la especificación de los procesos. [5]

3.1 Proceso de configuración del agente índice

Nombre: configuración_ind.
Descripción: según el tipo de agarre, en este proceso lo que se logra es llevar las articulaciones del dedo índice a su posición de pre-agarre.
Actividades: llevar a pos_pre GDL(1y2)ind; llevar a pos_pre GDL(3)ind; mantener GDL(1y2)ind; mantener GDL(3)ind.
Accionadores: mensajes: pos_pre_GDL(1y2)ind; pos_pre_GDL(3)ind.
Mensajes: : pos_pre_GDL(1y2)ind; pos_pre_GDL(3)ind; pos_pre_alcanzada_GDL(1y2)ind; pos_pre_alcanzada_GDL(3)ind.
Protocolos: ninguno

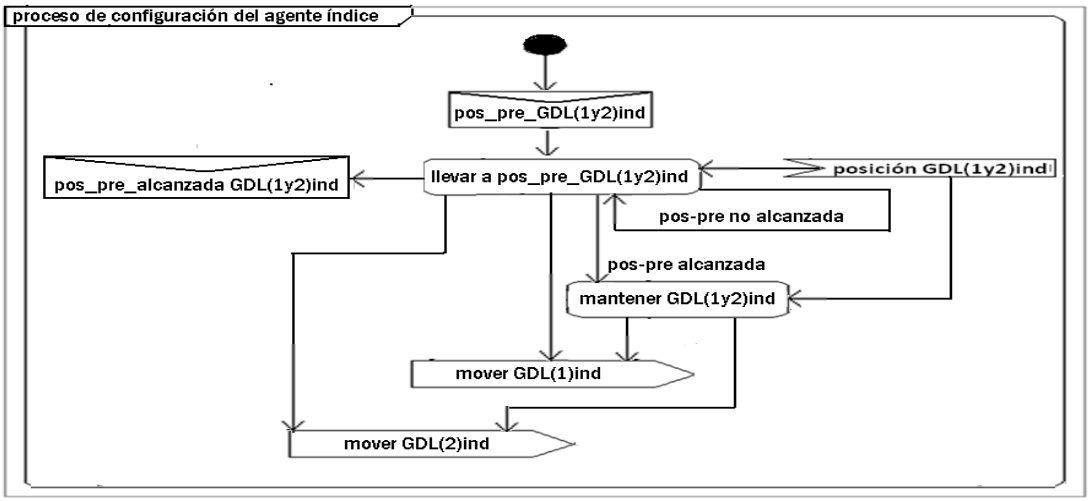


Figura B.3.5. Diagrama de proceso de configuración del agente índice.

3.2 Proceso de flexión del agente indice

De acuerdo al mensaje *orden_mover_ind* la actividad *actualizar s-p* es la encargada de actualizar el s-p o distancia a la cual debe llegar el agente con relación a la superficie del objeto. Si en el mensaje *orden_mover_ind*existe alguna información de distancia, entonces esa distancia corresponde al nuevo s-p que debe alcanzar el agente. Por lo contario si en el mensaje no existe ninguna información, entonces el s-p es el establecido en 1cm. Una vesactualizado el s-p, se realiza la actividad *llevar a posición_max.* Esta actividad es la encargada de llevar los grados de libertad encargados de la flexión-extensión del agente índice a sus posiciones máximas (mover GDL(1) y GDL(2)). Esta actividad se realiza hasta que la distancia detectada por el agente sea aproximadamente igual al s-p, una ves ocurrido esto se deja de ejecutar la actividad *llevar a posición_max*y se envía el mensaje *ind_alcanza_s-p* y se da inicio a la actividad *mantener GDL(1y2).* En esta actividad los grados de libertad encargados de la flexión-extensión del agente índice (GDL(1y2)) se mantienen en la última posición que tenían antes de que se dar inicio a esta última actividad. Según el protocolo para la flexión la orden de movimiento se envía dos veces, por lo tanto cuando la actividad *actualizar s-p* recibe por segunda ves el mensaje *orden_mover_ind*, el set-point es actualizado a cero, es decir, la distancia de contacto con el objeto.

Nombre: flexión_índice

Descripción: En este proceso lo que se logra es que el dedo índice toque el objeto al mismo tiempo que los otros dedos mientras realiza su movimiento de flexión.
Actividades: actualizar s-p, llevar a posición_máx, mantener GDL(1y2).
Accionadores: orden_mover_ind.
Mensajes: ind_alcanza_s-p. (Enviado al agente pulgar).
Protocolos: flexión.

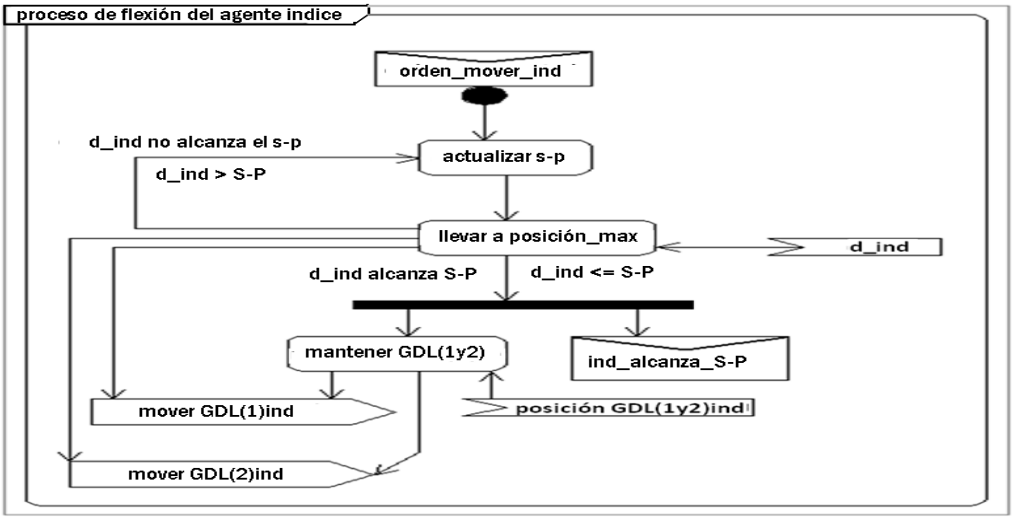


Figura B.3.6 Diagrama de proceso para la flexión-extensión del agente índice.

3.3 Proceso de aducción-abducción del agente indice.

Nombre: aducción-abducción_Ind.
Descripción: En este proceso lo que se logra es que la superficie de la punta del dedo índice quede lo mas paralela posible a la superficie local del objeto y de evitar colision con el agente cercano.
Actividades: mantener GDL(3)ind; llevar GDL(3)ind a ||.
Accionadores: pos_pre_alcanzada GDL(3)ind
Mensajes: pos_pre_alcanzada GDL(3)ind; posición GDL(3)med?; posición GDL(3)med.
Protocolos: ad/abducción.

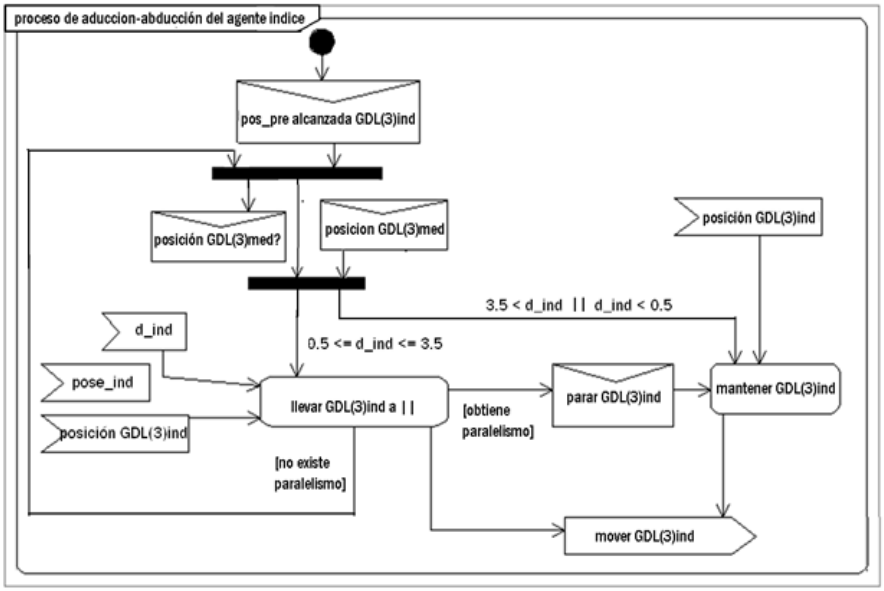


Figura B.3.7. Diagrama de proceso para la aducción-abducción del agente índice

4. Descriptores de mensajes internos del agente indice.

- **Set-point actualizado por parte del agante índice**

  Nombre: s-p actualizado
  Descripción: En este mensaje se indca el set-point actual que debe alcanzar el agente índice
  Distribución: plan *actualizad s-p* → capacidad *flexión-extensión_ind*
  Proposito: entregar el s-p actualizado.
  Información llevada:s-p actualizado
  Cobertura y superposición:cobertura completa, no hay superposición.

- **Postura de pre-agarre de los grados de libertad GDL(1) y GDL(2) del agente índice ha sido alcanzada.**

  Nombre: pos_pre alcanzada GDL(1y2)ind.
  Descripción: mensaje en el cual se avisa que GDL(1y2) han alcanzado su posición de pre-agarre.
  Distribución: capacidad: *flexión-extensión_ind* →plan: *pos_pre_ind_alcanzada*
  Proposito: avisar la llegada a la postura de pre-agarre.
  Información llevada: ninguna.
  Cobertura y superposición: cobertura completa, no hay superposición.

- **Postura de pre-agarre de grado de libertad GDL(3) del agente índice ha sido alcanzada.**

  Nombre: pos_pre alcanzada GDL(3)ind.
  Descripción: mensaje en el cual se avisa que GDL(3) han alcanzado su posición de pre-agarre.
  Distribución: capacidad: *aducción_abducción_ind* → plan: *pos_pre_ind_alcanzada*
  Proposito: avisar la llegada a la postura de pre-agarre.
  Información llevada: ninguna.
  Cobertura y superposición: cobertura completa, no hay superposición.

- **Postura de pre-agarre de los grados de libertad GDL(1) y GDL(2) del agente índice**
  Nombre: pos_pre_GDL(1y2)ind.
  Descripción: mensaje que avisa sobre la postura de pre-agarre que deben alcanzar GDL(1y2) del agente índice.
  Distribución: plan: *asiganar pos_pre*_ind →capacidad: *flexión-extensión_ind*
  Proposito: informar sobre las posiciones de pre-agarre de GDL(1y2) del agente índice.
  Información llevada: posiciones de pre-agarre de GDL(1y2) del agente índice.
  Cobertura y superposición: cobertura completa, no hay superposición.

- **Postura de pre-agarre de grado de libertad GDL(3) del agente índice ha sido alcanzada.**

  Nombre: pos_pre alcanzada GDL(3)ind.
  Descripción: mensaje en el cual se avisa que GDL(3) han alcanzado su posición de pre-agarre.
  Distribución: capacidad: *aducción_abducción_ind* → plan: *pos_pre_ind_alcanzada*
  Proposito: avisar la llegada a la postura de pre-agarre.
  Información llevada: ninguna.
  Cobertura y superposición: cobertura completa, no hay superposición.

- **Postura de pre-agarre del grado de libertad GDL(3) del agente índice**

  Nombre: pos_pre_GDL(3)ind.
  Descripción: mensaje que avisa sobre la postura de pre-agarre que deben alcanzar GDL(3) del agente índice.
  Distribución: plan: *asiganar pos_pre*_ind → capacidad: *aducción_abducción_ind*
  Proposito: informar sobre las posiciones de pre-agarre de GDL(3) del agente índice.
  Información llevada: posiciones de pre-agarre de GDL(3) del agente índice.
  Cobertura y superposición: cobertura completa, no hay superposición.

- **Parar el movimiento de los grados de libertad GDL(1y2) del agente índice.**

  Nombre: parar GDL(1y2)ind
  Descripción: mensaje usado como aviso para detener los movimientos correspondientes a GDL(1y2) en las posiciones actuales.
  Distribución: planes: *llevar a pos_pre_GDL(1y2)ind, llevar a posisión_max_ind* → plan: *mantener GDL(1y2)ind.*
  Proposito: detener los movimentos de GDL(1y2).
  Información llevada: posición en la que se quiere se detengan GDL(1y2). (posición actual)
  Cobertura y superposición: cobertura completa, no hay superposición.

- **Parar el movimiento de grado de libertad GDL(3) del agente índice.**

  Nombre: parar GDL(3)ind
  Descripción: mensaje usado como aviso para detener los movimientos correspondientes a GDL(3) en las posiciones actuales.
  Distribución: planes: *llevar a pos_pre_GDL(3)ind, llevar a posisión_max_ind* → plan: *mantener GDL(3)ind.*
  Proposito: detener los movimentos de GDL(3).
  Información llevada: posición en la que se quiere se detengan GDL(3). (Posición actual)
  Cobertura y superposición: cobertura completa, no hay superposición.

- **Actualización de GDL(3) del agente índice.**

  Nombre: actualización de GDL(3)ind.
  Descripción: cada que se realizan movimientos de GDL(3) del agente índice este valor debe ser actualizado en la base de datos.
  Distribución: plan: *llevar GDL(3)ind a* ||→ plan: *actualizar BD_ind.*
  Proposito: actualizar GDL(3) en la *BD dedo indice*
  Información llevada: valor actual de GDL(3)
  Cobertura y superposición: cobertura completa, no hay superposición.

5. Descriptores de los planes internos del agente indice

- **Asignar los valores de los gardos de libertad para formar la postura de pre-agarre del agente índice.**

  Nombre: asignar_pos_pre_ind
  Descripción: Asignar a las capacidades los valores para formar la postura de pre-agarre
  Accionador: llegada del valor del tipo de agarre.
  Contexto: si previamente la posición que determia la postura de pre-agarre no ha sido asignada.
  Mensajes de entrada: ninguno.
  Mensaje de salida: pos_pre_GDL(1y2)ind; pos_pre_GDL(3)ind.
  Datos usados:BD dedo índice.
  Datos producidos:ninguno.
  Falla:no poder acceder a la base de datos.
  Recuperacion de la falla:volver a percibir el valor que identifica el tipo de agarre. si persiste la falla entonces la posición actual es tomada como la postura de pre-agarre.
  Procedimiento:

1- Acceder a la base de datos para extraer las posiciones para formar la postura de pre-agarre, de acuerdo al valor del tipo de agarre.
2- Enviar las posiciones correspondientes a cada una de las capaciadades.

- **Actualizar el valor del set-point del agente índice.**

  Nombre: actualizar s-p.
  Descripción: según, si en la orden de movimiento se contiene un valor de distancia, este plan actualiza el nuevo s-p.
  Accionador: llegada del mensaje *orden_mover_ind.*
  Contexto: si previamente no ha sido actualizado el set-point
  Mensajes de entrada: *orden_mover_ind*
  Mensaje de salida:*s-p actualizado*
  Datos usados: ninguno.
  Datos producidos: ninguno.

Falla: s-p que no se encuentre dentro de los valores esperados.
Recuperacion de la falla: tomar el s-p como cero.
Procedimiento:

1. Detallar la información al interior del mensaje *orden_mover_ind.*
2. Si existe información de distanci entonces, el s-p es actualizado a dicha distancia. De lo contrario se deja el s-p por defecto (1 cm)
3. Enviar mensaje de salida.

- **Posición de pre-agarre del agente índice alcanzada.**

Nombre: pos_pre_ind_alcanzada
Descripción:verficador de que la postura de pre-agarre del agente índice ha sido alcanzada.
Accionadores: *pos_pre alcanzada GDL(1y2)ind; pos_pre alcanzada GDL(3)ind.*
Contexto: cuando la postura de pre-agarre del agente índice ha sido alcanzada
Mensajes de entrada: *pos_pre alcanzada GDL(1y2)ind; pos_pre alcanzada GDL(3)ind*
Mensaje de salida:*pos_pre_alcanzada_ind.*
Datos usados: ninguno.
Datos producidos: ninguno.
Falla: tiempo prolongado de espera de los dos o de un mensaje de entrada.
Recuperacion de la falla: no hay solo esperar. Ya que no se pueden iniciar otros movimientos si antes no se ha enviado el mensaje que avisa sobre la allegada a la postura de pre-agarre.
Procedimiento:

1. Cuando llegan los dos mensajes de entrada, simplemente se envía el mensaje de salida.

- **Respuesta a las petiones de posición de GDL(3) del agente índice.**

Nombre: respuesta a posición ind
Descripción: recepciona las peticiones de posición de GDL(3) del agente índice y envía esta posición en un mensaje.
Accionadores: *posición_GDL(3)_ind?.*
Contexto: durante el proceso de aducción-abduccíon del agente medio.
Mensajes de entrada: *posición_GDL(3)_ind?.*
Mensaje de salida:*posición_GDL(3)_ind.*
Datos usados: BD dedo índice.
Datos producidos: ninguno.
Falla: posición no encontrada en la base de datos.
Recuperacion de la falla: nada. Simplemente en el mesaje se avisa sobre la ocurrencia del suceso.
Procedimiento:

1. Recepcion del mensaje de entrada.
2. Descargar de la base de datos la posición de GDL(3) y enviárla en el mensaje de salida.

- **Llevar a posición máxima los grados de libertad del agante índice**

Nombre: llevar a posición_max_ind
Descripción: este plan se encarga de flexionar el dedo índice hasta alcanzar sus limites articulares.
Accionadores: *s-p actualizado.*
Contexto: siempre y cuando no se hayan alcanzado los limites articulares o si la distancia (d_ind) es diferente al s-p.
Mensajes de entrada: *s-p actualizado.*
Mensaje de salida:*ind_alcanza s-p*
Datos usados: ninguno.
Datos producidos: ninguno.
Falla: si al detenrse el movimiento de flexión la distancia detectada es mucho menor al s-p.
Recuperacion de la falla: realizar un movimiento de extensión, hasta lograr que la distancia sea aproximadamente igual al s-p.
Procedimiento:

1. Observar la información de distancia que contiene el mensaje de entrada.
2. Mover GDL(1y2) hasta el alcanzar esta distancia.
3. Enviar el mensaje de salida que detiene el movimiento de GDL(1y2).

- **Mantener en la posición actual GDL(1y2) del agente índice.**

Nombre: mantener GDL(1y2)ind.
Descripción: este plan se encarga de mantener en el estado actual GDL(1y2) del agente índice.
Accionadores: *parar GDL(1y2)ind.*
Contexto: si antes GDL(1y2) se encuentraban en movimiento
Mensajes de entrada: *parar GDL(1y2)ind.*
Mensaje de salida: *ninguno.*
Datos usados: ninguno.
Datos producidos: ninguno.
Falla: si no se logra detener el movimiento.
Recuperacion de la falla: reintentar.
Procedimiento:

1. Observar La información de la posición actual que contiene el mensaje de entrada.
2. Mantener GDL(1y2) en estaposicion.

- **Llevar a postura de pre-agarre GDL(1y2) del agente índice.**

Nombre: Llevar a pos_pre_GDL(1y2)ind.
Descripción: este plan se encarga de llevar a su postura de pre-agarre GDL(1y2)
Accionadores: *pos_pre_GDL(1y2)ind*
Contexto: si antes GDL(1y2) no han alcanzado la postura de pre-agarre.
Mensajes de entrada: *pos_pre_GDL(1y2)ind*
Mensaje de salida:*pos_pre alcanzada GDL(1y2)ind*

Datos usados: ninguno.
Datos producidos: ninguno.
Falla: ningúna información en el mensaje de entrada.
Recuperacion de la falla: mantener posiciones actuales y enviar el mensaje de salida.
Procedimiento:

1. Recpcionar el mensaje de entrada y mirar la posición de la postuar de pre-agarre en su interior.
2. Llevar GDL(1y2) hasta alcanzar la postura de pre-agarre y enviar el mensaje de salida.

- **Llevar a postura de pre-agarre GDL(3) del agente índice.**

Nombre: Llevar a pos_pre_GDL(3)ind.
Descripción: este plan se encarga de llevar a su postura de pre-agarre GDL(3)
Accionadores: *pos_pre_GDL(3)ind*
Contexto: si antes GDL(3) no han alcanzado la postura de pre-agarre.
Mensajes de entrada: *pos_pre_GDL(3)ind*
Mensaje de salida: *pos_pre alcanzada GDL(3)ind*
Datos usados: ninguno.
Datos producidos: ninguno.
Falla: ningúna información en el mensaje de entrada.
Recuperacion de la falla: mantener posiciones actuales y enviar el mensaje de salida.
Procedimiento:

1. Recpcionar el mensaje de entrada y mirar la posición de la postuar de pre-agarre en su interior.
2. llevar GDL(3) hasta alcanzar la postura de pre-agarre y enviar el mensaje de salida.

- **Mantener en la posición actual GDL(3) del agente índice.**

Nombre: mantener GDL(3)ind.
Descripción: este plan se encarga de mantener en el estado actual GDL(3) del agente índice.
Accionadores: *parar GDL(3)ind.*
Contexto: si antes GDL(3) se encuentraban en movimiento
Mensajes de entrada: *parar GDL(3)ind.*
Mensaje de salida:*ninguno.*
Datos usados: ninguno.
Datos producidos: ninguno.
Falla: si no se logra detener el movimiento.
Recuperacion de la falla: reintentar.
Procedimiento:

1. Observar La información de la posición actual que contiene el mensaje de entrada.
2. mantener GDL(3) en esta posicion.

- **Mover el grado de libertad GDL(3) del agente índice de tal manera que la punta del dedo representado por este agente quede lo mas paralela posible a la superficie local del objeto.**

  Nombre: Llevar GDL(3)ind a ||
  Descripción: este plan se encarga de llevar la punta del dedo inidice lo mas paralela posible a la superficie local del objeto.
  Accionadores: *pose_ind*
  Contexto: si la distancia hasta el objeto esta en el rango [0.5 cm<= d_ind <= 3.5 cm]
  Mensajes de entrada: *posición_GDL(3)_med.*
  Mensaje de salida:*posición_GDL(3)_med?; actualización de GDL(3)ind; parar GDL(3)ind.*
  Datos usados: ninguno.
  Datos producidos: ninguno.
  Falla: sobre paso de los límites articulares
  Recuperacion de la falla: volver a posición previa.
  Procedimiento:

1. Percibir la pose (orientación de la superficie local del objeto)
2. Solictar la posición de GDL(3) del agente medio.
3. Realizar movimientos de aducción o abducción según pose.
4. Al moverse enviar el mensaje con la posición actual de GDL(3) para que su valor sea actualizado en la base datos.
5. Al terminar de posicionar la punta el dedo índice lo mas paralela posible a la superficie local del objeto, eviar el mensaje *parar GDL(3)ind* para detenr los movimientos de aducción-abducción.

- **Actualizar el valor de GDL(3) en la BD dedo índice**

  Nombre: actualizar BD_ind.
  Descripción: este plan se encarga de actualizar el valor de GDL(3) del agente índice en la base de datos que le corresponde a este agente.
  Accionadores: *actualización de GDL(3)ind*
  Contexto: si antes no ha sido actualizada con ese valor la BD dedo índice.
  Mensajes de entrada: *actualización de GDL(3)ind.*
  Mensaje de salida: ninguno.
  Datos usados: *Bd dedo indice*
  Datos producidos: *Bd dedo indice*
  Falla: si no se logra acceder a la base de datos.
  Recuperacion de la falla: reintentar.
  Procedimiento:

1. Tomar el valor de GDL(3) que viene en el mensaje de entrada.
2. Actualizar GDL(3) en la base de datos a su nuevo valor.

B.3.2 Agente Pulgar

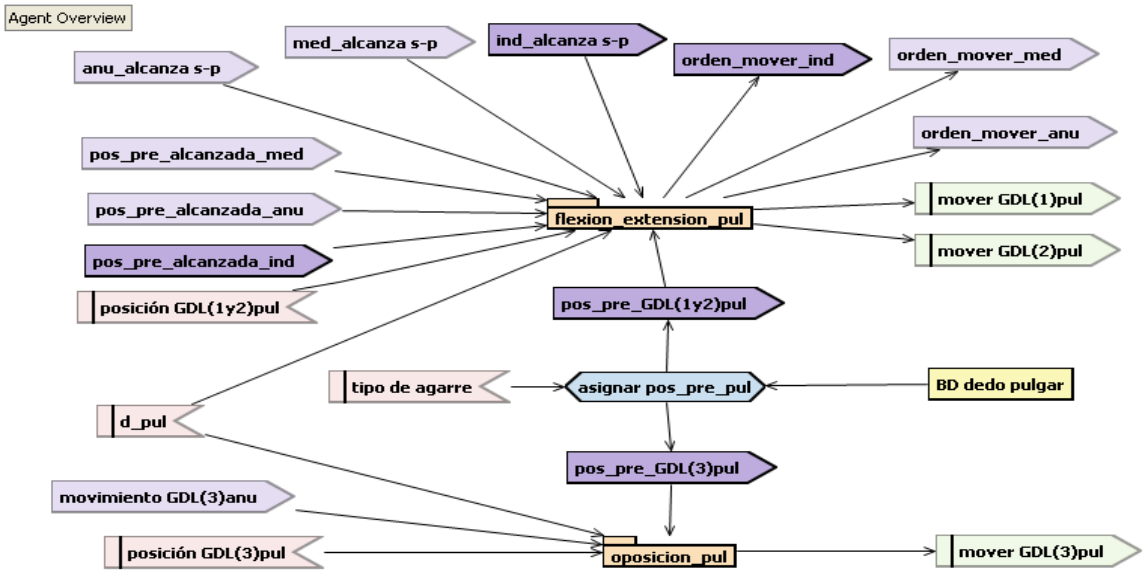


Figura B.3.8. Diagrama de visión general del agente pulgar

1. Capacidades

1.1 Capacidad flexión_extensión_pul

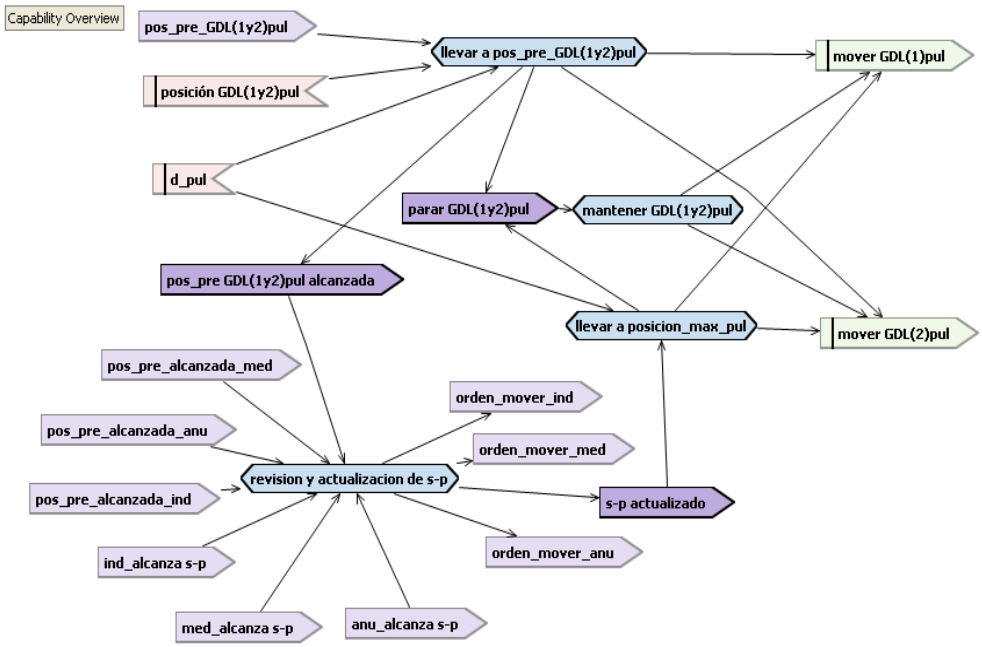


Figura B.3.9. Diagrama de visión general de la capacidad de flexión_extensión_pul

Nombre: flexión-extensión_pul

Descripción: Esta capacidad calcula la nueva posición de GDL(1y2), es decir, de los grados de libertad que realizan los movimientos de flexión-extensión del dedo pulgar. Además actualiza el set-point asegurando que todos los dedos toquen el objeto almismo tiempo.
Objetivos: mover GDL(1y2)pul a s-p ; mover GDL(1y2)pul a contacto; mover GDL(1y2)pul a pos_pre; obtener GDL(1y2)pul.
Procesos: *configuración_flexión_pul.*
Protocolos: *flexión.*
Mensajes de salida:;orden_mover_med; orden_mover_ind; orden_mover_anu.
Mensajes de entrada: ind_alcanza s-p; med_alcanza s-p; anu_alcanza s-p; pos_pre_alcanzada_med; pos_pre_alcanzada_anu; pos_pre_alcanzada_ind; pos_pre_GDL(1y2)pul.
Mensajes internos: parar GDL(1y2)ind; pos_pre_GDL(1y2) alcanzada; s-p actualizado.
Percepciones: d_pul; posición GDL(1y2)pul.
Acciones: mover GDL(1)pul; mover GDL(2)pul.
Capacidades incluidas: ninguna
Datos Usados (Importados): BD dedo pulgar.
Datos Producidos (Exportados): ninguno.
Datos Internos: ninguno.
Capacidades anidadas: ninguna.
Planes incluidos: llevar a posición max_pul; mantener GDL(1y2)pul; llevar a pos_pre_GDL(1y2)pul; revisión y actualización de s-p.
Notas:

1.2 Capacidad de oposición_pul

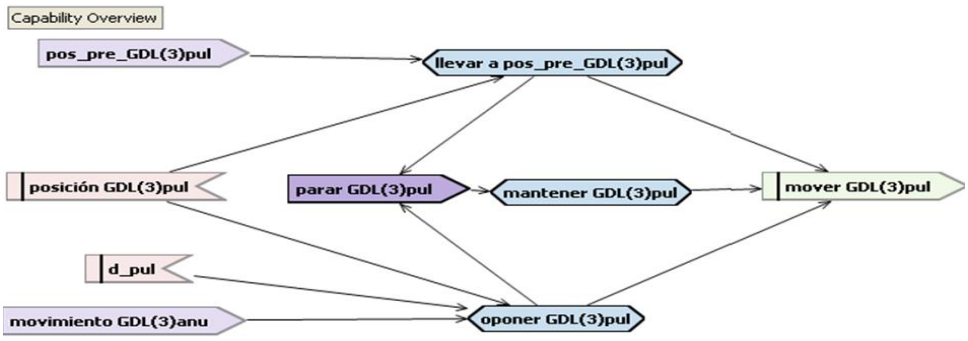


Figura B.3.10. Diagrama de visión general de la capacidad de oposicion_pul

Nombre: oposicion_pul

Descripción: Esta capacidad calcula la nueva posición de GDL(3), es decir, del grado de libertad que realiza el movimiento oposición del dedo pulgar.

Objetivos: mover GDL(3)pul a opo; mover GDL(3)pul a pos_pre; obtener GDL(3)pul.
Procesos: *oposicion_pul*
Protocolos: *flexión.*
Mensajes de salida: ninguno.
Mensajes de entrada: pos_pre_GDL(3)pul; movimiento GDL(3)anu.
Mensajes internos: parar GDL(3)pul.
Percepciones: d_pul; posición GDL(3)pul.
Acciones: mover GDL(3)pul.
Capacidades incluidas: ninguna
Datos Usados (Importados): BD dedo pulgar.
Datos Producidos (Exportados): ninguno.
Datos Internos: ninguno.
Capacidades anidadas: ninguna.
Planes incluidos: mantener GDL(3)pul; llevar a pos_pre_GDL(3)pul; oponer GDL(3)pul.
Notas:

2. Procesos

2.1 Proceso de configuracion y flexión del agente pulgar

Nombre: configuracion_flexion_pul

Descripción: En este proceso lo que logra el agente pulgar, además de alcanzar su postura de pre-agarre, es tocar el objeto al mismo tiempo que los otros dedos mientras realiza su movimiento de flexión.
Actividades: llevar a pos_pre_GDL(1y2)pul; mantener GDL(1y2)pul; llevar a pos_pre_GDL(3)pul; revisión y actualización de s-p; llevar a posición_max_pul.
Accionadores:mensajes: pos_pre_GDL(1y2)pul; pos_pre_GDL(3)pul.
Mensajes: pos_pre_GDL(1y2)pul; pos_pre_GDL(3)pul; parar GDL(1y2)pul; pos_pre GDL(1y2)pul alcanzada; pos_pre GDL(3)pul alcanzada; pos_pre_alcanzada_med; pos_pre_alcanzada_ind; pos_pre_alcanzada_anu; orden_mover_anu; orden_mover_med; orden_mover_ind; s-p actualizado.
Protocolos: flexión.

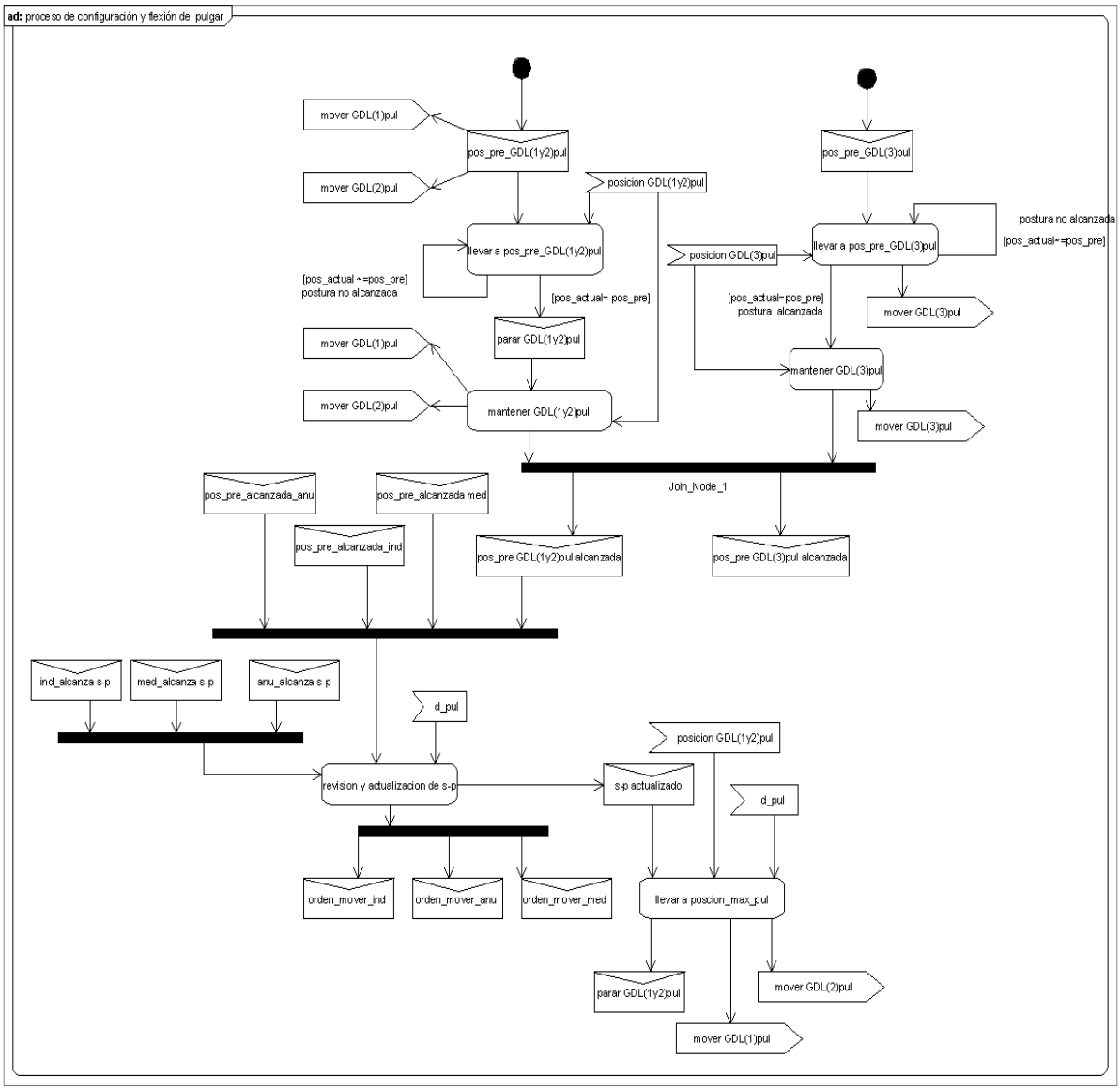


Figura B.3.11. Diagrama de Proceso de configuración y flexión del agente pulgar.

2.2 Proceso de oposicion del agente pulgar

Nombre: oposicion_pul

Descripción: En este proceso lo que logra el agente pulgar se oponga de manera exitosa al agente anular.
Actividades: oponer GDL(3)pul; mantener GDL(3)pul.
Accionadores: movimiento GDL(3)anu.

Mensajes: movimiento GDL(3)anu; parar GDL(3)pul.
Protocolos: ninguno.

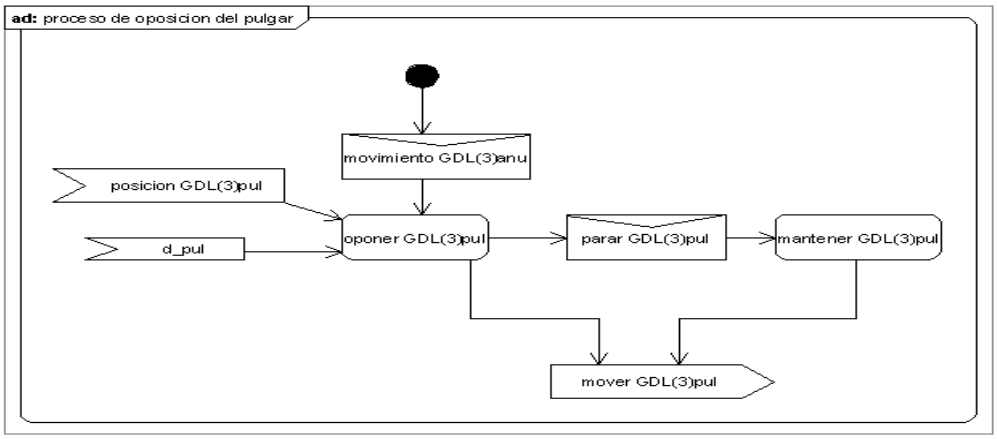


Figura B.3.12 Diagrama de proceso de oposición del agente pulgar.

3. Descriptores de los mensajes internos del agente pulgar

- **Posicion de pre-agarre de los grados de libertad GDL(1y2) han alcanzado su postura de pre-agarre.**

  Nombre: pos_pre GDL(1y2)pul alcanzada
  Descripción: En este mensaje se indca que GDL(1y2) del agente pulgar han alcanzado su postura de pre-agarre.
  Distribución: plan: *llevar a pos_pre_GDL(1y2pul* → plan: *revisión y actualizacion de s-p.*
  Proposito: avisar que la postura de pre-agarre de GDL(1y2) ha sido alcanzada.
  Información llevada: distancia desde la falange próxima y desde la punta del dedo pulgar.
  Cobertura y superposición: cobertura completa, no hay superposición.

4. Descriptores de los planes internos del agente pulgar

- **Revisar la información de los mensajes de entrada y de acuerdo a esta actualizar el nuevo s-p.**

  Nombre: revisión y actualización de s-p
  Descripción: plan que como objetivo principal es el que todos lo agentes toquen el obejto al mismo tiempo.
  Accionador: llegada de cada uno de los mensajes de entrada.
  Contexto: cuando llegen todos los mensajes de entrada.

Mensajes de entrada: ind_alcanza s-p; med_alcanza s-p; anu_alcanza s-p; pos_pre_alcanzada_med; pos_pre_alcanzada_anu; pos_pre_alcanzada_ind; pos_pre_GDL(1y2)pul alcanzada.

Mensaje de salida: orden_mover_ind; orden_mover_anu; orden_mover_med; s-p actualizado.
Datos usados: ninguno.
Datos producidos: ninguno.
Falla: espera prolongada a la llegada de uno de los mensajes de entrada.
Recuperacion de la falla: actualizar s-p únicamente con la información de los mensajes que se tienen.
Procedimiento:

1. Esperar a que llegen todos los mensajes de entrada.
2. Mirar la información de distancia que traen estos mensajes y es*cog*er la menor de estas
3. Enviar los mensajes de salida pero con la distancia menor en su interior.

B.3.3 Agente Medio

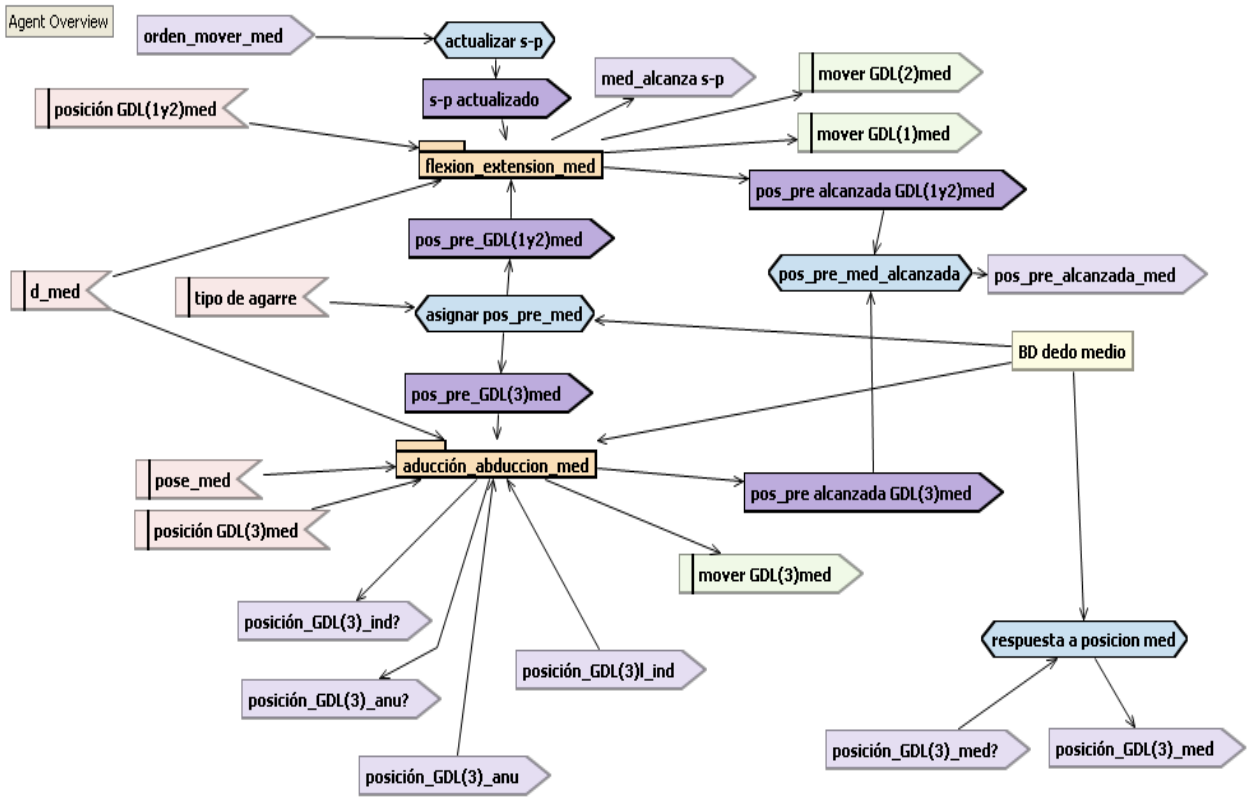


Figura B.3.13 Diagrama de visión general del agente medio

1. Capacidades

1.1 Capacidad de flexion_extensión_med

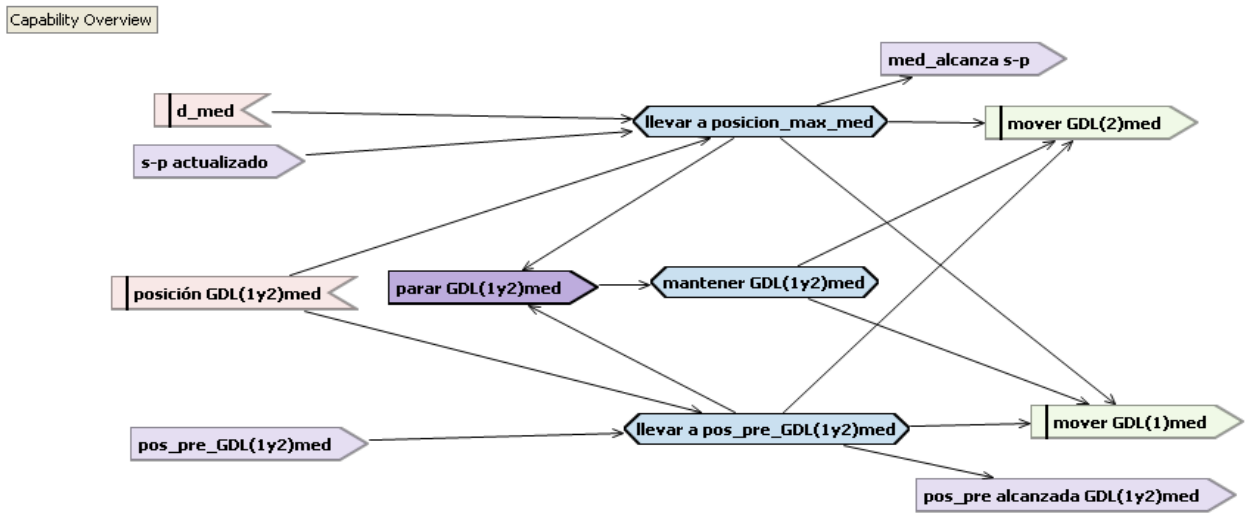


Figura B.3.14 Diagrama de visión general de la capacidad de flexion_extension_med.

1.2 Capacidad de aducción_abducción_med

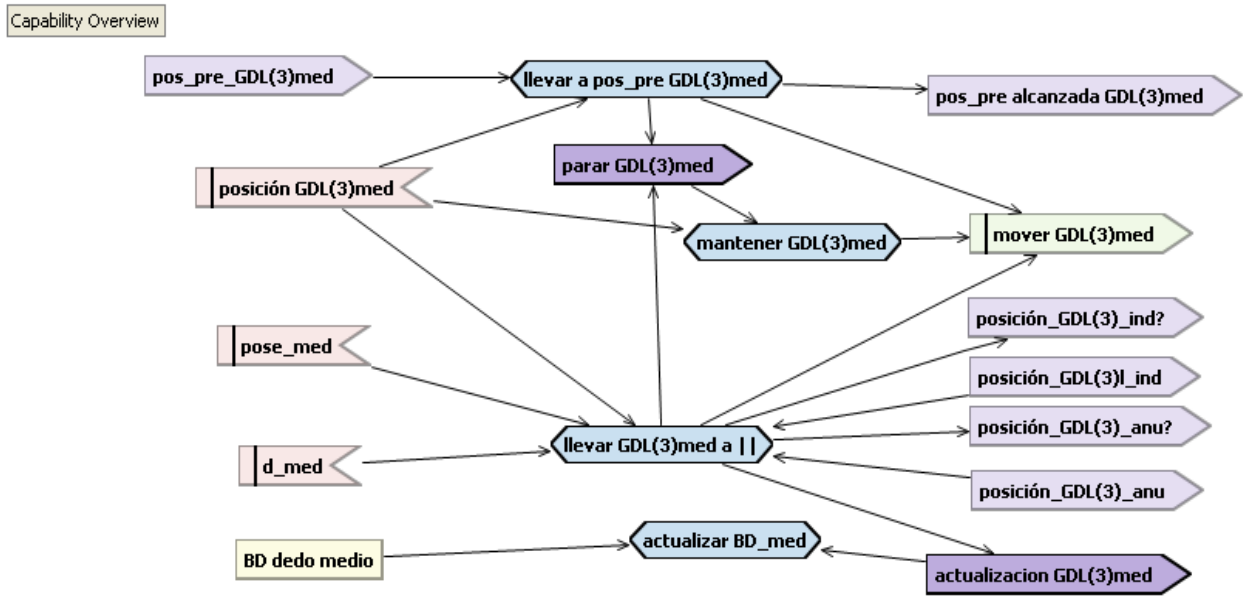


Figura B.3.15. Diagrama de visión general de la capacidad de aducción_abduccion_med.

2. Procesos

2.1 Proceso de configuración del agente medio

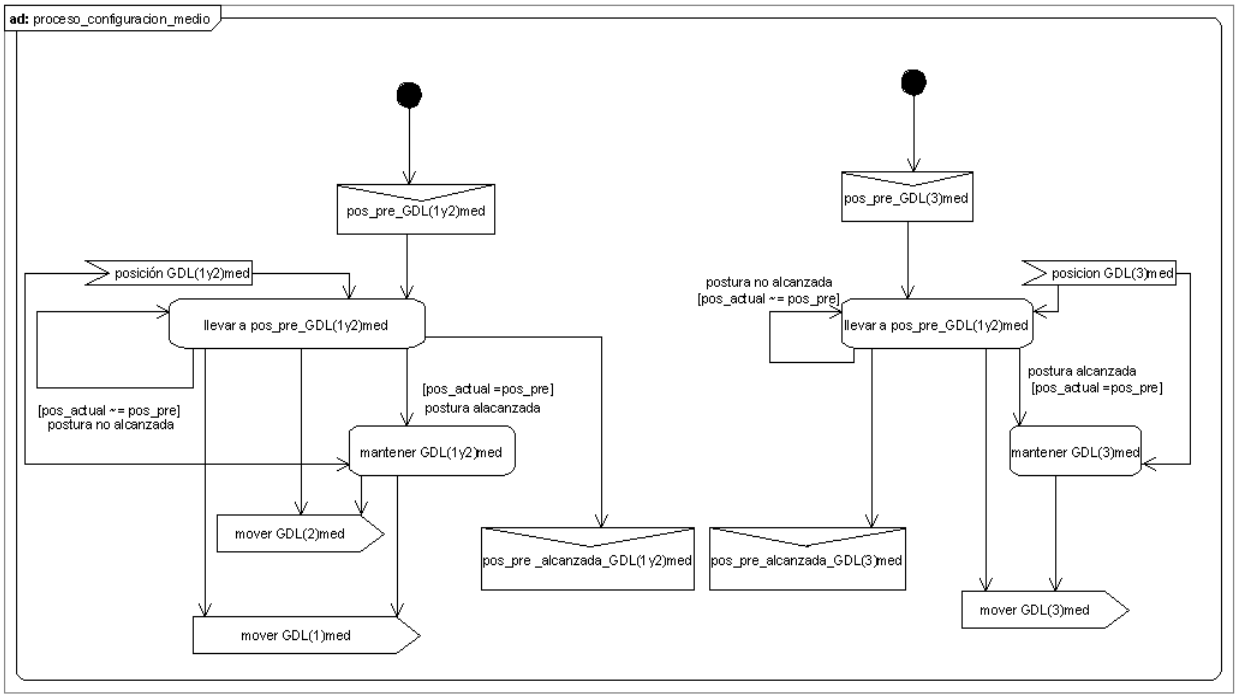


Figura B.3.16. Diagrama del proceso de configuración del agente medio

2.2 Proceso de flexion-extension del agente medio

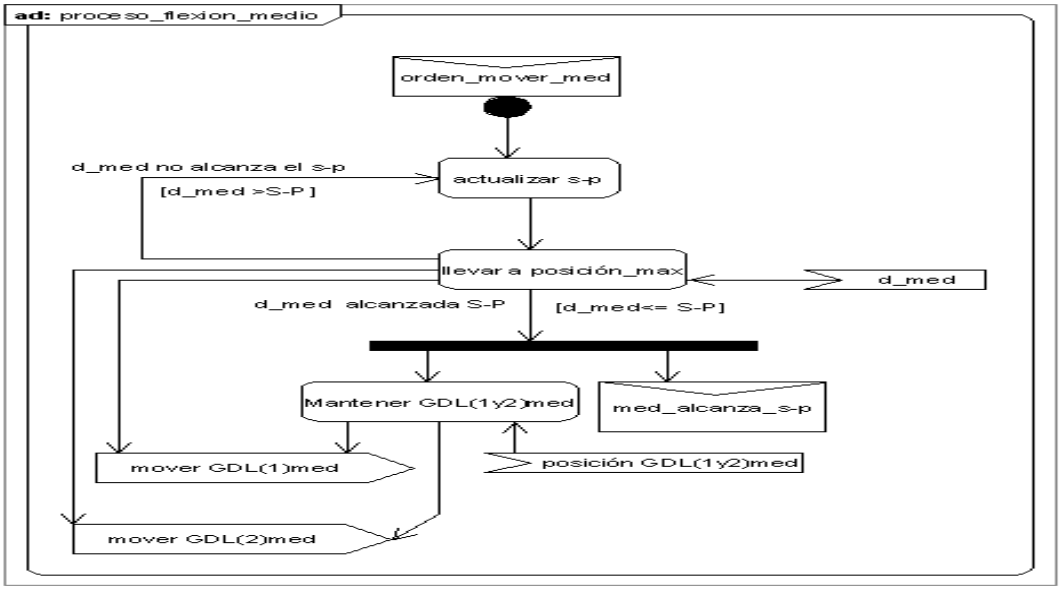


Figura B.3.17. Proceso de flexion_extension del agente medio

2.3 Proceso de aducción-abducción del agente medio

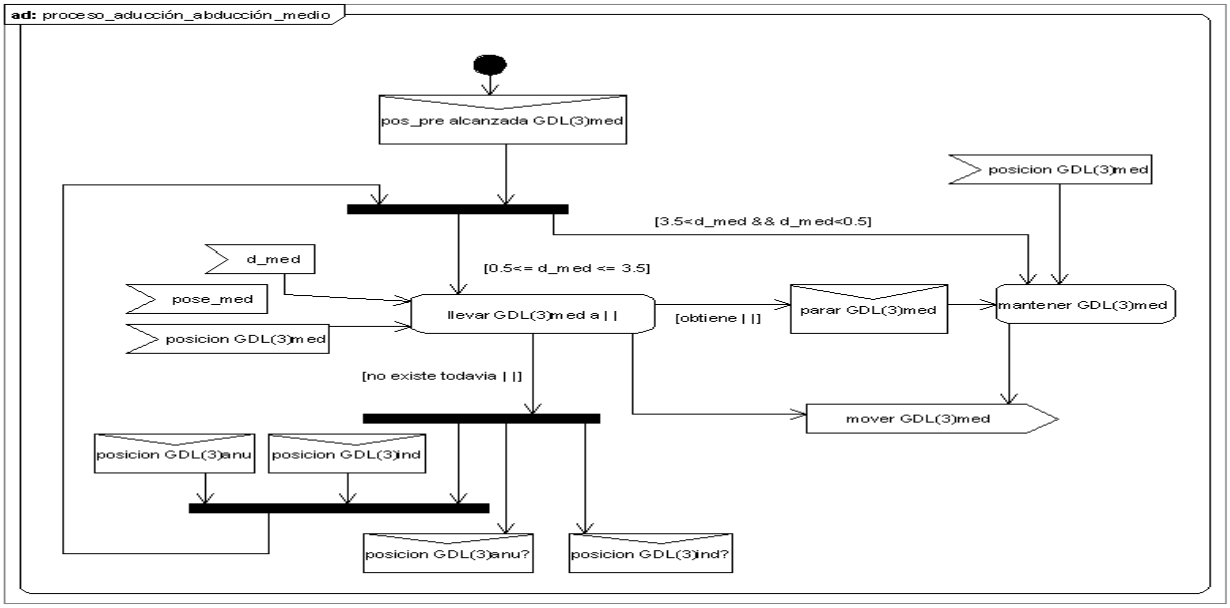


Figura B.3.18. Proceso de aducción_abducción del agente medio

B.3.4 Agente Anular

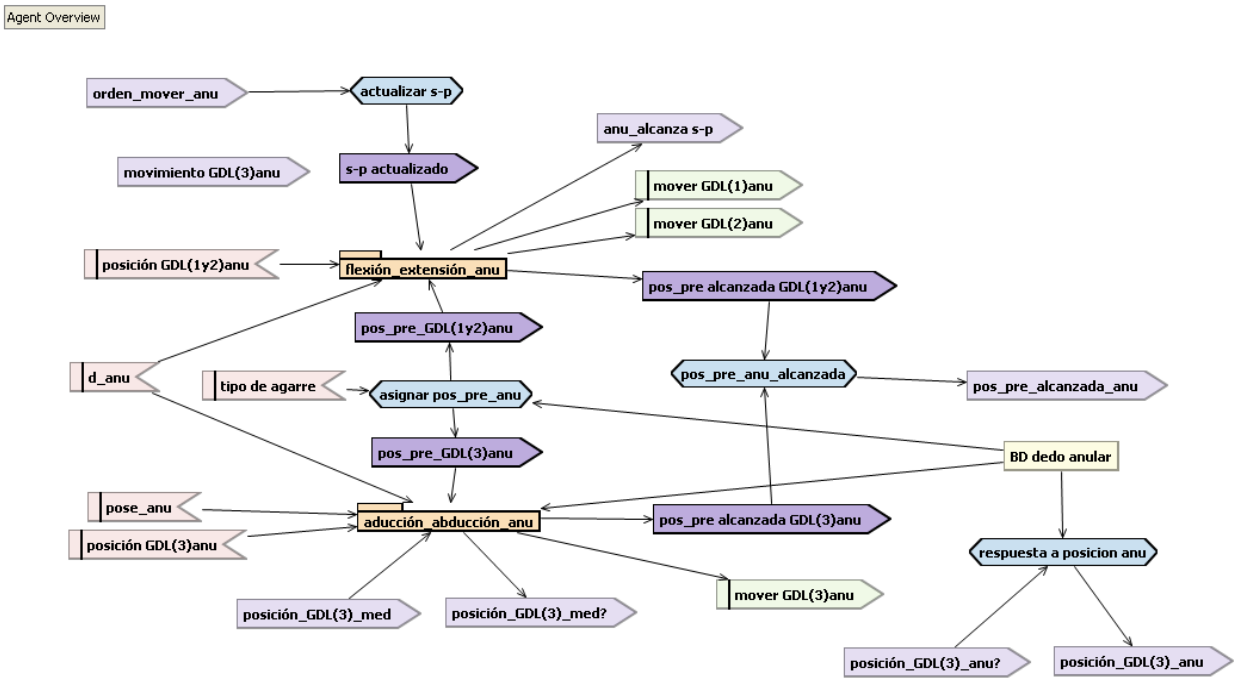


Figura B.3.19. Diagrama de visión general del agente anular.

1. Capacidades

1.1 Capacidad de flexion_extensión_anu

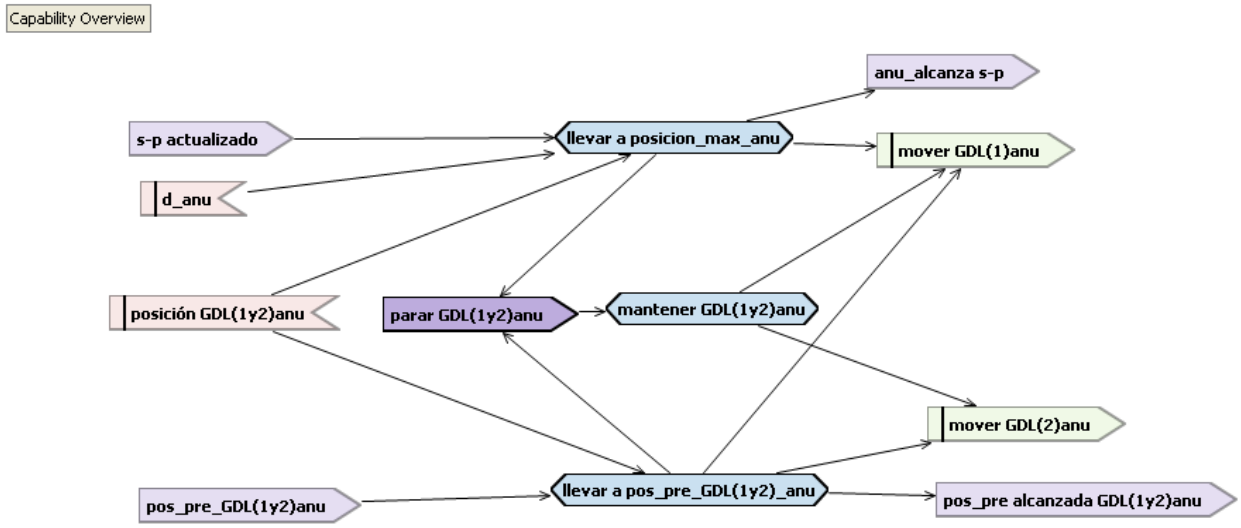


Figura B.3.20. Diagrama de visión general de la capacidad de flexion_extension_anu.

1.2 Capacidad de aducción_abducción_anu

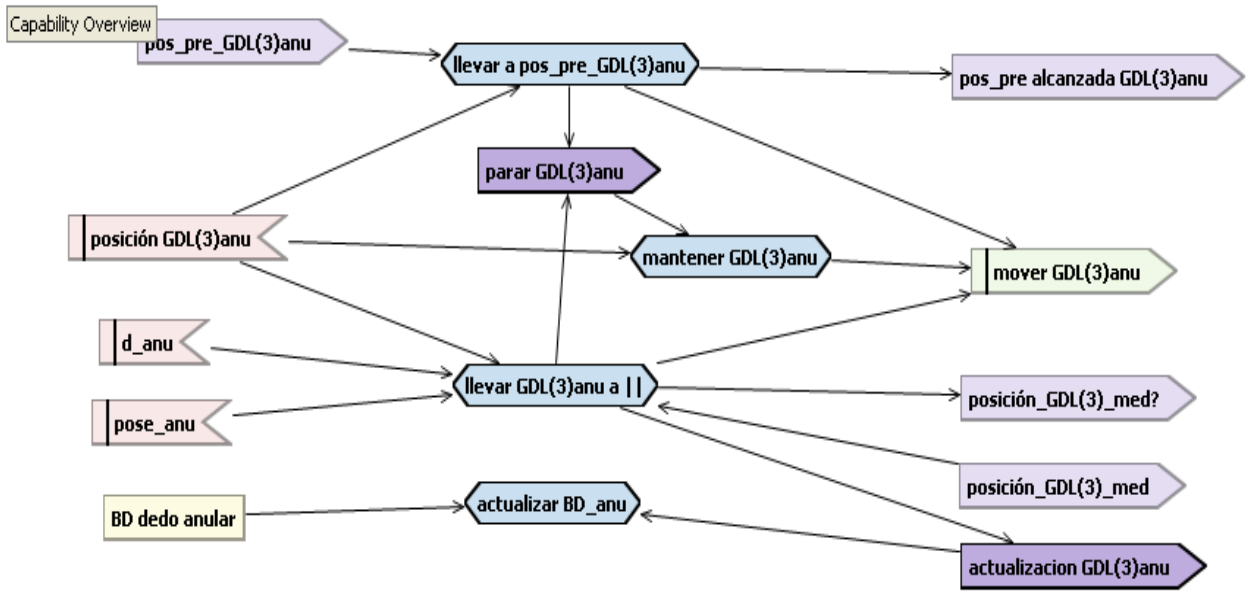


Figura B.3.21. Diagrama de visión general de la capacidad de aducción_abduccion_anu.

2. Procesos

2.1 Proceso de configuración del agente anular

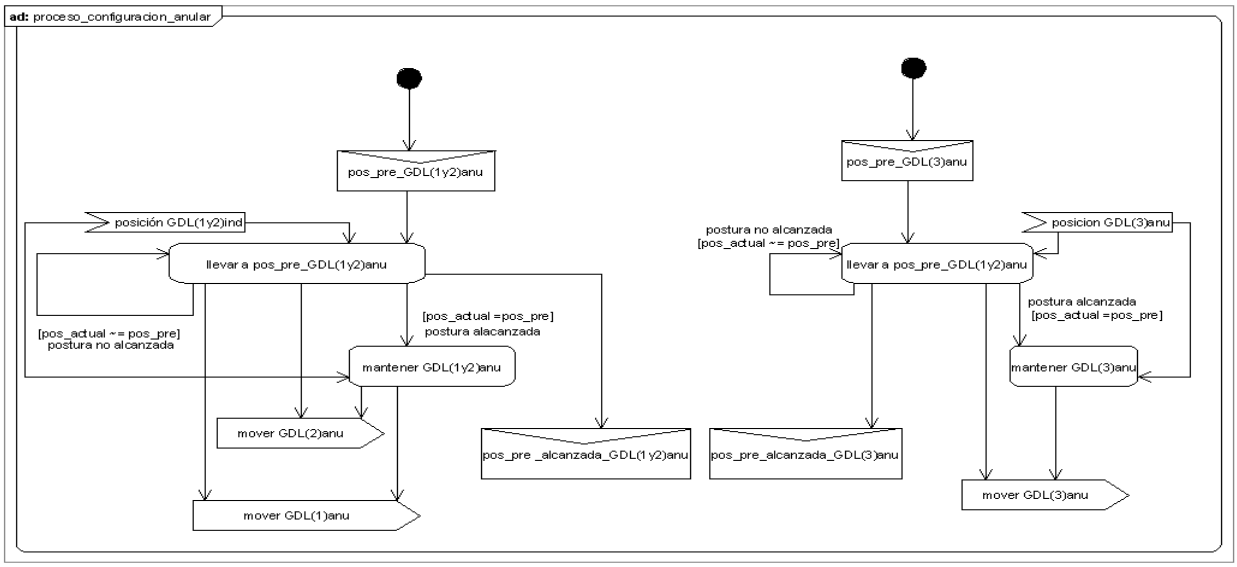


Figura B.3.22. Diagrama del proceso de configuración del agente anular

2.2 Proceso de flexion-extension del agente anular

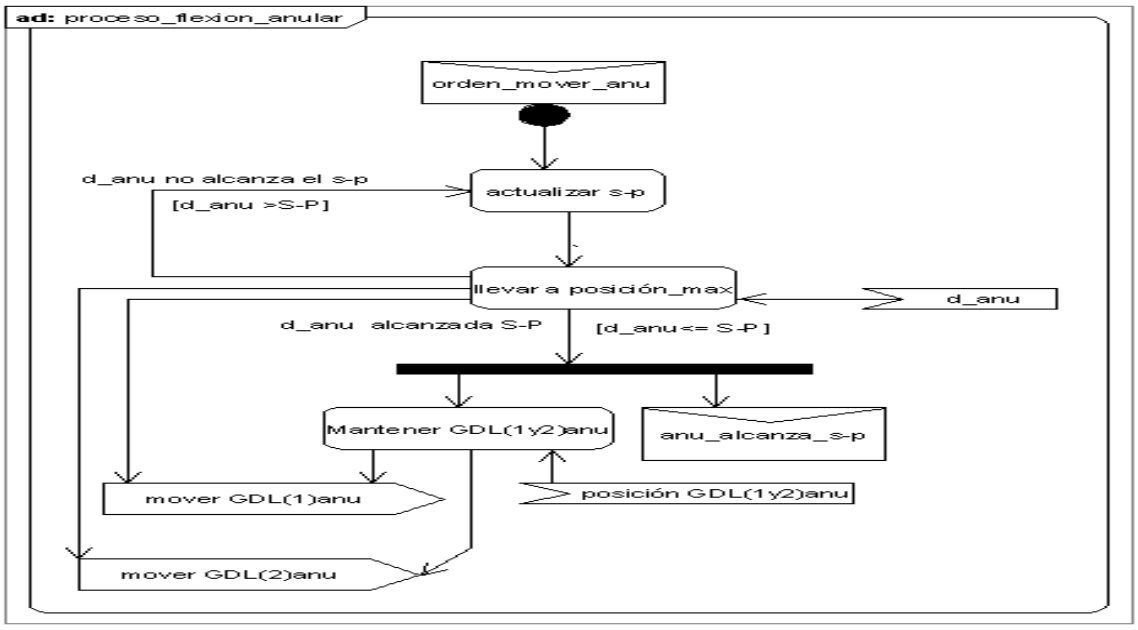


Figura B.3.23. Proceso de flexion_extension del agente anular

2.3 Proceso de aducción-abducción del agente anular

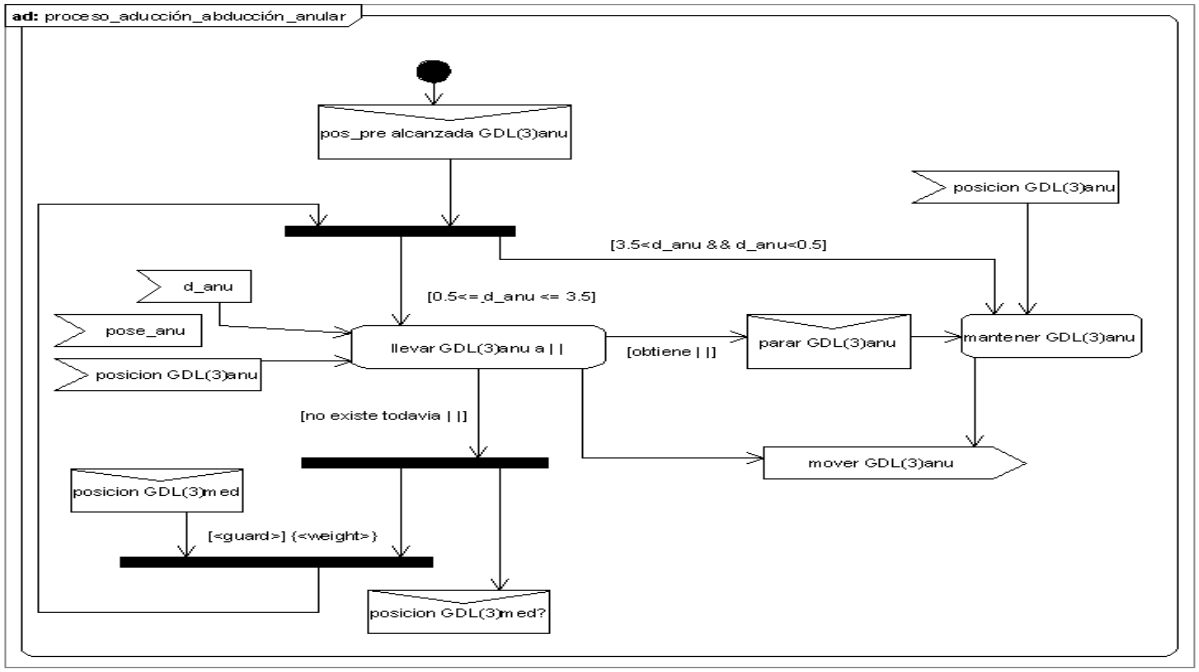


Figura B.3.24. Proceso de aducción_abducción del agente anular.

ANEXO C: SIMULADOR *GraspIt!*

C.1 DESCRIPCIÓN GENERAL DEL SOFTWARE

GraspIt! fue creado para servir como una herramienta para la investigación del agarre. Es un simulador que puede introducir geometrías arbitrarias de manos, robots y obstáculos alojados en un mundo de simulación completo. El motor de *GraspIt!* incluye una detección de colisión rápida y un sistema para determinar la localización de los contactos, permitiéndole al usuario manipular interactivamente un robot o un objeto y crear contactos entre ellos. Una vez se ha creado el agarre, una de las características clave del simulador es el conjunto de métricas de calidad del agarre. Cada agarre es evaluado con una medida numérica de calidad y métodos de visualización permiten al usuario observar los puntos débiles y crear proyecciones arbitrarias 3D del espacio *Wrench* 6D del agarre.

En general, *GraspIt!* usualmente sirve para dos propósitos: Primero, puede ser usado como una herramienta de desarrollo, para ejecutar y probar algoritmos de control de Robots. En este sentido, *GraspIt!* sirve como reemplazo del mundo real: en simulación, un algoritmo puede ser probado en muchos diseños de manos, objetos y configuraciones de obstáculos sin costo alguno y mucho mas rápido que en el mundo real. En segundo lugar, *GraspIt!* puede ser usado como una plataforma computacional que emula un robot que opera en el mundo real. Por ejemplo, el robot real puede obtener un modelo del objeto objetivo y entonces usa *GraspIt!* para evaluar rápidamente múltiples agarres o escenarios de manipulación. A menudo estos escenarios se combinan y la misma configuración usada para el desarrollo de un algoritmo puede ser usada para cálculos durante la ejecución en tiempo real. *GraspIt!* cuenta también con otras características adicionales, entre las que podemos encontrar el motor de dinámicas y la capacidad de planificación de agarre.

El motor de dinámicas calcula los movimientos de un grupo de elementos robóticos conectados, tales como un brazo y una mano, bajo la influencia de fuerzas motoras controladas, fuerzas de rozamiento en las articulaciones, fuerzas de contacto y fuerzas externas. Esto le permite al usuario simular completamente la dinámica de una tarea de agarre. En general, los algoritmos de planificación del agarre dependen de ambientes simulados para evaluar rápidamente muchas posturas de manos y encontrar la que permita un agarre estable. Hay muchas posibles implementaciones de éste concepto; los planificadores que se incluyen en *GraspIt!* pueden usualmente encontrar múltiples agarres estables de un objeto rápidamente, usualmente en menos de un minuto, tomando en cuenta obstáculos y otra restricciones. En las siguientes secciones se hace una breve descripción de algunas características del software, con el propósito de mostrar los beneficios más sobresalientes que este sistema ofrece en la investigación del agarre robótico.

C.2 MANUAL DE INSTALACIÓN DE *GraspIt!*

El proceso de instalación en Windows de *GraspIt!* Consiste de dos pasos. Primero, se debe instalar un conjunto de librerías externas, listadas abajo. Luego, se debe compilar el código mismo de *GraspIt!* y configurar las variables de entorno.

GraspIt! Necesita las siguientes librerías externas:

Qt es una aplicación multi-plataforma. Esto permite a *GraspIt!* tener un sistema de ventanas independientes de la plataforma, diálogos, etc.
En esta instalación se usa la versión 4.5 de Qt de fuente abierta, la cual esta disponible bajo dos licencias. Para mas detalles de opciones de licencia revise el sitio web de Qt.

Coin es un API grafico de escenas que *GraspIt!* utiliza para todas sus necesidades de representación 3D. En esta instalación se usa la versión 2.4.6 de Coin. y esta disponible bajo dos licencias; aquí se usa la licencia de fuente abierta. Mire el sitio web de *Coind3D* para más información de la licencia.

SoQt es un enlace GUI, esencialmente la unión entre *Qt* y Coin. La versión de *SoQt* que usamos es *SoQt 1.4.1*, la cual provee una integración completa con *Qt4*. Esta disponible bajo las mismas condiciones de Coin, en el mismo sitio web.

Lapack es una librería para la calculo de problemas de algebra lineal que *GraspIt!* usa para varias tareas relacionadas con matrices, tal como multiplicación, solución de sistemas lineales, descomposición de valores singulares, etc.

La información del proyecto *GraspIt!* viene en la forma de un archivo de proyecto de Qt multi-plataforma, el cual es procesado por *QMake*. Dependiendo del sistema, se convertirá al archivo de proyecto de Qt en un archivo *Makefile* o en un archivo de proyecto de *Microsoft visual estudio.*

En esta instalación se asume se esta usando el compilador de *Microsoft Visual Studio.* Algunos de los *path* usados en este manual son tomados de *MS Visual Studio 2008*; el cambio a su versión particular de compilador debería ser sencillo, ya que se ha probado con *MS Visual Studio 2005.*

C.2.1 Configuración De Variables De Entorno En Windows

Ejemplo de configuración de variable entorno: Damos clic en *Inicio* y luego clic derecho en *Mi PC*, ahí vamos a *Propiedades* y en la pestaña de Opciones Avanzadas le damos clic en *Variables de Entorno*.

Aparece un cuadro de diálogo con las variables de sistema y las variables de usuario. Para configurar una variable de sistema se debe:

1. buscar la variable, la seleccionamos y damos clic en *Modificar*. Por ejemplo la variable ***Path***
2. Modificamos el valor de la variable. Por ejemplo, para la variable *Path* agregamos al final del texto un punto y coma y después escribimos la ruta que agregar. D:*GRASP*\Qt\4.5.1\qt\bin
3. Damos clic en Aceptar en todos los cuadros de diálogo.

Para configurar una variable de usuario se realiza un procedimiento similar, pero normalmente es una nueva variable. Para configurar una nueva variable de usuario se debe:

1. En las variables de usuario demos click en ***nueva o add***,
2. Digitamos el nombre de la variable
3. Digitamos el valor de la variable.
4. Damos clic en *Aceptar* en todos los cuadros de diálogo.

Por ejemplo, agregamos la variable de entorno QTDIR.
Nombre de la variable QTDIR
Valor =D:*GRASP*\Qt\4.5.1\qt

En adelante cuando para un ***Path*** en el manual se dice: $QTDIR esto es igual al path de la carpeta raíz de la herramienta. Por ejemplo para Qt podría ser: D:*GRASP*\Qt\4.5.1\qt.

C.2.2 Instalacion De Qt

Copie Qt del CD. Luego instálela en el directorio de su elección. Instale la versión regular, no el instalador *mingw.*

Advertencia: No use espacio en el nombre del *path* donde instala *Qt*, ni en ninguna de las herramientas que instale mas adelante.

Configure las siguientes variables de entorno de usuario:

- **Nombre:** QTDIR
- **Valor:** el directorio donde se instalo Qt (ej: D:*GRASP*\Qt\4.5.1\qt)
- **Nombre:** QMAKESPEC
- **Valor:** *win32-msvc2008.*

El valor de esta variable depende de su compilador *C++* y la plataforma. Mirar *$QTDIR/mkspecs* para una lista de opciones.

Configure la variable de entorno del sistema **Path**:

- **Nombre:** Path
- **Valor:** Se agrega $QTDIR/bin.

También se podría necesitar agregar algunos paths específicos de su compilador a sus variables de entorno. Si la configuración de Qt falla, trate de mirar cual archivo fallo para encontrarlo y luego agregarlo al path respectivo del ***path, INCLUDE o LIB***. Tenga cuidado dónde agrega una variable de entorno particular, ya que cuando se agrega un path, el sistema usara el primer path que encuentre, el cual podría no ser el que usted planeo. Aquí están como ejemplos, los path que se necesitan agregar cuando se usa *MS Visual Studio 2008*.

- ***path:***

  -El path a *nmake.exe*
  Ej: D:\Program *File*s (x86)\VisualStudio\VC\bin;
  -El path a los *dlls comunes*
  Ej: D:\Program *File*s (x86)\VisualStudio\Common7\IDE

- ***INCLUDE:***

  D:\Program *File*s (x86)\VisualStudio\VC\include

- ***LIB:***

  D:\Program *File*s (x86)\VisualStudio\VC\lib

  Siga las instrucciones dadas con la distribución de Qt para la instalación.

Nota Importante: la configuración de Qt podría tomar un par de horas, mas o menos 4.5 horas. Se recomienda construir tanto las versiones *debug* como la *release* de Qt, de esta manera se dispondrá más adelante de todas las opciones de enlace.

A continuación se procede a configurar Qt para *visual estudio versión 2008*. Vamos a *Inicio* → *Todos los Programas* → *Microsoft Visual C++ 2008* → *Visual Studio Tools* → *Símbolo del Sistema de Visual Studio 2008*. Esto nos abre una consola en el directorio del *Visual C++* y habilita los paths para el *maker*, y el *linker.*

En esta consola nos vamos a *QTDIR* (o el directorio donde hayan descomprimido Qt), una vez allí ejecutamos:

>> configure -debug-and-release

Este comando prepara el código para que se pueda compilar con el compilador de *Microsoft Visual Studio 2008*, una vez termine el proceso veremos un mensaje como este:

Qt is now configured for building. Just run nmake.
To reconfigure, run nmake confclean and configure.

Ejecutamos *nmake* para que Visual C++ compile.

Nota: Este proceso dura más o menos 4.5 horas y no se debe olvidar que se deben tener rutas diferentes, así que debe adaptarlas a su configuración.

Avertencia: Otros programas usan archivos binarios de *Qt*, y a menudo incluyen los *dll's* de Qt en su distribución (tal como *QtGui4.dll*). Si existe alguna otra ruta a los *dll* de *Qt* en la variable de entorno path antes que *$QTDIR/bin*, el sistema enlazará *GraspIt!* a una versión incorrecta causando a menudo fallas al correrlo. Para estar seguros de que usted esta enlazando correctamente los *dll's*, coloque *$QTDIR/bin* al principio de su path.

C.2.3 Instalacion De Coin

Copie *Coin* del CD. Luego descomprímala en el directorio de su elección.
Configure las siguientes variables de entorno:

- **Nombre:** *COINDIR*
- **Valor:** El directorio donde usted descomprimió *Coin* (ej: *C:/Coin3D/2.4.6*).
- **Nombre:** Path
- agregue el path a coin2.dll (ej: *C:/Coin3D/2.4.6/bin*).

C.2.4 Instalacion De SoQT

Copie *SoQt* del CD. Luego descomprímala en el directorio de su elección.
Usted necesita generar *SoQt* del código fuente. El código fuente viene en archivos de solución de *MS Visual Studio*, escoja el path para *Visual Studio 2008*. Es decir, acceda a *$SoQt\build\msvc8*.

De nuevo, recomendamos generar tanto la versión *debug* como *release*. Por lo tanto, debe compilar *soqt1.sln* y generar la versión *debug* y luego la versión *release*.

Después de generar el código, eche un vistazo en *$COINDIR/bin* para asegurarse de que han sido construidos los *dlls* de SoQt (*soqt1.dll* y/o *soqt1d.dll*).

C.2.5 Instalacion De Lapack

Copie la versión de *Intel Math Kernel Library* (MKL) del CD. Luego instálela en el directorio de su elección. Para enlazar *GraspIt!* configure la variable de entorno *MKLDIR*:

- **Nombre:** *MKLDIR*
- **Valor:** El directorio donde instalo MKL *($MKLDIR*)
- **Nombre:** Path
- **Valor:** *$MKLDIR\ia32\bin*

Luego, en el archivo de proyecto principal de *GraspIt!* configure la bandera apropiada (según se explica mas abajo).

Si el enlace con MKL falla, revise el archivo de proyecto de *GraspIt!* especifico para Windows (***graspit-lib-WINDOWS.pro***), encuentre el bloque de *blas/lapack* y trate de adaptarlo a su configuración.

C.2.6 Instalación De *GraspIt!*

Copie *GraspIt!* del CD. Luego descomprímalo en el directorio de su elección. Configure la siguiente variable de entorno.

- Nombre: *GRASPIT*
- Valor: el directorio donde usted descomprimió *GraspIt!*

A continuación generar *QHull.* Para ello, abra y genere la solución de *MS Visual Studio* ***$GRASPIT/qhull/windows/qhull.sln*.** Una vez mas, recomendamos generar tanto las versiones *debug* y como *release*. Asegúrese de que *qhull.lib* ha sido instalado en el directorio apropiado *($GRASPIT/qhull/windows/Debug and/or $GRASPIT/qhull/windows/Release*).

NOTA: Edite *$GRASPIT/graspit.pro* de acuerdo con la instalación particular.
Para es*cog*er la versión de Lapack a usar, en nuestro caso la de Intel *Math Kernel Library (MKL)*, solo cambie el valor de *LACPACK a mkl*. A continuación se muestra la porción de código donde se hace el cambio

#select blas and lapack libraries, Windows-only
#possible values: mkl clapack
LAPACK = ***mkl*** (linea cambiada)

Para escoger la versión de *GraspIt!* a generar se debe cambiar la configuración apropiada (*Debug o Release*). A continuación se muestra la porción de código donde se hace el cambio para generar el modo *release*:

#comment out this line for compiling in Release mode
#usually, compiling in Release mode delivers a significant gain in performance
#in MS Visual Studio *also* set the project mode to Release
#CONFIG += *Grasp*itdbg

Advertencia: No es suficiente cambiar entre la construcción de *Debug y release* usando solamente la opción en *MS Visual Studio*. Esto enlazara las librerías apropiadas de Qt pero no enlaza con las librerías de *Coin, SoQt y Qhull*; la manera correcta de seleccionar una construcción *Debug o Release* es también editando el archivo *graspit.pro.*

Luego de seleccionar el modo de la compilación, cree el archivo de proyecto *GraspIt!* de *MS Visual Studio* del archivo de proyecto de Qt. Para ello, ejecute el siguiente comando dentro de *$GRASPIT* en la consola de comandos de MS Visual Studio:

qmake -t vcapp -o graspit.vcproj graspit.pro

Esto crea el archivo de proyecto *GraspIt! MS Visual Studio*. Abra *graspit.vcproj* en MS Visual Studio. Genere *GraspIt!* En este momento ya puede correrlo.

Vaya a la carpeta *$GRASPIT\bin* de doble click sobre *graspit.exe* y disfrútelo!

C.3 MANUAL DE USO DE *GraspIt!*

Para comenzar, usar el menú ***File → Open*** y cargar el archivo ***dlr_flask.wld***. Éste es un mundo de simulación simple que solo contiene una mano (el modelo DLR) y un objeto (un *matraz*). Ver Figura C.3.1.

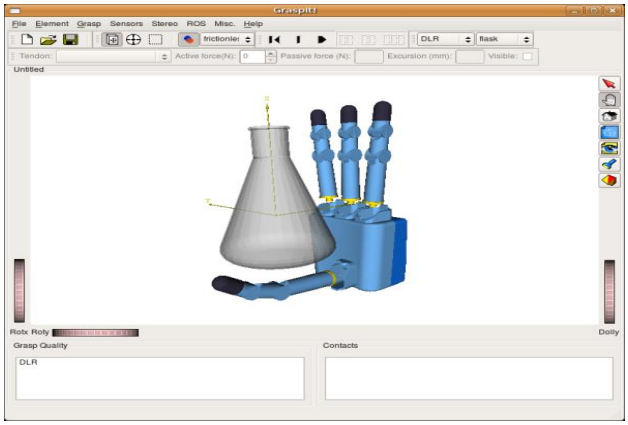


Figura C.3.1 El mundo cargado desde el archivo dlr_*flask*.wld [6]

También se puede comenzar con un mundo de simulación vacio y luego importar robots y objetos, uno al tiempo, usando la opción ***File*** → **Import**. El mundo de simulación creado se puede guardar mediante la opcion opción ***File*** → **Save** en un archivo de mundo, tipo *.wld*.
En este corto tutorial, se usará para la simulación algunos archivos de mundos suministrados con la distribución de *GraspIt!*

C.3.1 La Ventana Principal Y Los Controles.

La mayor parte de la ventana principal de *GraspIt!* es ocupada por el visor Inventor[1], el cual representa el mundo virtual. En la parte derecha hay una barra de herramientas: esta es la barra de herramientas Inventor la cual es responsable de la interacción con la cámara.

Los primeros dos botones en la barra de herramientas determinan el estado del visor. Seleccionando , el visor esta en modo **Interacción.** Éste es el único modo en el cual se puede interactuar con los objetos en el mundo de simulación. Seleccionando , el visor pasa al modo **cámara.** Este es el único modo en el cual se puede mover la cámara.

Se puede también cambiar entre **modo interacción** y **modo cámara** presionando las teclas <*ESC*> *o* <*ALT*>, aunque no trabajan de la misma manera en todos los sistemas.

C.3.2 El Modo Cámara

Cuando el visor esta en modo **cámara**, se puede mover la cámara virtual de las siguientes maneras:

- **Rotar** - Presionar el botón del lado izquierdo del ratón y arrastrar.
- **Trasladar** - Presionar <*CTRL*> y el botón izquierdo del ratón y arrastrar; alternativamente, tambien se puede presionar el botón medio del ratón y arrastrar.
- **Zoom** - presionar simultaneamente el botón izquierdo y medio del ratón y arrastrar, o rotar la rueda que se encuentra en el medio de los ratones.

Adicionalmente, son útiles los siguientes botones del la barra de herramientas Inventor:

[1] El visor Inventor es una escena gráfica escrita en C++ en la cual se pueden cargar diferentes objetos y realizar tareas de interacción comunes tales como: acercar o alejar un objeto, verlo desde diferentes perspectivas o simplemente trasladarlo o rotarlo.

- - Éste botón *auto*máticamente mueve la cámara para que toda la escena pueda verse completamente en la ventana.
- - La herramienta de búsqueda permite dar clic sobre un objeto en la escena. Después del clic, la cámara se acerca al objeto, y en donde se dio el clic será el nuevo centro de rotación de la escena.

C.3.3 El Modo Interacción

Cuando el visor esta en modo **interacción,** se puede interactuar con los objetos en la escena. El tipo de interacción es determinado por los siguientes botones en la barra de herramientas de *GraspIt!*:

- - **Trasladar un cuerpo o mover una articulación**. Al dar clic sobre un cuerpo o la base (palma) de un robot da como resultado que se dibuje una caja alrededor de éste. Esta caja actúa como un manipulador de traslación. Dando clic sobre los lados de la caja y arrastrando el ratón, el cuerpo o robot puede ser trasladado en las dos dimensiones alineadas con la cara de la caja a la cual se dio clic. Presionando <*SHIFT*> se restringe este movimiento a un solo eje.

  Dando clic sobre una cadena cinemática da como resultado que se dibujen los manipuladores de las articulaciones para cada DOF. Estos manipuladores se pueden utilizar para mover una articulacion de la cadena seleccionada, dando click sobre el manipulador de la articulacion y arrastrando en la direccion deseada.

- - **Rotar**. Al dar clic en un cuerpo o el enlace base del robot aparecerá un manipulador de rotación esférico. Arrastrando la esfera del manipulador permite rotar en tres dimensiones el cuerpo seleccionado. Haciendo clic en una de las bandas dibujadas sobre la esfera, el cuerpo puede ser girado sobre un eje a la vez.
  Arrastrando las crucetas retículos, la esfera puede ser recentrada para girar alrrededor de un punto diferente.

- - **Seleccionar**. Al dar clic sobre un cuerpo éste es seleccionado, y se indica mostrando la estructura reticular del cuerpo. Presionando <*SHIFT*> permite seleccionar múltiples cuerpos, o dando clic sobre un enlace del robot ya seleccionado se seleccionará todo el robot *auto*máticamente. Una vez un cuerpo (o cuerpos) han sido seleccionados, sus propiedades de colisión y de material se muestran en la siguiente parte de la barra de herramientas. Tales propiedades o propiedades adicionales pueden ser cambiadas usando

el menu que se describe mas adelante. Después de seleccionar un cuerpo, se puede eliminar de la escena presionando *<DEL>.*

Trate de usar la herramienta para flexionar los dedos del robot. Note que cuando los dedos del robot tocan el objeto, se indica el contacto mediante un cono de color rojo y no se permite mas flexion de los dedos. El mismo comportamiento aplica al mover objetos o robots.

Los siguientes botones en la barra de herramientas de *GraspIt!* se aplican al cuerpo que ya ha sido seleccionado:

- **Cambiar propiedades colisión**. Hay tres maneras de usar esta propiedad:

- Si no se ha seleccionado ningún cuerpo, este botón permite cambiar la propiedad de colisión del mundo completo de *on* a *OFF*, y viceversa.
- Si uno o mas cuerpos han sido seleccionados, este botón cambia la propiedad de colisión de estos. Un cuerpo con la propiedad de colisión en *OFF* puede atravesar otro cuerpo.
- Si se seleccionan dos cuerpos, este botón permite deshabilitar la propiedad de la colisión SOLO entre estos dos cuerpos.

-

- Material. Este menu configura el material de los cuerpos seleccionados, el cual afecta el coeficiente de fricción cuando tiene lugar un contacto.

  Al deshabilitar la colisión en el mundo completo, note que los objetos o los dedos pueden moverse libremente, aun si esto da como resultado una colisión. Asegúrese de cambiar de lugar los objetos de modo que no existan colisiones entre ellos **antes** de volver a activar la propiedad de colisión; si esto no se hace, será imposible mover los objetos en colision.

  Cree uno o más contactos entre la mano y el objeto. Una vez se tiene el contacto, seleccione uno de los cuerpos en contacto (tal como el enlace del robot que está tocando el matraz o el matraz mismo) y cambie su material. Note como cambia también el cono de fricción que marca el contacto.

C.3.4 Ejemplo De Agarre

Para comenzar cargue de nuevo ***dlr_flask.wld*** y asegúrese que todos los elementos del mundo de simulación estén en sus posiciones originales. Después usar menú ***Grasp → Auto Grasp***. Esto causará que todos los dedos del robot se flexionen hasta que estar en contacto con el matraz impidiendo todo movimiento adicional. El resultado es un agarre del objeto.

Usar el menú ***Grasp → Quality Measures*** para crear una nueva medida de calidad, que será usada para este agarre. Por defecto, el diálogo que aparece para la medida de calidad creará una nueva medida de calidad del agarre llamada ***New Quality Measure*** del tipo ***Epsilon***

usando un espacio *wrench L1*. Dar clic en ***Add/Edit***, y luego en ***OK***. La nueva medida de calidad del agarre, al mismo tiempo que su valor, se mostrará en la parte de abajo-izquierda de la ventana principal de *GraspIt!*.

También se puede crear una proyección del espacio *Wrench* del agarre. Usar el menú ***Grasp → Create GWS Projection***. Hacer clic en las tres casillas de verificación marcadas con **tx**, **ty** y **tz** y dar clic en **OK**. *GraspIt!* mostrará el espacio de fuerzas que este agarre puede aplicar sin generar un par neto. Notar que si se mueve la cámara en el visor principal de *GraspIt!*, la cámara que muestra la proyección GWS también se mueve. Los ejes de la proyección GWS estan siempre alineados con los del visor principal.

C.3.5 Un Ejemplo De Dinámica

Comenzar cargando el mundo de simulación ***barrettGlassDyn.wld***. Luego dar inicio al motor de dinámicas presionando el botón ▶ de la barra de herramientas de dinámicas. Note que las articulaciones del robot se mueven ligeramente y la copa rueda suavemente sobre la mesa. Los controladores PD, en las articulaciones del robot, están simplemente manteniendo la posición actual oponiendose a la gravedad.

Usar el menú ***Grasp → Auto Grasp*** para cerrar los dedos de la mano. Note que la mano comienza a cerrarse y levanta la copa en su agarre. Después que se estabiliza el agarre, seleccione la copa y cambie sus propiedades de material a ***frictionless***. La copa entonces se desliza del agarre y termina resbalando fuera de la mesa. Se puede pausar el motor de dinámicas en cualquier momento dando clic en el botón ***pause*** de la barra de herramientas de dinámicas.

C.3.6 Los Menus

A continuacion se describe un subconjunto de las funcionalidades que se encuentran en los menús.

File Menu

- **New** - Vacía el mundo de simulación actual y vuelve a ceros el tiempo de simulación.
- **Open**... Carga una nueva configuración del mundo.
- **Save**, **Save As**... Guarda la configuración del mundo actual. Las velocidades no se guardan.
- **Save Image**... Hace una toma de la escena usando la cámara actual y guarda la imagen en un formato jpg. A la imagen se le aplicara un filtro anti-aliasing.

- ***Import Robot*** ... Carga un robot desde un archivo de configuración de robot y lo coloca en el origen del mundo. Despues de cargar un robot o un cuerpo, click ***view all*** si el objeto importado no se muestra en el visor.
- ***ImportObject****...* Carga un modelo Inventor y lo coloca en el mundo como un cuerpo que puede ser agarrado. El archivo Inventor debe tener la masa y el material definidos en los comentarios. El cuerpo se hace transparente para que sean visibles los contactos.
- ***ImportObstacle****...* Carga un modelo Inventor y lo coloca en el mundo como un objeto estático. Estos objetos deben ser movidos por el usuario pero sus movimientos no son calculados durante la simulación dinámica.
- ***Edit Settings****...* Permite al usuario cambiar las configuraciones del programa en curso. Las configuraciones son guardadas en el registro de Windows, y en un archivo rc en Linux.
- ***Exit*** – Para salir del programa.

Element

- ***Translate*** - Hace lo mismo del botón Translate de la barra de herramientas.
- ***Rotate*** Hace lo mismo del botón Rotate de la barra de herramientas.
- ***Select*** Hace lo mismo que botón Select de la barra de herramientas.
- ***Collisions ON/OFF*** Hace lo mismo del botón de la barra de herramientas que determina si hay o no colisión.
- ***Body Properties****...* Muestra un cuadro de diálogo que le permite al usuario editar las propiedades de los cuerpos que han sido seleccionados.

Grasp

- ***Auto Grasp*** Realiza un agarre *auto*maticamente. Cuando las dinámicas no están corriendo, éste cierra los dedos de una mano hasta que se alcancen los límites de las articulaciones o hasta que se detecta un contacto. Las velocidades relativas de las articulaciones estan definidas en el archivo de configuración del robot. Cuando las dinámicas están corriendo, un generador de trayectorias creará las trayectorias de las posiciones con que se cerrarán los dedos. Entonces, los controladores de las articulaciones ajustarán la fuerza aplicadas a las articulaciones usando las posiciones generadas como un conjunto de *set-points*.
- ***Create GWS Projection****...* Abre un cuadro de diálogo que le permite al usuario es*cog*er una proyección del espacio *Wrench* 6D del agarre. Después de es*cog*ida la proyección, *GraspIt!* abre una nueva ventana mostrando el espacio *Wrench* proyectado. El volumen es actualizado cada vez que el agarre cambia.
- ***Quality Measures****...* Le permite al usuario crear una nueva medida de calidad que será evaluada cada vez que el agarre cambia.
- ***Planner****...* Abre un cuadro de diálogo que contiene la configuracion del planificador de agarres *auto*mático. Por el momento este solo trabaja con la mano Barrett.

C.3.7 Tipos De Datos Principales Y Archivos De Datos

Hay tres tipos principales de entidades en *GraspIt!* El primero es el ***Body*** (cuerpo), el cual se caracteriza por tener una geometría (una forma), una transformación (su posición) y unas propiedades de material (tal como el coeficiente de fricción). El segundo tipo es el ***Robot***, el cual está compuesto de múltiples cuerpos (como sus enlaces, por ejemplo) asi como también información de su cinemática, actuación, etc. Finalmente, un ***World*** (mundo) agrupa instancias de otras clases y las coloca en las posiciones correctas una con respecto a otras. Cada una de estas clases tiene su propio formato de archivo de datos, el cual *GraspIt!* puede cargar (pero que, con excepcion del Mundo, no puede guardar).

1. Los cuerpos

En un mundo de simulación de *GraspIt!* Hay dos tipos de cuerpos principales: cuerpos estáticos (también conocidos como obstáculos) y cuerpos dinámicos (tal como los enlaces de un robot y los objetos). Los objetos estáticos no participan en la dinámica, pero proveen superficies de colisión para los cuerpos dinámicos. Se debe notar que esta diferencia se aplica principalmente cuando se usa el motor de dinámicas; en otra circunstancia los cuerpos dinámicos también se pueden usar como cuerpos estáticos.

Cada cuerpo en *GraspIt!* tiene los siguientes datos asociados con él:

- **Geometría**: Esta describe la forma del cuerpo, la cual se guarda como una grafica de escena Inventor, un formato similar a *VRML*. Esta estructura puede tener nodos de formas puras tales como cubos, esferas, cilindros y conos, como también conjuntos de polígonos 2D que definen una superficie.

  Aunque *GraspIt!* (a través del gráfico de escena Inventor) puede mostrar todos estos tipos de geometrías, el sistema de deteccion de colisión solo trabaja con triángulos. Cuando es importado algún cuerpo a un mundo *GraspIt!*, éste es convertido a una estructura triangular (haga de cuenta una malla), y estos triángulos son usados para la detección de colisión y para encontrar los puntos de contacto.

  Cuando un cuerpo es cargado desde un archivo o es creado por el usuario, el origen de su sistema de coordenadas es el origen de su geometría. Esto puede ser complicado de manejar, ya que el origen de la geometría es, por su naturaleza, una cuestión bastante arbitraria. Para contrarrestar esto los cuerpos dinamicos también usan la noción del centro de masa. Sin embargo, el origen del sistema de coordenadas de un cuerpo es siempre el origen de su geometría. En estos archivos se asume que las unidades son los milimétros.

- **Material**: El material afecta la cantidad de fricción posible sobre los contactos del cuerpo. Para cada par de materiales se define un coeficiente de fricción. Cuando no se usa el motor de dinámicas, *GraspIt!* usa un coeficiente de fricción estático para todos los cuerpos y este coeficiente solo afecta los cálculos de la calidad del agarre, pero no el movimiento relativo

de los cuerpos. Durante una simulacion dinamica el coeficiente de friccion afecta el movimiento relativo de los cuerpos en contacto. Si la velocidad relativa en un punto de contacto excede un límite, configurado por defecto en 1mm/seg, se usa el coeficiente de fricción cinético.

- **Transformacion**: Cada cuerpo hace el seguimiento de la posición 3D y la orientación de su marco de referencia con relacion al sistema de coordenadas del mundo *GraspIt!*

- **Nombre**: El nombre del cuerpo se deriva de su nombre de archivo, excepto en el caso de los enlaces de un robot, los cuales se nombran usando el nombre del robot, y el numero de su cadena cinemática y numero de enlace.

  Los cuerpos dinámicos también tienen las siguientes propiedades:

- **Masa**: Ésta es expresada en gramos.

- **Centro de masa**: Ésta es una posición 3D expresada con relación al sistema de coordenadas del cuerpo. Existen dos usos principales para esta propiedad; el primero es para proporcionar un punto estable para los cálculos de calidad del agarre; el segundo es para ser usado como punto de referencia para la transformación entre fuerzas y pares en la dinámica. El centro de masa es más estable que el origen del sistema de coordenadas del cuerpo, el cual es arbitrario.

- **Tensor de inercia**: Éste es la matriz de distribución de masa 3x3 estandar. Se expresa con relación a un marco de coordenadas que está alineado con el sistema de coordenadas del cuerpo, pero con origen en el centro de masa. Cuando se guarda en un archivo, se multiplica por el factor de escala 1/masa. De esta manera, se pueden aplicar cambios en la masa cambiando solamente el valor de la masa, vista mas arriba.

- **Estado dinámico**: Dos valores, q y v, guardan la posición y la velocidad actuales del centro de masa del cuerpo con relacion a las coordenadas del mundo. q es expresado como un vector 7x1: los primeros tres valores son la posición, y los cuatro últimos son la rotación en la forma quaternion. v es expresado como un vector 6x1: los primeros tres valores son la velocidad lineal del cuerpo, y los últimos tres valores son la velocidad rotacional. Cuando la dinámica actualiza el estado de cada cuerpo, la transformada del cuerpo también se actualiza. Si se mueve un cuerpo en el modo estático, el valor de la posición del estado dinámico también se actualiza.

2. Los Archivos De Cuerpo

La versión 2.1, de *GraspIt!* emplea un formato *XML* para guardar todos sus datos. Para los cuerpos, se requieren dos tipos de información: la información específica de *GraspIt!*, tal como los parámetros descritos mas arriba, y su propia geometría. En el resto de este manual los

ejemplos que se decriben son relacionados con el modelo de la mano creado por el grupo de investigación en *Auto*mática Industrial de la Universidad del Cauca en el proyecto “prótesis robotica de una mano” [7]. Por razones de smplicidad a este modelo se le llamará como mano del grupo.

Cada cuerpo esta contenido en dos archivos:

- Un archivo .*XML* que contiene toda la información específica de *GraspIt!*, asi cmo un puntero al archivo que contiene la geometría.

- El archivo de geometría. Actualmente existen pocas opciones para los archivos de geometría:

- Formato *VRML* (.wrl), leído directamente a través de *Coin*.
- Formato inventor (.iv), también leído por *Coin*.
- formato *OFF* (.*OFF*), leído por el propio analizador sintáctico de *GraspIt!* Este formato fue añadido, principalmente, para cargar objetos directamente desde el Princeton Shape Benchmark (*PSB*) [8]; hasta ahora no se ha probado con otros archivos que no son parte del *PSB*.

El siguiente ejemplo es un archivo de la falange superior del dedo índice de la mano del grupo.

<?*XML* version="1.0" ?>
<root>
<material>metal</material>
<*mass*>100</*mass*>
<*cog*>0.00 0.0 0.0</*cog*>
<inertia_matrix>3540.1 0 0 0 3350.1 -42.614 0 -42.614 1056.9</inertia_matrix>
<geometry*File*>superior2.iv</geometry*File*>
</root>

La siguiente lista de etiquetas será utilizada por *GraspIt!* para describir el archivo de cuerpo:

- **<geometry*File*>** - esta es la única etiqueta obligatoria; contiene la ruta para cargar el archivo de geometría, con relacion al directorio en donde se colocó el archivo *XML*. Requiere además un tipo de especificador especial que puede ser uno de los siguientes:

- Inventor- para archivos WRML (.wrl) e Inventor (.iv).
- *OFF*- Para archivos *OFF* (.*OFF*).

- **<*material*>** - puede ser alguno de los materiales usados por *GraspIt!*. Si no se especifica, se asume que se trata de un material genérico con un coeficiente de fricción de rango medio.

- ***<youngs>*** - El módulo de Young del objeto, el cual está en MPa (megapascales). Éste puede ser usado para simular los contactos de cuerpos blandos y para calcular el par por fricción del contacto. Si no se especifica esta propiedad, *GraspIt!* asume que el cuerpo es rígido y usa el modelo de contacto rígido más común.

  Propiedades dinamicas:

- ***<mass>*** - la masa del cuerpo expresada en gramos.

- ***<cog>*** - centro de gravedad. El valor esta compuesto de 3 entradas que describen las coordenadas del centro de gravedad relativo al origen de la geometría.

- ***<inertia matrix>*** - el tensor de inercia del objeto representado por una matriz 3X3 como una lista de 9 números tipo double.

Nota: Si no se especifican alguno de los parámetros dinámicos, *GraspIt!* intentará calcularlos *auto*máticamente, basándose en la geometría del objeto y asumiendo una distribución de masa uniforme. Lo anterior funcionará bien en la mayoría de los casos, pero también puede producir algunas fallas. Para tener un completo control sobre los parámetros dinámicos del cuerpo, es mejor especificarlos explícitamente en el archivo. Para más ejemplos, ver los archivos de cuerpo incluidos en la distribucion, en *$GRASPIT!/models/objects.*

Los archivos de geometría con los formatos *VRML* (.wrl) e Inventor (.iv) pueden ser creados por diferentes plataformas software. Pero el formato mas común es el formato *VRML* ya que la mayoría del software de diseño 3D traen consigo la opción de guardar los diseños en este formato. En *Solid Edge* al igual que en *Matlab* se pueden crear diseños 3D y luego guardarlos en el formato *VRML* (.wrl). En solid edge, una vez realizado el diseño en 3D se debe guardar como una imagen en formato *.Wrl*. En el Toolbox de Realidad Virtual de Matlab se incluye V-Realm Builder como un editor 3D nativo. Con esta herramienta se pueden crear representaciones gráficas 3D y representar un estilo de árbol jerárquico de todos los elementos en el mundo virtual. Cada una de las piezas (dedos y palma) del modelo de la mano del grupo fue construido en Solid Edge y se guardaron como archivos de imagen con el formato (.wrl). Para cargar estos archivos en *GraspIt!* se deben eliminar primero todas las configuraciones con respecto a la cámara y otros que no correspondan a la forma del objeto, ya que estas no conciernen a la forma del objeto en si (por ejemplo, la posición y orientación de la cámara o la distancia focal de la misma). Si lo anterior no se hace, cuando *GraspIt!* carga el cuerpo (objeto) este podría salir con algún color diferente o si ya se encuentra algún cuerpo en el mundo de simulación este no será visible.

3. Robots

Un robot esta constituido de múltiples enlaces (*links*), conectados en cadenas cinemáticas. Un enlace es simplemente un cuerpo dinámico como se describió anteriormente. Un robot siempre

tiene un enlace base (llamado palma para manos) y una o más cadenas cinemáticas conectadas a este. Cada cadena, a su vez, está formada por una sucesión de enlaces, conectados por articulaciones. Para definir un robot se necesitan dos cosas: los archivos de Cuerpo para todos los enlaces que son parte del robot, más un archivo de configuración general del robot, el cual tiene toda la información cinemática y las referencias pertinentes hacia los archivos de cuerpo de los enlaces. A continuación se describirá el archivo de configuraron de un Robot.

Los archivos de configuracion de un robot pueden parecer desalentadores al principio, y al usarlos pueden parecer un poco molestos. Sin embargo, se puede utilizar uno de los robots que vienen incluidos en *GraspIt!* y utilizar los bloques para construir su propio robot. En la versión 2.1 de *GraspIt!* los archivos de los robots se guardan en un formato *XML*.

En general, un archivo de configuración de Robot contiene los siguientes datos:

- La Palma- este es simplemente un puntero dirigido al archivo de Cuerpo que contiene la palma.

- grados de Libertad- contenidos en la etiqueta *XML* <dof>. Esto se describirá con más detalle mas adelante.

- cadenas Cinemáticas- contenidos en la etiqueta *XML* <chain>. Esta propiedad se describirá con más detalle mas adelante.

  Un archivo de configuracion de Robot puede contener otras etiquetas opcionales. Estas incluyen cosas específicas tales como: la informacion sobre los denominados *eigengrasp*, etc. Estas se describirán con más detalle mas adelante.

  Para mostrar como es un archivo de configuracion de Robot, a continuación se describirá el archivo del modelo de la mano del grupo.

4. La etiqueta raiz y la palma

La primera etiqueta *XML* <robot>, encierra el contenido total del archivo. Esta etiqueta tiene tambien un atributo o tipo de propiedad especifico, el cual le dice a *GraspIt!* si es un robot genérico, una mano o una subclase particular de estas. Una mano debe usar el tipo “Hand”. En algunos casos, si un robot tiene características especiales o su propio algoritmo de cinemática inversa, es necesario usar una subclase de estos tipos genéricos, tales como “*Barrett” o “Puma560”.*

<robot type="Hand">
... (resto del archivo del robot)
</robot>

La siguiente etiqueta define el nombre del archivo para el enlace correspondiente a la palma:

<palm>palma.*XML*</palm>

5. Los grados de libertad

La siguiente información es sobre los DOF`s. Cada uno de los grados de libertad se describe mediante la etiqueta <dof>. Note que un DOF puede estar conectado a una o más articulaciones en las cadenas cinemáticas. Tambien note que el ORDEN en que aparece cada etiqueta <dof> en el archivo es importante: los dof,s seran referenciados mas tarde por su posicion en este orden.

Para cada DOF, la etiqueta *XML* contiene la siguiente información:

- Un **atributo** (o tipo de propiedad) para la etiqueta <dof>. Su valor es una letra que describe el tipo de DOF. Para la mano del grupo, todos los DOF`s son del tipo “rigid” (rígido) representados por la letra “r”. Éste es el tipo mas común de DOF en *GraspIt!*. Siempre se puede usar este tipo de DOF a menos que esté construyendo un robot con articulaciones acopladas (múltiples articulaciones conectadas a un solo DOF).

  Las demas propiedades de un DOF se almacenan en etiquetas hijo, y se listan a continuacion.

- ***<defaultVelocity>*** - La velocidad por defecto para el DOF durante una operación de *auto* agarre. Ésta se usa para predefinir el movimiento de cierre de la mano. Para manos antropomórficas, estas directrices predefinidas le dicen a *GraspIt!* como mover cada DOF durante un *auto*-agarre, o como “cerrar la mano”. Esto generalmente significa el movimiento de los DOF`s de las articulaciones proximal (MCP), media (PIP) y distal (DIP) en la direccion de “flexion”, y no un movimiento en el sentido de abduccion-adduccion (hacia los lados) de las mismas.

- ***<maxEffort>*** - La fuerza máxima que el DOF puede aplicar a cada articulación a que esté conectado. La unidad es N * 1.0e6 * mm para pares y N * 1.0e6 para fuerzas.

- ***<Kp>*** and ***<Kd>*** - Son los coeficientes Kp y Kd para el controlador PD de fuerza construido internamente para el DOF.

- ***<draggerScale>*** - La escala visual del manipulador que permite al usuario controlar el grado de libertad a través del ***GUI*** de *GraspIt!*

- Hay un número de etiquetas hijo, o sub-etiquetas, opcionales, dependiendo del tipo de grado de libertad. Para el DOF “rigid” no son necesarias más sub-etiquetas.

A continuacion se muestra un ejemplo de etiqueta <dof> del archivo Mano_Grupo.*XML*

```
<dof type="r">
<defaultVelocity>1.0</defaultVelocity>
<maxEffort>5.0e+9</maxEffort>
<Kp>1.0e+10</Kp>
<Kd>1.0e+7</Kd>
<draggerScale>10</draggerScale>
</dof>
```

6. Las cadenas cinematicas

Cada cadena cinemática se guarda en la etiqueta ***chain***. Esta etiqueta no tiene propiedades (atributos) y puede contener las siguientes sub-etiquetas.

- ***<transform>*** - La transformacion desde el origen de la palma (el cual es también el origen del robot) a la base de ésta cadena, la cual está donde está situada en la primera articulación de la cadena. Notese que una etiqueta <transform> puede tener un número no especificado de sub-etiquetas, cada una conteniendo una traslación, una rotación o ambas.

- ***<joint>*** - Corresponde a una articulación en la cadena cinemática. Esta etiqueta puede contener la siguiente información:

  Esta etiqueta tiene el atributo type que define el tipo de articulación. Esta puede ser “Revolute” o “Prismatic”.

  Como sub-etiquetas:

  Los parámetros Denavit-Hartenberg (D-H) (REF) de esta articulación,

- ***<d>***
- ***<a>***
- ***<alpha>***
- ***<theta>***
-

- Uno de los cuatro parámetros D-H debe estar conectado a un grado de libertad del robot. Para las articulaciones de revolución “revolute” este será <d> y para las articulaciones prismáticas “prismatic” será <a>. Por ejemplo, para describir que una articulación de revolución está conectada al grado de libertad quinto del robot, la etiqueta *XML* tendrá la forma <d>d5</d>.

- También se puede especificar una relación lineal entre el valor del DOF y el valor de la articulación (en grados para una articulacion de revolucion). Esto se hace de la forma d#*k+c (¡sin espacios!). Por ejemplo, para especificar que el valor de la articulación será un tercio del valor del segundo DOF, mas un *OFF*set de 30 grados, la etiqueta *XML* tendrá la forma: <d>d2*0.33+30.0</d>

- los otros 3 parámetros D-H tendrán valores fijos, especificados en su respectiva etiqueta *XML*. Por ejemplo <a>46. en la cual se asume que la siguiente articulacion estará a 46 unidades (mm) de la articulacion actual.
- ***<minValue>*** - El valor límite inferior de la articulación.
- ***<maxValue>*** - El valor límite superior de la articulación.
- son posibles otras sub-etiquetas, pero estas no se utilizan en el ejemplo de la mano del grupo. Pueden contener cosas como el coeficiente de fricción de la articulacion, rigidez, etc.

- ***<link>*** Corresponde a un enlace de la cadena

  Esta etiqueta tiene el atributo denominado "*dynamicJointType*" (tipo de junta dinámica). Este atributo nos dice como el enlace se conecta con el anterior. Los valores que puede tomar son los siguientes: "*Revolute*", "*Prismatic*", "*Universa*l", "*Ball*", o "*Fixed*". Dependiendo de cual se use, la cadena tomará algunas de las articulaciones de la lista anterior y las unirá para crear una conexión de éste tipo. Por ejemplo:

- Revolute: El enlace está conectado a uno anterior a través de una sola articulación de revolución.
- Universal: El enlace está conectado a uno anterior usando dos articulaciones de revolución, usualmente con ejes perpendiculares.
- Ball: El enlace está conectado a uno anterior usando tres articulaciones de revolución, usualmente con ejes perpendiculares.

  El valor de la etiqueta <link> es un puntero al archivo de Cuerpo que contiene el enlace.

  El siguiente es un ejemplo de una cadena cinemática del archivo Mano_Grupo.*XML*:

```
<chain>
<transform>
<fullTransform>
        (-0.0361362 -0.386356 -0.013152 +0.921548)[+10.8706 -0.1 -102.478]
</fullTransform>
</transform>
<joint type="Revolute">
<theta>d0+180</theta>
<d>-5</d>
<a>44</a>
<alpha>90</alpha>
```

<minValue>-70</minValue>
<maxValue>0</maxValue>
</joint>
<joint type="Revolute">
<theta>d1</theta>
<d>0</d>
<a>39
<alpha>0</alpha>
<minValue>0</minValue>
<maxValue>70</maxValue>
</joint>
<joint type="Revolute">
<theta>d2</theta>
<d>0</d>
<a>0
<alpha>0</alpha>
<minValue>0</minValue>
<maxValue>70</maxValue>
</joint>
<link dynamicJointType="Revolute">infpulgar.*XML*</link>
<link dynamicJointType="Revolute">mediapulg.*XML*</link>
<link dynamicJointType="Revolute">suppulg.*XML*</link>
</chain>

7. Worlds

A continuación se describe como los robots y cuerpos son agregados y eliminados de un mundo, asi como el formato de un archivo de mundo, el cual almacena el estado actual del mismo.

Cuando se abre *GraspIt!* se inicia con un mundo vacio. El usuario puede cargar un mundo previamente guardado seleccionando *File → Open*, o agregar elementos al mundo vacio. Para importar un obstáculo (un cuerpo estático) o un objeto (un cuerpo dinámico) se debe usar *File→ImportObstacle* o *File→ImportObject*, y es*cog*er el archivo de Cuerpo que se necesite. Note que cualquier archivo de Cuerpo (independientemente de si se refiere a un cuerpo estatico o a uno dinamico) puede ser cargado como un obstaculo (*GraspIt!* simplemente ignorara la dinamica). Sin embargo, cuando un archivo de cuerpo es importado como un Objeto, *GraspIt! auto*máticamente lo instancia como un cuerpo dinámico. Tratará de encontrar los parámetros dinámicos en el archivo de cuerpo y si no los encuentra se le asignarán por defecto. Tenga presente que al asignar valores por defecto los resultados pueden ser impredecibles.

Para eliminar un cuerpo, seleccionarlo, y presionar la tecla **supr**. Para remover un robot, primero se debe seleccionar el robot entero (dando doble-click en uno de sus eslabones cuando la herramienta de selección esta activa) y presionar la tecla **supr**.

Nota: Todo cuerpo o robot recien importado aparecera en el origen del mundo. Es conveniente mover los cuerpos existentes fuera del origen antes de importar uno nuevo. Si no lo hace, el cuerpo nuevo se sobrepone con el ya existente y sera necesario conmutar temporalmente la propiedad de colision a *OFF* para poder mover y separar los cuerpos.

Cuando se selecciona “Save” en el menú, *GraspIt!* guarda el estado actual del mundo, en un archivo que contiene un formato compatible con *XML*. Este archivo puede contener las siguientes etiquetas:

- ***<obstacle>*** - El cuerpo es cargado como un obstáculo.

  Contiene las siguientes sub-etiquetas:

- ***<filename>*** - es un puntero al archivo que contiene el cuerpo que será cargado como un obstáculo. La ruta es relativa a $*GRASPIT!*.
- ***<transform>*** - la posición y orientación del obstáculo en el mundo de simulación. Como siempre una etiqueta ***<transform>*** puede contener múltiples sub-etiquetas en donde se especifica una traslación, una rotación o ambas. Usualmente, en los archivos de mundo, las transformaciones son especificadas con una sola sub-etiqueta del tipo ***<fullTransform>,*** la cual contiene la transformación completa codificada como un Quaternion.

- ***<GraspableBody>*** - El cuerpo es cargado como un Cuerpo que se puede agarrar. Es idéntica a la etiqueta <obstacle>, excepto que *GraspIt!* lo cargará como un cuerpo que se puede agarrar, inicializando sus propiedades dinámicas y haciéndolo parte de los cálculos dinámicos.

- ***<robot>*** - se carga a un Robot

  Contiene las siguientes sub-etiquetas:

- ***<filename>*** - un puntero al archivo *XML* de Robot. La ruta es relativa a $*GRASPIT!*.
- ***<dofValues>*** - Un atributo tipo String que guarda los valores de todos los DOF`s del Robot. Note que este podría representar un solo número por DOF, o más información, dependiendo del tipo de DOF.
- ***<transform>*** - la posición y orientación del Robot en el mundo, como se describió en el caso del obstáculo.

- ***<connection>*** - Indica una conexión entre dos robots. Esto significa que un Robot es conectado al final de una cadena cinemática de otro Robot, tal como se conecta una mano al final del brazo de un robot.

Esta etiqueta contiene las siguientes sub-etiquetas:

- ***<parentRobot>*** - Es el índice del robot padre en el mundo, el cual está dado por el orden en el cual aparecen las etiquetas <robot> en el archivo de Mundo.
- ***<parentChain>*** - Es el número de la cadena cinemática del robot padre a la cual el otro robot es conectado.
- ***<childRobot>*** - Es el índice del robot hijo en el mundo, el cual está dado por el orden en el cual aparecen las etiquetas <robot> en el archivo de Mundo.
- ***<mountFilename>*** (opcional) – Especifica un cuerpo que es, opcionalmente, usado como una pieza de montaje entre dos robots.
- ***<transform>*** la transformacion del *OFF*set constante entre el último enlace de la cadena cinemática del robot padre y el enlace base del robot hijo.

- ***<camera>*** (opcional)- especifica la posición, orientación y punto focal de la cámara virtual.

C.3.8 Detección De Colision Y Contacto

Un componente clave del motor de *GraspIt!* es la detección de colisión y contacto. Cuando la colision está activada, cada vez que se interactúa con un mundo en *GraspIt!*, ya sea moviendo un objeto o usando los manipuladores de las articulaciones de un robot, el motor de *GraspIt!* impedira cualquier movimiento entre dos objetos en colision y lo detendra en el momento del contacto. Los puntos del contacto son entonces marcados con indicadores de contacto. Aunque *GraspIt!* (a través del gráfico de escena Inventor) puede mostrar todos los tipos de geometrías, el sistema de detección de colisión solo trabaja con triángulos. Cuando un cuerpo geométrico es cargado, *GraspIt!* usará el gráfico de escena Inventor para dividir la superficie en triangulos, los cuales son usados para construir el modelo del cuerpo para el sistema de detección de colisión. Si se desea cambiar el umbral se debe dirigir al código. Su valor está dado en el miembro estático ***THRESHOLD*** de la clase ***Contact****.*

1. Colisión

En general, comprobar la colisión entre dos objetos es muy costoso a nivel computacional. Uno de los métodos más usados para minimizar el coste computacional es comprobar previamente la colisión entre los volúmenes envolventes de los objetos, de manera que solo si éstos colisionan se realice un test de colisión más detallado. Un volumen envolvente (Bounding Volume **BV**) para un conjunto de objetos es un volumen cerrado que contiene completamente la unión de todos los objetos del conjunto. Para que sea útil, un volumen envolvente debe cumplir las siguientes propiedades:

- Debe ajustarse tanto como sea posible al objeto.

- El cálculo del volumen envolvente debe ser poco costoso, sobre todo si éste debe re-calcularse con cierta regularidad.
- Debe representarse utilizando poca cantidad de memoria.
- El volumen envolvente debe poder rotarse y transformarse fácilmente.
- El cálculo de colisión entre volúmenes envolventes debe ser poco costoso en comparación con el test de colisión entre objetos.

Los volúmenes envolventes más representativos son las esferas envolventes (BS), las cajas envolventes alineadas con los ejes (AABB), las cajas envolventes orientadas (OBB), y la envolvente convexa (CB) (utilizándose normalmente las siglas en inglés de estos términos). En la figura C.3.2 se pueden ver estos volúmenes envolventes, sobre objetos en 2 dimensiones, y cómo se ajustan unos mejor que otros a un objeto determinado.

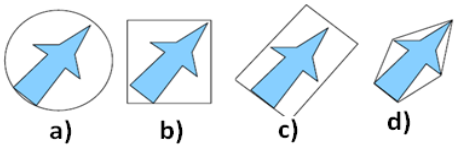


Figura C.3.2. Tipos de volúmenes envolventes.

a) Esfera envolvente. b) Caja envolvente alineada con los ejes. c) Caja envolvente orientada. d) Envolvente convexa. [9]

Con una relación de aspecto y factor de escala constantes de los objetos encerrados por los volúmenes envolventes, el número de intersecciones entre los volúmenes envolventes es casi el mismo que el número de intersecciones entre los objetos [10]. Aunque el test de colisión entre volúmenes envolventes añade un pequeño coste a la detección de colisión, en ocasiones no se produce colisión, y el resultado neto es que el costo computacional se hace mucho menor. El sistema usado en *GraspIt!* para la detección de colisión es el **PQP** (Proximity Query Package) desarrollado por *Larsen* y colegas [11]. Los volúmenes envolventes usados en esta técnica difieren de los mencionados más arriba y se generan al barrer un punto, una línea y un rectángulo con una esfera. Así, el barrido de un punto resulta en una esfera, el barrido de una línea resulta en un cilindro con capas hemisféricas en cada final, y el barrido de rectángulo resulta en una caja redondeada. Estos volúmenes son denominados, **PSS (**Point Swept Sphere), **LSS** (Line Swept Sphere) y **RSS** (Rectangle Swept Sphere) respectivamente y se muestran en la Figura C.3.3.

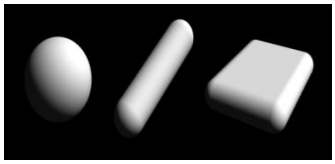


Figura C.3.3. Volúmenes envolventes generados por el barrido de una esfera. [11]
A la izquierda una esfera, en el centro un cilindro y a la derecha un rectángulo redondeado.

En el sistema de detección de colisión PQP los objetos deben ser modelados como un conjunto de polígonos, siendo utilizados los triángulos para obtener algoritmos más rápidos.

Cuando un cuerpo es cargado en *Grasp*It, los triángulos del modelo del objeto son pasados al sistema PQP en donde se convertirán en las hojas de un árbol jerárquico de volúmenes envolventes (BV's). Los árboles jerárquicos son creados de arriba a abajo. En el extremo superior del árbol, denominado raíz del árbol, el conjunto total de los triángulos que modelan al objeto es encerrado dentro de un volumen envolvente. Este volumen envolvente es orientado a lo largo de los ejes principales del conjunto, y dividido con un plano perpendicular al eje más largo del conjunto (Ver Figura C.3.4). Cada uno de los subgrupos creados es un hijo de la raíz del árbol, y estos a su vez son encerrados y subdivididos recursivamente hasta alcanzar a los triángulos individuales del modelo del objeto, que no pueden ser divididos. Para determinar si dos cuerpos colisionan, el sistema comienza con los dos nodos raíz y chequea si existe traslape. Si los volúmenes están separados entonces no hay colisión, en caso contrario el sistema examina el primer hijo y verifica su traslape con el nodo raíz del otro árbol. La jerarquía de las evaluaciones puede ser descrita por lo se denomina el árbol de prueba de los volúmenes envolventes (Bounding Volume Test Tree **BVTT**) [11].

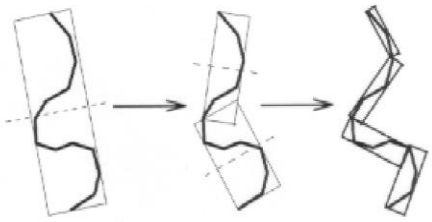


Figura C.3.4. Jerarquía de Volúmenes Envolventes. [9]

Un ejemplo simple puede observarse en la figura C.3.5. En esta figura se observa un árbol de prueba de los volúmenes envolventes (BVTT) creado a partir de dos jerarquías de volúmenes envolventes (Bounding Volume Hierarchy **BVH**) realizadas a partir de dos objetos. Los nodos del árbol de prueba BVTT son la secuencia recursiva de la evaluación del traslape entre los volúmenes envolventes del primer objeto y los volúmenes envolventes del segundo objeto. AX representa el nodo raíz del árbol de prueba BVTT y verifica el traslape entre los volúmenes envolventes en la raíz del árbol que representa a los dos objetos. Si estos se traslapan, la búsqueda de triángulos traslapados continúa a través de los otros nodos. BX y CX representan los dos hijos del nodo raíz del BVTT y son la evaluación entre el volumen del nodo raíz de un objeto traslapado y los dos hijos del nodo raíz del otro objeto. Luego se evalúa el hijo del nodo raíz que se traslapó con los hijos del nodo raíz del otro objeto y así sucesivamente hasta encontrar el conjunto de triángulos traslapados. Si en el momento de evaluar un par de volúmenes envolventes estos no se traslapan entonces el sub-árbol debajo de este nodo es podado de la búsqueda.

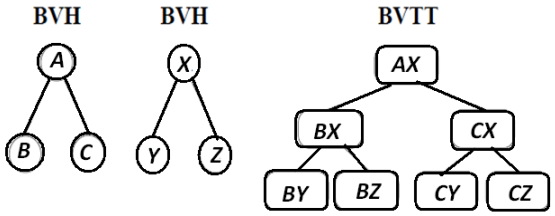


Figura C.3.5. Ejemplo de un árbol de prueba de volumen envolvente (BVTT). [11]

El tipo de volúmenes envolventes usado en el árbol de jerarquía determina la eficiencia del algoritmo, pero la selección de uno u otro depende de los requerimientos que el sistema deba cumplir. De hecho, en *GraspIt!* se crean dos tipos jerarquías, una usa los volúmenes envolventes OBB (Oriented Bounding Box) y la otra usa los volúmenes envolventes RSS (Rectangle Swept Sphere). La primera se usa para la detección de colisión y la segunda para la detección de proximidad.

En *GraspIt!* para determinar los puntos de contacto, si se detecta una colisión (ver figura C.3.6 a)), la posición de los cuerpos debe regresarse al instante donde ocurrió el primer contacto. Para encontrar este instante, *GraspIt!* regresa los objetos a sus localizaciones antes de la colisión y llama al sistema **PQP**, que en lugar de probar si existe colisión, realiza la prueba de los volúmenes envolventes hasta alcanzar hojas en los arboles (triángulos) y calcula la distancia mínima entre ellos. Este procedimiento se repite para todas las ramas posibles. Si la distancia es menor que la encontrada anteriormente, se guarda como la distancia mínima y guardan además los puntos en cada triángulo más cercanos entre sí. Si durante la prueba de una rama la distancia entre los dos volúmenes envolventes es mayor que la distancia mínima actual, el algoritmo no necesita descender más por esta rama.

Típicamente, durante una manipulación interactiva los enlaces de la mano se mueven solamente unos pocos milímetros entre actualizaciones. Al regresar a la actualización anterior a la colisión, la distancia mínima que retorna PQP debe estar en este rango (milímetros).

Debido a que la posición de la muñeca y la de las articulaciones son representadas con números de punto flotante de precisión fija, es imposible encontrar exactamente su valor en el momento del contacto. Por lo tanto en *GraspIt!* se define una región de contacto fina alrededor de la superficie de cada cuerpo, y si la distancia entre los dos cuerpos es menor a este umbral (denominado *umbral de contacto* y por defecto fijado en 0.1 mm) entonces se considerará que existe contacto entre ellos. Para mover el enlace de la mano a una distancia que se encuentre en el rango anterior se usa una técnica de búsqueda binaria [2] (ver figura C.3.6 b)).

[2] En la ciencia computacional, una búsqueda binara es un algoritmo para localizar la posición de un elemento que ha sido guardado en una lista. Este algoritmo inspecciona el elemento medio de la lista: si es igual entonces la posición fue encontrada; en caso contrario se busca en la lista de la mitad hacia arriba o de la mitad hacia bajo dependiendo de si el valor es más grande o más pequeño que elemento medio.

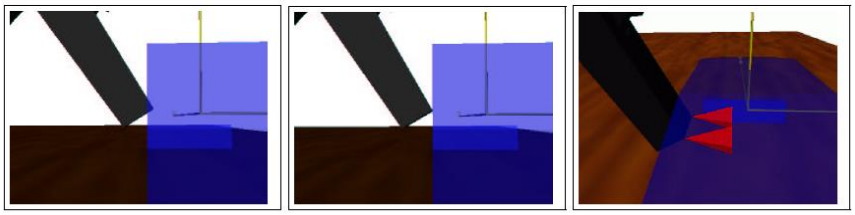


Figura C.3.6. Detección de colisión y proceso de detección del contacto
(a) se detecta la colisión de un enlace de la mano Barrett con un lado del teléfono. (b) se realiza una búsqueda para encontrar el ángulo de la articulación que hace que el enlace se encuentre a 0.1 mm del teléfono. (c) Se determina la geometría del contacto y se colocan los conos de fricción en los vértices que limitan la región de contacto (en este caso una línea). [6]

2. Contactos

Después de llevar los cuerpos a una distancia de contacto, el sistema debe determinar las regiones de contacto entre los dos cuerpos. Esto se hace usando una modificación del algoritmo de información de distancia **PQP** implementado en *GraspIt!* El proceso comienza buscando en el árbol de prueba **BVTT** los pares de triángulos que están separados una distancia menor al umbral de contacto. Para cada par encontrado, el sistema debe determinar la posible región de traslape entre ellos. Para hacer esto, se examina que tipo de contacto produce la distancia mínima entre los triángulos (ver figura C.3.7). Los contactos vértice-vértice y vértice-borde son considerados inestables y son ignorados. La región de contacto para un contacto borde-borde es un simple punto, y este es definido sobre cada triángulo como el punto más cercano al otro triangulo. Todos los otros pares topológicos incluyen una cara del triángulo, y en estos casos los vértices del otro triángulo son proyectados ortogonalmente sobre el plano que contiene la cara. El triangulo proyectado es usado para interceptar el primer triángulo, y los vértices de la región resultante son proyectados de regreso al triángulo al segundo triangulo. De los puntos que son proyectados de regreso, solamente se consideran los más cercanos a la cara del primer triangulo. Un contacto vértice-cara solamente tiene un punto más cercano, pero un contacto cara-cara puede tener una región de contacto limitada por al menos 6 vértices, todos equidistantes a la cara del primer triangulo. Cada uno de estos vértices es considerado un punto de contacto separado y es añadido con su correspondiente punto proyectado, al conjunto de puntos de contacto entre los dos cuerpos. Este conjunto crece según la cantidad de pares de triángulos examinados, pero solamente se tendrán en cuenta los contactos que son únicos ya que triángulos adyacentes sobre una superficie tendrán puntos de contacto idénticos con un triángulo sobre la otra superficie. Después de realizar la prueba BVTT, el sistema divide el conjunto de contactos en subconjuntos que tienen la misma normal de contacto. Si hay más de dos puntos de contacto en un sub-conjunto, el sistema chequea si todos estos puntos son co-lineales (como en un contacto borde-cara). Si es así, remueve todos los contactos excepto los puntos finales de la línea. Si no son co-lineales (como en el caso de un contacto cara-cara), el

sistema encuentra la envoltura convexa plana del conjunto de contactos y remueve los puntos de contacto interiores. Estos puntos de contacto interiores no afectan a los mecanismos de agarre y en la mayoría de los casos son ignorados [6]. Una vez ya se tiene el conjunto final de contactos entre los cuerpos, el sistema dibuja un cono de fricción (de color rojo) para cada punto de contacto, el cual sirve para marcar visualmente la posición del contacto y su normal (ver figura C.3.6 c)). El ancho de este cono identifica que tan grande es la fuerza de fricción producida por la fuerza aplicada a lo largo de la normal del contacto.

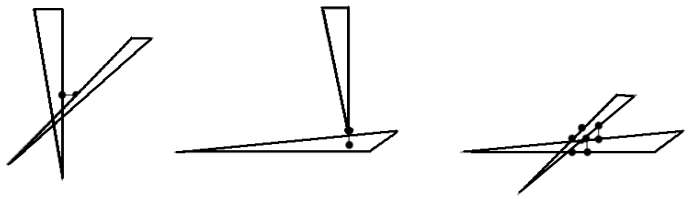


Figura C.3.7. Algunas de las posibles regiones de contacto [6]

La región es entre un par de triángulos que realmente no se tocan pero que están a una distancia menor que el umbral de contacto (no esta a escala). Izquierda: un contacto borde-borde es ubicado usando los puntos más cercanos sobre los dos triángulos. Medio: un contacto vértice-cara el cual es definido por el vértice más cercano sobre un triángulo y su punto proyectado sobre la cara del otro triangulo. Derecha: un contacto cara-cara el cual tendrá entre 3 y 6 puntos de contacto (en la figura 4 puntos).

En *GraspIt!* un contacto es definido como el punto en el cual dos cuerpos están separados por una distancia menor que el umbral de contacto, pero No se interpenetran. El motor de detección de colisión de *GraspIt!* localiza estos puntos y realiza algunas operaciones que se explican mas abajo. Un contacto en *GraspIt!*, definido en la clase **Contact**, encapsula la siguiente información:

- La localización del contacto en cada uno de los dos cuerpos (los puntos sobre los dos cuerpos que estan separados por menos que el umbral de contacto).

- La normal del contacto (definida como el vector normalizado entre los dos puntos mencionados anteriormente).

- El espacio *Wrench* del contacto: El espacio de fuerzas y pares que pueden ser aplicados al contacto. Este es un concepto crucial, ya que es la base de muchos cálculos de calidad de agarre y de las simulaciones dinámicas. En el caso más simple, que es el caso de la fricción de Coulomb [12], este espacio es un cono 3D.

- Un marcador visual que muestra la posición del contacto en el ***GUI*** de *GraspIt!* En el caso de contactos en cuerpos rígidos, se mostrará un cono de fricción alineado con la normal del contacto.

Se debe notar que en *GraspIt!* se tratan a los contactos como **puntos**. La razón es que la geometría en *GraspIt!* nunca se deforma, así que nunca se podrán calcular explícitamente **áreas** de contacto. Sin embargo, hay dos aspectos importantes. Primero, dos cuerpos rígidos pueden ser localmente similares de modo que más de un punto se encuentren dentro del umbral de contacto. Si esto pasa, la detección de colisión retornará muchos puntos de contacto en un área pequeña. En este caso, todos los puntos de contacto dentro del area son eliminados, manteniendo solamente los contactos que están en el contorno del área (la frontera de la envoltura convexa[3] de la ubicación de los contactos). De acuerdo con la teoria de contactos sobre la que descansa, esto no afectará los cálculos de calidad del agarre, en cambio reducirá el tiempo de cálculo ya que se reduce el número de contactos.

Segundo, *GraspIt!* puede simular algunos efectos de cuerpos blandos en contacto sin calcular explícitamente la deformación. Esto se hace usando una versión diferente del espacio *Wrench* del contacto.

C.3.9 Implementacion Del Software

Para muchos algoritmos, el motor de colisión es un obstáculo computacional. Es muy importante tener un motor eficiente, pero también se requiere que sea muy robusto y que trabaje bien sobre mallas de triángulos (en oposición a las primitivas analíticas). La versión *GPL* de *GraspIt!* trae su propia implementación de detección de colisión, usando varios métodos comunes de jerarquia de cubos envolventes. Sin embargo, la detección rápida de colisión es un área de investigación de por sí, y es seguro que existen muchas maneras de mejorar esta implementación.

Desde una perspectiva software, se construyeron librerías de detección de colisión y contacto para que sean lo más modulares posible. Esto permite cambiar la detección de colisión actual por otra con mejores características o según requiera el usuario. Para aquel interesado, debe examinar la clase *CollisionInterface* y sus subclases.

C.3.10 Medidas De Calidad Del Agarre

El proposito de esta seccion es proporcionar una guia practica de como acceder a las herramientas de calidad del agarre en *GraspIt!*. Se debe tener en cuenta que las medidas de calidad del agarre son un área activa de investigación; para un exámen de sus aspectos teoricos por favor remitirse a la seccion de publicaciones del manual *GraspIt!*

[3]En matemática se define la envoltura convexa de un conjunto de puntos *X* de dimensión n como la intersección de todos los conjuntos convexos que contienen a *X*.

En *GraspIt!* un agarre esta completamente definido por un objeto y un conjunto de contactos, presumiblemente creados por una mano. En el contexto del cálculo de la calidad del agarre, la postura de la mano, o su posición relativa al objeto no son importantes; todo lo que importa es la posición de los contactos. La interface para todos los cálculos de calidad es la clase ***Grasp***, la cual contiene los contactos entre una mano y un objeto. Sin embargo, una instancia de esta clase nunca necesita saber sobre los ángulos de las articulaciones, etc.

Para crear un agarre se debe cargar una mano y un objeto dentro de un mundo de simulación, colocarlos en forma adecuada la una con especto al otro y cambiar los valores de las articulaciones de la mano hasta formar múltiples contactos. Para comprender mejor este proceso, cargar el archivo de mundo ***dlr flask.XML*** que viene con la distribución de *GraspIt!* y usar ***Grasp → Auto*** *Grasp* de los comandos del menú. Esto da como resultado múltiples contactos con todos los dedos y la palma.

C.3.11 Dos Medidas De Calidad Bajo El Espacio *Wrench* Del Agarre

Todas las medidas de calidad del agarre, calculadas en *GraspIt!*, son basadas en la construcción del Espacio *Wrench* del agarre (GWS) a partir de los Espacios *Wrench* individuales de todos los contactos [13]. Exsisten múltiples maneras de construir un GWS y por lo tanto múltiples maneras de calcular una medida de calidad. En *GraspIt!* una Medida de Calidad (QM) es más que simple un número - es un objeto que está siempre presente desde el momento que se crea hasta el momento en que se destruye. En cualquier momento, se puede pedir a la **QM** que se actualice y proporcione el valor de calidad calculado. Se puede acceder a esta funcionalidad a través de ***Grasp*** → **Quality** measures que se encuentra en el menú. La ventana de Quality measures permitirá:

- **Crear** una nueva medida de calidad, y asignarle:
- un nombre
- un tipo de Medida de Calidad
- un tipo de GWS.
- **Editar** (cambiar el nombre o tipo) una medida de calidad ya construida.

  Una vez se ha construido una QM, la ventana principal de *GraspIt!* mostrará el tipo, asi como el valor calculado más reciente, el cual se mostrará en el espacio asignado en la esquina de abajo-izquierda del GUI.

  Se debe notar que este diálogo no dice nada de la construcción del GWS. Siempre que una QM necesita un tipo particular de GWS, esta se construye. Sin embargo, múltiples QM`s pueden compartir un GWS: si una QM requiere un tipo de GWS que ya existe, será re-direccionada a ese GWS en vez de construir uno nuevo. *GraspIt!* elimina un GWS que no es utilizado por ninguna QM.

Desde un punto de vista de código, las medidas de calidad se organizan de la siguiente manera:

- La clase ***Grasp*** está al tanto de la mano, el objeto y los contactos entre estos.

- Las implementaciones de la interface ***QualityMeasure*** pueden calcular las medidas de calidad y suministrar los resultados. Cualquier implementación de ***QualityMeasure*** necesita una instancia de una clase ***GWS*** (pero la misma instancia de la clase ***GWS*** puede ser compartida por múltiples ***QualityMeasures***).

- Las implementaciones de **GWS** realmente construyen la *GWS* a partir del conjunto de contactos. Usualmente esto se hace calculando la envoltura convexa de los Espacios de *Wrench* de Contacto individuales empleando *QHull*, una excelente librería para construir envolturas convexas de dimension alta. Construir un *GWS* es un proceso bastante costoso (especialmente cuando están involucrados muchos contactos) y un cuello de botella para muchos algoritmos en *GraspIt!*

En general, un GWS es un poliedro convexo de 6 dimensiones. Como tal, no puede ser visualizado directamente. Sin embargo, *GraspIt!* permite ver una proyección 3D de un GWS. Par ello, usar del menú ***Grasp →Create GWS Projection***. Note que un GWS (y sus proyecciones) son independientes de las medidas de calidad y no se necesita una QM para visualizar un GWS. Sin embargo, si un GWS no contiene el origen (el agarre no tiene cierre de forma (*form-closure*)) o si el GWS es un objeto 6D degenerado, el proceso de construcción puede ser abortado y la visualización puede no mostrar nada. En general, los agarres con cierre de forma con un GWS 6D verdadero crean proyecciones 3D correctas.

C.3.12 Conceptos Generales Y Tipos De Planificadores

La planificación del agarre es una de las herramientas más útiles en *GraspIt!*. El punto principal de este proceso es la habilidad del sistema para evaluar rápidamente muchas posturas de manos, desde un punto de vista funcional (ej. a través de medidas de calidad del agarre). Usando un ambiente simulado le permite al usuario probar agarres mucho más rápido que en la vida real y también a bajo costo. Las medidas de calidad suministran informacion de los agarres, a menudo más que el solo resultado binario de éxito/fallo. En resumen, el concepto general es este: tratar con una gran cantidad de agarres y ver con cual trabajar. Por supuesto, hay un número infinito de posibles implementaciones, optimizaciones, refinamientos, etc. que pueden ser implementados a partir de esta idea simple. *GraspIt!* trae un par de planificadores de agarre, cada uno de estos es diferente, pero todos se fundamentan en el mismo concepto presentado anteriormente.

Los planificadores en *GraspIt!* estan agrupados en tres familias:

- El planificador basado en Primitivas, primitivas no en el sentido de antiguo, sino en el sentido de descomposiciones primitivas del objeto agarrado.

- El planificador *Eigengrasp*, el cual se basa en la reducción de la dimension del espacio de posturas de la mano.

- El planificador apoyado en Base de Datos, el cual se fundamenta en una gran base de datos de agarres pre-calculados para planear agarres de nuevos objetos.

1. Planificador del agarre basado en primitivas

El primer paso del proceso de planificación basado en primitivas es generar una versión simplificada de la geometría del objeto, que consiste de un conjunto de formas primitivas, tales como esferas, cilindros, conos y cajas, únicamente. La elección de las primitivas determina las estrategias a ser usadas para agarrar el objeto. En la figura C.3.8 se puede observar un ejemplo de un vaso de café que ha sido descompuesto en sus formas primitivas. Estas primitivas corresponden a un cilindro y una caja, las cuales aproximan la forma y tamaño de la copa y el asa (oreja del vaso de café)

Figura C.3.8. Modelo de un vaso de café y su representación en primitivas [14]

De acuerdo a la versión simplificada del objeto, el planificador genera un conjunto de posiciones iniciales de agarre. Cada posición inicial de agarre consiste de una postura de pre-agarre y de una posición y orientación de la palma de la mano. La postura de pre-agarre es la posición que toman las articulaciones de los dedos antes de realizar los movimientos que conllevan al agarre final de un objeto [15]. La postura de pre-agarre de la mano se determina de acuerdo a la representación del objeto en términos de sus primitivas. Por ejemplo lo mas seguro es que para un cilindro se utilice una postura de pre-agarre cilíndrica, para una esfera una postura de pre-agarre esférica, y así con todas las posturas de pre-agarre. En [14] se exponen todas las características de las posturas de pre-agarre que deben ser usadas para cada una de las primitivas simples que representan el objeto. En la figura C.3.9 se puede observar un ejemplo de la generación de posibles posiciones iniciales de agarre de primitivas simples. Las esferas pequeñas representan la posición inicial para el centro de la palma. Las flechas representan la orientación de la palma de la mano. La flecha mas larga describe la dirección de acercamiento del agarre (perpendicular a la cara de la palma), y la flecha más corta muestra la dirección del pulgar (siempre perpendicular a la dirección de acercamiento del agarre).

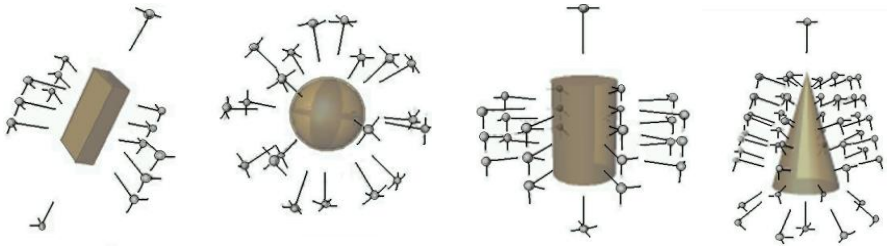


Figura C.3.9. Ejemplos de generación de agarres de primitivas simples [14]

En la figura C.3.9 se puede observar que sobre las primitivas simples se ha generado un cierto número de posiciones iniciales de agarre. En *GraspIt!* se definen 4 parámetros que determinan el número de posiciones iniciales que debe ser es*cog*ido de acuerdo a las primitivas del objeto. Para conocer más sobre las reglas empíricas que determinan la cantidad de posiciones iniciales del agarre, dirigirse a [14].

Después de generar las posiciones inciales, el planificador realiza cada agarre y lo evalúa. Para llevar a cabo el agarre, el planificador coloca la mano en una de las posiciones iniciales. Si existe colisión en esta posición, el agarre no es realizado y el sistema continúa con la siguiente posibilidad, es decir, con la siguiente posición inicial. Pero si lo anterior no ocurre, la mano es llevada a lo largo de la dirección de acercamiento del agarre hasta que se detenga el movimiento debido a un contacto con el objeto. Si los dedos no han contactado un obstáculo, ellos se cerraran alrededor del objeto hasta que los contactos o los límites de las articulaciones no permitan más movimiento. Si los dedos contactan un obstáculo el sistema regresa la mano una distancia pequeña del objeto e intenta de nuevo el agarre. Los intentos se repiten hasta que cada uno de los dedos alcance el objeto y el agarre pueda ser evaluado.

Este planificador esta disponible en el menú ***Grasp*** →**Planner**. Este tiene un par de restricciones: solo trabaja con la mano Barrett y solamente si el usuario también suministra la version basada en primitivas del objeto que va ser agarrado. Cuando el cuadro de diálogo del planificador está abierto, *GraspIt!* intentará cargar la versión basada en primitivas del objeto desde el directorio **$*GRASPIT!*/models/ objects/primitives**. Para crear un archivo de primitivas, ver los ejemplos en el directorio de primitivas incluido en la distribucion. Note que un archivo de primitivas incluye solamente esferas, cubos, cilindros y conos. Para mayor detalle consultar las publicaciones relevantes.

Los planificadores llevan consigo dos etapas. La ventana de diálogo del planificador tiene dos grupos, uno por cada etapa. La primera etapa (accesible via el grupo de botones de la izquierda) es para generar muchos pre-agarres para el objeto. Los pre-agarres son generados basados en la versión basada en primitivas del objeto. Se pueden generar muchos pre-agarres según sean los recursos computacionales y el tiempo que se tengan para probarlos. El número de pre-agarres generados es controlado por los factores de densidad. Se puede es*cog*er, o un factor de densidad maestro (muestreo *auto*mático) y permitir a *GraspIt!* hacer el resto, o es*cog*er, separadamente, las densidades de muestreo de las diferentes dimensiones. Alternativamente, se pueden pre-especificar los pre-agarres exactos que van a ser probados

cargandolos desde un archivo, lo cual es útil para propósitos de depuración. Una vez se tiene el conjunto de parámetros deseados se debe dar clic en el botón Generate para generar los pre-agarres.

La segunda etapa calcula los agarres que resultan de los pre-agarres es*cog*idos. Note que la ejecución del agarre se hace sobre el objeto actual (aunque los pre-agarres son generados sobre la versión primitiva simplificada). También se puede es*cog*er cual será la Medida de Calidad que será usada para clasificar estos agarres. Si ya existe una QM utilizable, ésta se puede seleccionar desde la lista desplegable. Si no, usar el botón New para crear una nueva QM. Una vez se ha seleccionado la medida de calidad deseada se está listo para probar todos los pre-agarres dando clic en el botón Test.

Una vez ha terminado la prueba, la mano regresará a su posición inicial y el botón Show se habilitará. Usar el botón Show para mostrar uno a uno los agarres encontrados y ordenados según su medida de calidad.

Importante: Si se desea, se puede visualizar el proceso de prueba marcando la ventana *Visualize*. Esto significa que el proceso de ejecutar todos los agarres será presentado en pantalla y se podrá ver la mano ejecutando cada uno de estos. Se debe marcar en la ventana Visualize antes de dar click en el boton Generate para que esta trabaje. La visualización hace mas completa la demostración, pero la presentacion en pantalla hace considerablemente mas lento el proceso de planificación.

Se ha comprobado que cuando la presentación en pantalla se deshabilita el cuello de botella computacional del planificador basado en primitivas es la detección de colisión y contacto.

2. Planificador basado en Eigen*Grasp*

Esta clase de planificadores se basa en el concepto de Eigen*Grasp* propuesto por *Ciocarlie* y colegas [16] y que fue explicado en detalle en la sección 3.3.5. Un planificador basado en Eigen*Grasp*s esta compuesto de tres cosas:

- Un conjunto de variables sobre las cuales buscar.
- Una función de calidad a optimizar sobre estas variables (usualmente de alguna manera relacionadas con la calidad del agarre)
- Un método de optimización (usualmente Simulated Annealing)

2.1 Eigen*Grasp*s

Los Eigen*Grasp*s definen un sub-espacio de un espacio dado de los DOF`s de la mano. Asumiendo que la mano tiene *d* DOF`s, cada eigen*Grasp* es un vector *d*-dimensional. Una base

que comprenda *b* Eigen*Grasp*s ortogonales puede definir un sub-espacio *b*-dimensional. Adicionalmente, este sub-espacio necesita un origen, el cual es también un vector *d*-dimensional, permitiendo así las siguientes opciones:

- Definir una postura de la mano usando las *b* amplitudes de los eigen*Grasp*s (en posición a los *d* DOF`s).

- Encontrar la proyección en el sub-espacio Eigen*Grasp* de una postura de mano, la cual no está necesariamente en este sub-espacio.

  El termino "Eigen*Grasp*" es un termino creado por los creadores de *GraspIt!*. En la literatura (particularmente en relacion con la mano humana) el mismo concepto es referido como "sinergias de la mano".

2.2. El conjunto de variables

Las variables sirven para dos propósitos principales: ellas definen la posición y la postura de la mano. La postura de la mano (DOF´s intrínsecos) es definida por las amplitudes Eigen*Grasp* y la posición de la mano (DOF´s extrínsecos) es parametrizada usando seis variables: tres para traslación y tres para rotación.

2.3. La función de calidad

La función de calidad es definida en términos de una función de energía, y toma la forma:

$$E = f(a, w) \quad \text{(C.3.1)}$$

En donde $a \in \mathcal{R}^{\boldsymbol{b}}$ es el vector de amplitudes para los b Eigen*Grasp*s, y $w \in \mathcal{R}^{\boldsymbol{6}}$ contiene la posición y orientación de la mano. La función de energía que se propone intenta llevar un número de puntos pre-seleccionados (sobre la mano robótica) en ponerse en contacto con el objeto (ej. ver figura C.3.10). La pre-selección de estos puntos de contacto se hace de forma analítica o empírica.

Figura C.3.10. Ubicación de los puntos de contacto deseados para las manos: DLR, Robonaut y la humana. [16].

La función de energía debe estar relacionada con la calidad del agarre. En el capitulo 6 se estudiaron dos medidas de calidad del agarre. Estas medidas de calidad se hallaban a partir de los contactos generados entre la mano y el objeto. Pero en este planificador los contactos que debería tener la mano con el objeto ya han sido pre-calculados, (ver figura C.3.10), de tal manera que cuando se calcule el Espacio *Wrench* del agarre (**GWS**), los *Wrenches* que pueden ser aplicados a cada contacto deseado se escalen dependiendo de la distancia entre el contacto deseado y la superficie del objeto. Si esta distancia es pequeña, el contacto tendrá una contribución positiva para el agarre y éste será guardado como un agarre de alta calidad. Si por el contrario, los contactos deseados están lejos de la superficie del objeto, estos no afectarán significativamente la medida de calidad del agarre. Una vez sea calculada la medida de calidad del agarre, esta influirá de forma negativa en la función de energía, es decir, que si la medida de calidad crece la función de energía disminuye y viceversa.

2.4. El método de optimización

En general lo que se quiere es que la función de energía sea mínima cuando la mano envuelva al objeto generando una gran área de contacto. El algoritmo de optimización usado es el **Simulated Annealing** [17]. Su naturaleza estocástica lo hace una buena elección ya que nuevos estados (agarres) son generados como vecinos aleatorios del estado actual y no es necesario calcular el gradiente de la función de energía, además, el algoritmo trabaja bien en funciones no lineales. Este algoritmo encuentra estados que no solamente ponen a la mano en contacto con el objeto, sino también crean posturas de agarre estables. Para manos con capacidades de agarre, si se considera una variable por cada DOF, el espacio de búsqueda es altamente dimensional haciendo al algoritmo de búsqueda ineficiente. Es por esto que muchos de estos algoritmos de búsqueda solamente trabajan bien en el espacio Eigen*Grasp*s, donde las variables intrínsecas son las amplitudes Eigen*Grasp*s. Sin embargo, no existen restricciones para que estos planificadores corran en el espacio compuesto por todos los DOF`s. De hecho, si se carga el modelo de una mano que no tiene definido los Eigen*Grasp*s, *GraspIt! auto*máticamente definirá el conjunto Eigen*Grasp* “identidad”, en donde un Eigen*Grasp* corresponde a cada DOF. En estas circunstancias, todos los planificadores basados en Eigen*Grasp*s corren de la misma manera que si se hubiesen definido los Eigen*Grasp*s.
En muchos casos el planeador en el espacio reducido comprendido por solo dos Eigen*Grasp*s no da como resultado una postura en donde la mano robótica envuelve perfectamente la superficie del objeto. Sin embargo, estas posturas frecuentemente pueden terminar en un agarre estable usando simples heurísticas [16]. Una heurística simple involucra cerrar cada dedo hasta que contacte con el objeto o con otro dedo.

En general, la planificación basada en Eigen*Grasp*s puede ser vista como la búsqueda de múltiples variables, o como un problema de optimización para una función altamente dimensional. Las variables (relacionadas con los DOF´s) definen el agarre y la función optimizada es la calidad del agarre.

Esta distribución de *GraspIt!* incluye información de los Eigen*Grasp*s para muchas manos. Para la mano humana, se proporcionan direcciones Eigen*Grasp* acordes con las descubiertas por Santello M. [18]. El modelo de la mano usado en esta referencia tenia 16 DOF. Por lo tanto, la unica version de la mano humana con 16 DOF`s que se encuentra en *GraspIt!* contiene todos los 6 Eigen*Grasp*s descubiertos en aquel estudio. La versión de 20 DOF tiene solamente 2 Eigen*Grasp*s dominantes, y ya que se realizó un mapeo empírico entre 16 DOF`s y 20 DOF`s, estos Eigen*Grasp*s pueden no ser tan exactos como los suministrados con la versión de 16DOF`s.

Tres manos mas, la Robonaut, la DLR y la Barrett, tienen información Eigen*Grasp* predefinida en esta version de *GraspIt!*. Para los modelos antropomórficos (Robonaunt y DLR) se ha realizado un mapeo empírico de los dos Eigen*Grasp* dominantes de la mano humana, para adaptarlos a la cinemática de la mano robótica..

2.5 Carga de la información de los eigen*Grasp*s

Para cualquier modelo de mano, la información eigen*Grasp* puede ser definida usando un archivo de texto el cual es cargado al mismo tiempo que el archivo de configuración de la mano. Usualmente, los archivos de información eigen*Grasp* son colocados junto con el resto de información que define un robot, tales como el archivo de configuración o los archivos de las geometrías de los enlaces. A continuación se mostrará el sub-espacio Eigen*Grasp* de dos dimensiones (b = 2) que fue creado para la mano del grupo. Este sub-espacio podrá encontrarse en el archivo **mano-grupo.egr**. Que será suministrado en la carpeta que contiene toda la configuración de la mano.

- La primera línea contiene la palabra **DIMENSIONS** seguida por el numero de DOF`s de la mano.

- Cada eigen*Grasp* empieza con la palabra clave **EG**. En la siguiente línea se encuentra el eigenvalue (valor del eigen*Grasp*) asociado con este eigen*Grasp* en particular. Por el momento este número no se usa en ninguna parte en el codigo, pero puede tener alguna utilidad en el futuro. Finalmente, en la siguiente línea se encuentra el vector d-dimensional que define el *EigenGrasp*.

- El número de Eigen*Grasp*s son arbitrarios según criterios del usuario. En este ejemplo se define un sub-espacio de 2 dimensiones con dos *EigenGrasp*s.

- El origen del sub-espacio se define exactamente como un eigen*Grasp*, pero esta precedido por la palabra **ORIGIN**. Si no se encuentra un origen en el archivo, el sistema usará un origen predefinido del sub-espacio: para cada DOF, el origen del sub-espacio se asume estar en (maxVal-minVal)/2.

- La información de la normalización es opcional. Si se desea, se puede definir como un eigen*Grasp*, precedida por la palabra **NORM**. Si esta información no se encuentra en el archivo, no se utilizará ninguna normalización.

DIMENSIONS 4

EG
0.51
1.0 0.0 0.0 0.0

EG
0.25
0.0 1.0 1.0 1.0

ORIGIN
0.0000
1.13 0.79 0.79 0.79

NORM
0.0000
1.57 1.22 1.22 1.22

Cuando se carga un robot, *GraspIt!* lee en el archivo de configuración del robot la información desde donde se cargara el archivo *EigenGrasp* (si existe). Primero encuentra la palabra ***EigenGrasp*s**, seguida, en la siguiente línea, por el archivo eigen*Grasp* a ser cargado. La ruta para el archivo Eigen*Grasp* es relativa al archivo de configuracion. Por ejemplo, toda esta información puede ser suministrada colocando las siguientes líneas en cualquier parte del archivo de configuración de la mano del grupo:

...
<*EigenGrasp*s>
eigen_mano_grupo/mano_tesis_eigen.egr</eigen*Grasp*s>
...

2.6 Utilizacion de los eigen*Grasp*s

Importar en el visor una mano robótica. Después usar el menú ***Grasp*** → **Eigen*Grasp* Interface**. Aparecerán dos ventanas: la ventana para ajustar la amplitud de los eigen*Grasp*s (unos controles deslizantes) y la ventana de opciones de los eigen*Grasp*s. Si no se ha cargado ninguna información desde un archivo, el sistema mostrará el conjunto de eigen*Grasp*s trivial, donde cada eigen*Grasp* corresponde a exactamente un DOF y el sub-espacio eigen*Grasp* es idéntico al espacio de DOF`s.

Al comenzar, la mano es colocada en una posición neutra (todas las amplitudes de los eigen*Grasp*s son iguales a 0). Moviendo cada deslizador se puede controlar la amplitud del eigen*Grasp* respectivo, es decir, se puede mover cada DOF tal como normalmente se haría en *GraspIt!* El resultado depende del modo de operación, definido por la casilla de verificación ***Rigid***.

Modo ***Rigid****:* En este modo solo se permiten movimientos en el espacio Eigen*Grasp*. Si se coloca un DOF individual fuera de este espacio, la postura de la mano es proyectada de nuevo al espacio eigen*Grasp*.

Modo ***non-rigid****:* El movimiento es libre. Cuando se cambia un DOF individual, los deslizadores de los eigen*Grasp*s muestran, tambien en este modo, las amplitudes eigen*Grasp* de la proyección en el espacio eigen*Grasp*. Sin embargo, se permite que la postura de la mano abandone el espacio eigen*Grasp*.

La detección de colisión **no funciona** mientras se usa la interface de eigen*Grasp*.

Cuando se mueven los deslizadores eigen*Grasp*, el movimiento es detenido tan pronto como el DOF alcanza el límite de su articulación. Alternativamente, puede ser posible bloquear solo el DOF detenido y permitir el movimiento de los otros DOF`s, pero sacaria la mano fuera del espacio eigen*Grasp*. Para mas detalles ver el código ***eigengrasp.h*****.**

2.7 Planificación del agarre en el espacio Eigen*Grasp*

A continuación se describirán los planificadores de agarre que corren en el espacio Eigen*Grasp*. Desafortunadamente, ellos no comparten la misma estructura del planificador basado en primitivas descrito anteriormente; aunque desde un punto de vista de ingeniería de software de manera similar comparten el mismo concepto de intentar con muchos agarres.

En esta familia de planificadores, la tarea de la planificación es vista como la búsqueda sobre múltiples variables, o como un problema de optimización para una función altamente dimensional. Las variables definen el agarre y la función optimizada es la calidad del agarre. En general, para definir un agarre se utilizan dos conjuntos de variables: las variables intrínsecas (para definir los DOF`s de la mano) y las variables extrínsecas para definir la posición de la mano con respecto al objeto objetivo.

Para manos con capacidades de agarre, si se considera una variable por cada DOF, el espacio de búsqueda es altamente dimensional haciéndola ineficiente.

y no permite una búsqueda eficiente. Es por esto que muchos de estos algoritmos solamente trabajan bien en el espacio eigen*Grasp*, donde las variables intrínsecas son las amplitudes eigen*Grasp*. Sin embargo, no existen restricciones para que estos planificadores corran en el

espacio compuesto por todos los DOF`s. De hecho, si se carga el modelo de una mano que no tiene definido los Eigen*Grasp*s, *GraspIt! auto*máticamente definirá el conjunto eigen*Grasp* "identidad", en donde un eigen*Grasp* corresponde a cada DOF. En estas circunstancias, todos los planificadores basados en eigen*Grasp*s corren de la misma manera que si se hubiesen definido los Eigen*Grasp*s.

Un planificador basado en Eigen*Grasp*s esta compuesto de tres cosas:

- Un conjunto de variables sobre las cuales buscar.
- Una función de calidad a optimizar sobre estas variables (usualmente de alguna manera relacionadas con la calidad del agarre)
- Un método de optimización (usualmente Simulated Annealing)

Para comenzar se debe cargar una mano y un objeto, y usar entonces el menú ***Grasp*** **→Eigen*Grasp* Planner**. Aparecerá una ventana de diálogo del Planificador Eigen*Grasp*. El lado izquierdo de la ventana es dedicado a las variables que son buscadas. El lado derecho es dedicado al proceso de planificación mismo, permitiendo es*cog*er uno de múltiples tipos de planificadores, ver detalles acerca de la ejecución de la planeación actual, etc.

2.8 Variables

Todas las variables de búsqueda son mostradas en la parte izquierda de la ventana de diálogo. Las variables sirven para dos propósitos principales: ellas definen la posición de la mano y la postura de la mano.

- La **Postura de la mano** es definida usando las amplitudes Eigen*Grasp*. Todas las variables son mostradas en la ventana de diálogo con el prefijo **EG**.

- La **Posición de la mano** puede ser parametrizada de maneras diferentes. La lista desplegable *Type*, ofrece una variedad de opciones de parametrización. Cuando en esta lista desplegable se cambia la selección, se puede observar que la lista de variables también cambia. Para una búsqueda exhaustiva completa, la opción por defecto es *Axis-angle*. En este caso la posición de la mano es parametrizada usando 6 variables: 3 para la traslación y 3 para rotación. Las otras maneras para especificar la posición de la mano son mas especializadas [16], [19], [20], [21].

En el código *GraspIt!*, la clase SearchState comprende todas las variables utilizadas en la búsqueda. Existe una casilla de verificación on, la cual puede ser usada para habilitar o deshabilitar la búsqueda sobre una variable particular. Si una variable es deshabilitada (su casilla no esta marcada), ésta no tendrá influencia en la búsqueda, y mantendrá su valor actual en todas las soluciones generadas. Para cada variable, existen otras opciones marcadas como

Input, Target, y Confidence. Estas opciones son utilizadas para muchas operaciones de planeación especializadas, usando entradas externas.

2.9 Energia de la busqueda

La energía de la búsqueda es una manera de verificar si un estado, encontrado durante la búsqueda, es realmente bueno. Puede también ser considerada una función de las variables de búsqueda.

La energía de la búsqueda más sencilla es aquella que ya ha sido utilizada en el planificador basado en primitivas: usa una medida de la calidad del agarre. Sin embargo, una medida de la calidad del agarre (QM) tradicional está definida solamente si la mano esta realmente en contacto con el objeto. Cuando se realiza la planificación en el espacio Eigen*Grasp* muchos estados de la mano estarán muy cercanos, pero no en contacto perfecto con el objeto. Una posibilidad podría ser usar la función *AutoGrasp* para cerrar la mano y calcular la QM para el agarre resultante. Sin embargo, este es un proceso lento y la idea principal es evaluar rápidamente muchas posturas de la mano. Como resultado, en *GraspIt!* se definen diferentes implementaciones del concepto de “energía de la búsqueda” que no requieren los contactos de la mano con el objeto.

Por defecto la función de energía es **Hand Contacts**. Esta formulación intenta llevar todos los enlaces del robot tan cerca como sea posible al objeto objetivo. Los cálculos de esta función de energía pueden ser altamente acelerados pre-especificando la ubicación de contactos “deseados” sobre cada enlace [20]. Es este el papel desempeñado por la casilla de verificación **Preset contacts**. Tales contactos pre-definidos pueden ser cargados desde un archivo. Para las manos ***Human y Barret***, los archivos de los contactos predefinidos son suministrados con la distribución de *GraspIt!*, Si cuando es marcada la casilla de verificación Preset contacs no ha sido cargado ningún archivo de contacto el sistema le preguntará al usuario que archivo desea cargar. Se debe notar que se puede especificar el archivo de los contactos pre-definidos en el archivo de configuración del robot y serán cargados junto con el robot. Desde el punto de vista del código, la locación de los contactos pre-definidos es cargada dentro una instancia de la clase ***Virtualcontact.***

Están disponibles otras formulaciones para la energía de la busqueda, pero muchas de ellas también incorporan la formulación Hand Contacs de alguna u otra manera.

2.10 Métodos de optimización

El principal método, que es la base de todos los planificadores Eigen*Grasp*, es Simulated Annealing (templado simulado) [17]. El tipo de planificación más básico es correr una búsqueda Simulated Annealing usando la función de energía Hands Contacts, sobre todas las variables Eigen*Grasp* más la posición de la mano parametrizada como angulo de los ejes.

Uno particularmente útil, del que también se da un ejemplo mas abajo, es el MultiThreaded Planner. Uno de los defectos de la función de energía Hands Contacts es que no garantiza cierre de forma: ésta encontrará estados en donde la mano se ajusta a la forma del objeto, pero no necesariamente con cierre de forma. Para contrarestar lo anterior, en el planificador MultiThreaded se corre un método de optimización exactamente como el anterior; sin embargo, siempre que encuentra un estado que tiene un buen valor para la función de energía Hands Contacts, se dispara un subproceso el cual buscará en esta region mas detalladamente, aplicando la función *AutoGrasp* y calculando la medida de calidad que resulta. El planificador MultiThreaded es la mejor opción si se trata de encontrar agarres, con cierre de forma, de un objeto arbitrario con una de las manos de *GraspIt!*

Desde el punto de vista de código, la clase abstracta pura EGPlanner es la base de todos estos planificadores: Esta tiene la habilidad para realizar los ciclos de iteracion, calcular la formulación de energía de la mano dada, guardar las mejores soluciones etc. La única cosa que esta no sabe como hacer es correr el algoritmo de optimización. Su clase hijo directo, el ***SimAnnPlanner*** combina ese marco con la optimización Simulated Annealing. Para más detalles es recomendable observar la clase ***EGPlanner*** y la clase ***SimAnnPlanner***.

2.11 Ejemplo Práctico:

- Cargar la mano *HumanHand*20DOF
- Cargar *flask*.iv
- Dirigirse al menú en *Grasp*→Eigen*Grasp* Planner
- Seleccionar Axis-angle como el tipo de espacio de búsqueda (Space Search)
- Seleccionar Hand Contacts como la formulación de Energia (Energy formulation)
- Marcar la casilla Preset contacts. Si pregunta que se cargue un archivo de contacto, se debe seleccionar all_18_contacts.vgr (este debe estar ubicado en $*GRASPIT!*/models/robots/*HumanHand*/virtual)
- Seleccionar Sim. Ann. como el tipo de planificador (Planner type)
- Configurar 70000 como el numero máximo de pasos (Max Steps)
- Dar clic en Init
- Dar clic en >
- Permitir correr al planificador hasta que finalice.
- Usar los botones Show Results (< Best >) para ver los resultados.

3. Planificador apoyado en base de datos

3.1. Base de datos de columbia para el agarre

La base de datos de Columbia para el agarre (CGDB) es un depósito de mas de 250,000 agarres realizados sobre aproximadamente 8,000 modelos de objetos 3D

Las características principales de la CGDB son las siguientes:

- El conjunto de objetos usado es el **Princeton Shape Benchmark (*PSB*)**[4] [8]**.** Este conjunto consiste de aproximadamente 1800 modelos de objetos 3D que pueden ser descargados libremente.
- Algunos de los objetos encontrados en el ***PSB*** son considerados como modelos de objetos que no son apropiados para experimentos de agarre, tal como insectos o aviones. Estos objetos son re-escalados para que su tamaño sea parecido al de un juguete. Para conseguir lo anterior, cada uno de los 1800 modelos es escalado usado 4 escalas diferentes, 0.75, 1.0, 1.25 y 1.5, donde 1.0 representa la escala original del objeto encontrado en el *PSB*. Por lo tanto, el CGDB puede usar aproximadamente 1,800*4= 7,200 modelos de objetos 3D.
- El CGDB contiene cierto número de agarres realizados sobre cada uno de estos modelos. Cada uno de estos garres es guardado con sus correspondientes medidas de calidad, forma de pre-agarre y puntos de contacto entre el objeto y la mano.
- Los agarres son realizados por 3 modelos de mano:
- La mano Barrett, asumiendo que los dedos estarán cubiertos de plástico
- La mano Barrett, asumiendo que los dedos estarán cubiertos de goma
- El modelo de la mano humana.

3.2. Planificación apoyada en base de datos

La planificación del agarre apoyada en base de datos trabaja encontrando objetos que tengan similitud geométrica con el objeto que se desea agarrar. A estos objetos se les conoce como objetos “vecinos”.

La versión del planeador, incluida en *GraspIt!*, funciona para modelos que únicamente se encuentran en la CGDB y como tal ya han pre-calculado información de sus vecinos. Lo anterior se debe a que en *GraspIt!* todavía no se han implementado las herramientas necesarias para encontrar la similitud geométrica entre nuevos objetos.

El proceso de planificación del agarre apoyado en base de datos puede ser explicado de manera más intuitiva por medio del siguiente algoritmo:

Requerimiento: agarrar el modelo del objeto α con la mano H, usando K vecinos y con un umbral de calidad τ.

$N \leftarrow$ Vecinos más cercanos de α.
$N = \{n_1 \ldots . n_K\}$ los *K* modelos de la CGDB mas similares en cuanto a forma al modelo de objeto a agarrar α.
$R \leftarrow \{\}$ {Los agarres con una medida de calidad por encima del umbral τ}

[4] ***PSB*****-** es una base de datos abierta al público que contiene modelos poligonales del *World Wide Web* y diferentes algoritmos de clasificación, que clasifican los modelos 3D según la función, forma y construcción.

for all $n_i \in N$ ***do***
Alinear α y n_i por medio de sus ejes principales
Co-localizar los centros de masa de α y n_i
$G \leftarrow$ Los pre-agarres pre-calculados sobre n_i.
for all $g \in G$ **do**
Transformar g al sistema de coordenadas de α.
Colocar H en la posición y configuración de g.
repeat
mover H hacia atrás, es decir, alejándose de α.
Until no exista colisión entre la palma de H y α
for all *dedo* ∈ dedos de H **do**
repeat
abrir dedo
until no exista colisión entre el *dedo y* α.
end for
repeat
mover H hacia adelante, es decir, acercándose a α.
Until exista contacto entre H y α
Cerrar todos los dedos de H.
If medida de calidad del agarre > τ. **then**
$R \leftarrow R \cup g$
end if
end for
end for
return R

4. El Motor De Dinámicas

GraspIt! tiene dos modos principales de operación. En el primero, el cual se ha asumido en todos los capitulos anteriores, el usuario interactúa directamente con el mundo de simulación moviendo objetos o moviendo los manipuladores de las articulaciones del robot. El sistema de detección de colisión deshabilita cualquier movimiento que lleve a dos cuerpos al estado de colisión, y detiene cualquier movimiento de este tipo en el punto de contacto. Nos referimos a este modo de operación como “operación estática”.

En la vida real, sin embargo, los objetos se mueven como resultado de las fuerzas externas y de contacto. En *GraspIt!* estos fenómenos son tomados en cuenta cuando se usa el motor de dinámicas. Este motor calcula las fuerzas de contacto que previenen la interpenetración, y las velocidades y aceleraciones de todos los cuerpos en respuesta a las fuerzas de contacto, las restricciones de las articulaciones y las fuerzas externas. El movimiento resultante de los cuerpos de la simulación completa de un proceso de agarre se obtiene paso a paso con un

procedimiento de integracion [22] (en oposicion a la sola evaluacion de la calidad de la postura final del agarre, como se hace en la operación estática).

5. Interacción Del Usuario En El Modo Dinámico

La principal diferencia entre el modo de operación dinámico y estático es la manera en como el usuario interactúa con el mundo de simulación. En la operación estática, el usuario mueve directamente los cuerpos y ajusta los valores de las articulaciones del robot. Nada de lo anterior es permitido en el modo dinámico. La única manera en que el usuario puede afectar el mundo de simulación es ajustando los valores deseados de los DOF`s del robot. Los controladores incorporados internamente para el robot se encargan del resto, simulando una verdadera operación dinámica. Los controladores de los DOF`s generan fuerzas motoras, y como resultado de éstas, las articulaciones del robot se mueven y potencialmente tocan otros objetos. Los cuerpos también se ven afectados por la gravedad y caerán fuera del mundo si no son sostenidos por un obstáculo (como un piso o una mesa). Los obstáculos son parte del motor de dinámicas solamente en el sentido de que proveen contactos para otros cuerpos, nunca se mueven.

Para iniciar o dar pausa al motor de dinámicas usar el botón ***Dynamics start*** en la barra de herramientas. Mientras el motor esta corriendo, se puede observar como avanza el reloj. Durante este tiempo también se puede observar como los cuerpos se mueven como resultado de la gravedad o como resultado de la fuerzas motoras de los DOF`s si los valores de éstos han sido ajustados.

Actualmente, la única manera del usuario especificar una posición deseada de un robot en el modo dinámico es usando la característica *AutoGrasp*. Para ello usar el menú ***Grasp→AutoGrasp*** justamente como se haría en el modo estático. Cuando el modo dinamico esta activo, esto ajustará los valores deseados de los DOF`s a sus valores máximos o mínimos, dependiendo de los valores de velocidad por defecto especificados en el archivo de configuración del robot. En el futuro se intentará proveer de una interfaz de usuario para fijar los valores deseados durante la ejecucion de la dinamica.

6. Controladores de DOF`s

Los controladores de DOF han probado ser muy difíciles de calibrar y sintonizar. En el momento, se implementan controladores PD y se intenta generar trayectorias con per*file*s de aceleración y velocidad suaves desde los valores de DOF actuales a los valores de DOF deseados. Para mas detalles ver la función ***DOF::PDPositionController(...)***. Estos controladores son muy sensibles a las ganancias proporcional y derivativa y fueron sintonizados empíricamente. Aún así, para simulaciones dinámicas complejas, frecuentemente se observan comportamientos erróneos. Se espera mejorar el mecanismo de control de los DOF en el futuro.

7. La Implementación

El motor de dinámicas trabaja por pasos de tiempo (de integracion). Para cada paso de integracion se realizan dos tareas:

- Mover todos los cuerpos dinámicos de acuerdo a las velocidades calculadas en el paso de integracion previo. Si ocurre una colisión, se interpola hacia atras hasta el momento exacto en que ocurrió el contacto. Se marcan todos los contactos en el mundo.

- Se formula resolver un Problema de Complementaridad Lineal (LCP, Linear Complementarity Problem), que da como resultado:

- Las fuerzas de contacto que impiden las inter-penetraciones.
- Las fuerzas articulares que mantienen las articulaciones en su lugar.
- Las velocidades de todos los cuerpos en el mundo como respuesta a las fuerzas descritas anteriormente, asi como también a las fuerzas externas (motoras, de gravedad, etc.).

Para mas detalles ver las funciones relacionadas con la dinámica en la clase ***World***.

El corazon del motor de dinámicas, y también su más importante esfuerzo computacional, es el solver del LCP. Para este se ha implementado una versión del algoritmo de Lemke, que se encuentra en el archivo ***dynamics.cpp***.

8. Bajo Desarrollo: Mejora De Dinámicas

Desafortunadamente, el motor de dinámicas muestra inestabilidades ocasionales, las cuales se trataran de mejorar. Estas inestabilidades pueden ser causadas por el solver LCP, por el paso de integración, o ambos. Al parecer, según expertos, el principal sospechoso es el LCP. En estos momentos se esta intentando cambiar el solver por uno mejor, basado en nuevos métodos propuestos en la literatura.

También se desea disminuir el tiempo de cálculo suministrando al LCP con un "*Warm start*" usando la solución del paso de integracion previo. Esto ha sido más difícil por el hecho de que los contactos son siempre destruidos y re-calculados entre cada paso de integracion, haciéndo más difícil hacer el seguimiento de las soluciones de pasos previos. Hemos puesto in place un mecanismo para "Heredar" los parámetros de contacto entre los pasos de tiempo, usted encontrará esto en la clase de contacto. Sin embargo, con el solver de LCP actual, no hemos estado capaces obtener una mejora usando esta información. La contacto "Herencia" ha sido dejada en el puesto (aunque dejar incapacitado por ahora) en la esperanza de que futuros solvers de LCP podría poder aprovecharlo sin embargo.

C.3.13 Acoplamiento De Las Articulaciones Y Manos Sub-Actuadas

En capítulos anteriores se han utilizado los términos “articulación” y “DOF” como si fueran la misma cosa y que un DOF corresponde siempre a una sola articulación. Esto no siempre es verdad en los modelos de manos reales, y tampoco siempre es cierto en *GraspIt!*. *GraspIt!* permite conectar múltiples articulaciones a un solo DOF (articulaciones acopladas) y permite soportar diferentes métodos de acoplamiento.

La mejor manera de concebir un DOF en *GraspIt!* es como un motor. Éste es el responsable de mover las articulaciones del robot, y es la única interfaz externa disponible del robot que logra mover las articulaciones. En el GUI de *GraspIt!*, los botones que permiten mover un robot no son realmente manipuladores de “articulaciones” sino manipuladores de DOF`s: al usuario se le presenta solo un manipulador por cada grado de libertad. Por ejemplo en el caso de la mano Barrett, se puede notar que existe solamente un manipulador por dedo, aun cuando cada dedo tiene dos articulaciones. Si se mueve un manipulador éste afectara a todas las articulaciones que están acopladas al mismo DOF.

Se dice que un robot es actuado completamente cuando existe un motor por cada articulación. Sin embargo, en muchos casos se tienen articulaciones acopladas, es decir, múltiples articulaciones conectadas a un solo DOF. En este caso, se dice que el robot es sub-actuado. El punto clave aquí es como este acoplamiento es implementado desde un punto de vista Hardware y si esto se puede replicar en software.

Un DOF en *GraspIt!* es responsable de decir como todas las articulaciones que están conectadas a él responden a los cambios en el DOF. Esta información se construye dentro de la clase DOF y sus implementaciones. Dependiendo de cómo se logra el acoplamiento de las articulaciones, un grado de libertad se comporta de manera diferente, es por eso que existen múltiples implementaciones. La pregunta mas importante es: ¿Cuando una de las articulaciones de un DOF no puede moverse (presumiblemente debido a algún contacto en sus enlaces) que pasa con las otras articulaciones conectadas al mismo DOF?

En el modo dinámico, un DOF es responsable de hacer cumplir el mismo modo de operación. Esto se hace vía las “restricciones del DOF” las cuales deben estar definidas en el solucionador **LCP** por el motor de dinámicas. Por ejemplo, un DOF con acoplamiento rígido debe asegurar que todos sus enlaces se muevan siempre a la misma velocidad. Finalmente, el DOF es también responsable de aplicar fuerzas a sus articulaciones (las cuales son transmitidas a los enlaces) en el modo dinámico. Ésta es la única manera de mover un robot en el modo dinámico. Además, las capacidades de generación de fuerzas también dependen de la implementación del DOF.

1. Acoplamiento Rígido

El tipo de acoplamiento más simple es el Rígido, implementado en la práctica a través de los mecanismos de transmisión con cables de acero. Esto significa que si una articulación de un DOF se detiene, todas las otras articulaciones del mismo DOF también se detienen. En el código, esto es implementado en la clase RigidDOF. Muchas manos en *GraspIt!* utilizan esta clase de sub-actuación como la Robonaunt, DLR, y la versión de la mano humana de 16 DOF. Para más detalles se recomienda ver la clase RigidDOF.

2. Transmisión Con Ruptura

Esta clase de acoplamiento permite a una articulación distal "separarse" y continuar cerrándose si ya se ha detenido un enlace proximal. El punto en el que se separa es marcado, y la junta proximal se re-engrana nuevamente solamente si el DOF regresa al punto de separación. Este es el tipo de transmisión construida en la mano Barrett. Para mas detalles se debe observar la clase ***BreakAwayDOF***.

C.3.14 Interfaz Con Matlab

En algunos computadores, el servidor ***TCP*** de *GraspIt!* presenta fallas de segmentación (segfault) al salir (debido a lo que parece ser un problema del ***socket*** de ***Qt*** que nosotros no hemos podido corregir). Por lo tanto, hemos comentado la línea de código que activa el servidor; si se quiere usar esta característica se debe activar esta línea. En el archivo **$*GRASPIT!*/src/main.cpp**, encuentre la línea correspondiente y quíte la declaración de comentario.

Cuando *GraspIt!* inicia, este inicializa un servidor TCP que acepta conexiones. Un protocolo de texto, que será descrito en documentaciones futuras, permite a programas externos emitir comandos y recolectar datos de simulación. Estos programas pueden estar corriendo en el mismo computador que *GraspIt!* o en un computador separado. Las funciones descritas en este documento permiten a MATLAB comunicarse con *GraspIt!*. Las funciones están implementadas como archivos MEX en lenguaje C. Cada una abre una conexión con el servidor de *GraspIt!* (el cual por ahora debe correr en la misma maquina), envia un comando de cambio o solicitud del estado del mundo, y cierra la conexión.

Para usar estas funciones, asegúrese de configurar el directorio de MATLAB a la carpeta "matlab" dentro de la carpeta de instalación raíz de *GraspIt!*. Escriba "help matlab" para obtener un resumen de las funciones disponibles, y escriba "help" y el nombre de la función para obtener información más detallada.

1. Comandos

computeNewVelocities – calcula la nueva velocidad de cada cuerpo dinámico.

getAverageContacts - retorna la posición y la fuerza de contacto promedio para uno o varios cuerpos.

getBodyName - retorna el nombre de uno o varios cuerpos.

getContacts – retorna todas las posiciones y fuerzas de contacto de uno o varios cuerpos.

getDOFVals – retorna las posiciones actuales de los grados de libertad de uno o varios robots.

getRobotName – retorna el nombre de uno o varios robots.

moveDOFs – mueve cinematicamente los DOFs de un robot a una posición deseada.

moveDynamicBodies – mueve cada cuerpo dinámico durante un paso de tiempo o hasta que haya contacto.

Render – instruye a *GraspIt!* para representar la escena.

setD*OFF*orces – configura las fuerzas que actúan en los grados de libertad de un robot o robots.

Comandos relacionados con cuerpos: getAveragecontacts, getBodyName, getContacts – estos comandos pueden ser usados en un conjunto especifico de cuerpos si estos son dados en un vector de índice de cuerpos, o si esto no es suministrado ellos operan sobre cada cuerpo actual definido en el mundo de *GraspIt!*. La función getBodyName puede ser usada sin argumentos para ver cual cuerpo corresponde a su respectivo índice. La posición del nombre del cuerpo en el array retornado es el índice del cuerpo en el mundo.

Comandos relacionados con robots: getDOFVals, getRobotName, setD*OFF*orces - estos comandos pueden ser usados en un conjunto especifico de robots si estos son dados en un vector de índice de robot, o si esto no es suministrado ellos operan sobre cada robot actualmente definido en el mundo de *GraspIt!*. La función getRobotName puede ser usada sin argumentos para ver cual robot corresponde a cual índice. La posición del nombre del robot en el array retornado es el índice del robot en el mundo.

La función moveDOFs requiere un solo índice de robot, por que *GraspIt!* no esta configurado cinematicamente para mover varios robots a la ves.

Comandos relacionados con dinámicas: moveDynamicBodies, computeNewVelocities – Estas dos funciones son usadas para avanzar la simulación dinámica por un paso de tiempo. El usuario suministra a moveDynamicBodies con un máximo paso de tiempo, y la función mueve cada cuerpo a lo largo de su trayectoria actual durante esa cantidad de tiempo. Sin embargo, si

un contacto ocurre durante ese periodo, la función mueve los cuerpos hasta ese punto, añade los nuevos contactos y retorna la cantidad de tiempo que ha pasado hasta el evento ocurrido. Un proceso similar ocurre cuando el límite de una articulación es alcanzado. El valor de tiempo debería ser pasado a el comando computeNewVelocities cuando esta es llamada. Este comando calcula para los impulsos que están actuando sobre cada cuerpo y la velocidad resultante de cada cuerpo dadas las restricciones actuales.

2. Un Paso De Tiempo

Un script que lleva a cabo una simulación dinámica contendrá un bucle, donde cada iteración del bucle avanza la simulación un paso de tiempo. Lo siguiente puede ser un posible ejemplo de script.

```
%el máximo paso de tiempo es comúnmente 2.5 milisegundos.
maxdt = 0.0025;
while 1
% avance de la simulación
h = moveDynamicBodies(maxdt);
computeNewVelocities(h);
% calculo de las fuerzas de control
%donde 6 9 12 son los índices de las puntas de los dedos del cuerpo
[CF,CP] = getAverageContacts([6; 9; 12]);

DP = getDOFVals;
DF = computeControlForces(DP,CF,CP);
%Esto debería ser suministrado por el usuario para que calcule las nuevas fuerzas de los DOF basado en las posiciones actuales de los DOF, las fuerzas de contacto y las posiciones.

setD*OFF*orces(DF);
%las fuerzas calculadas son aplicadas a los cuerpos y afectaran el próximo cálculo de velocidad.
end
```

C.4 INSERCIÓN DE UNA MANO ROBÓTICA EN *GraspIt!*

Para crear una mano en el simulador *GraspIt!* lo primero que se debe hacer es crear cada uno de los archivos de geometría de los enlaces (falanges de cada dedo y de la palma). Para crearlos existe una gran cantidad de software de diseño 3D en los cuales a los enlaces se les da una apariencia, color, brillo, tamaño, forma, transparencia etc. Es de vital importancia que el software en el cual se van diseñar los enlaces y la palma, pueda exportar los diseños en un

formato *VRML* (.wrl) o Inventor (.iv) ya que *GraspIt!* solo puede cargar los archivos de geometría con estos formatos.

Los archivos de geometría que contienen los formatos *VRML* (.wrl) e Inventor (.iv) pueden ser creados por diferentes plataformas software. Pero el formato mas común es el primero ya que la mayoría de software de diseño 3D traen consigo la opción de guardar los diseños en este formato. En ***Solid Edge*** al igual que en ***Matlab*** se pueden crear diseños 3D y luego guardarlos en el formato *VRML* (.wrl). En el Toolbox de Realidad Virtual de Matlab se incluye ***V-realm Builder*** como un editor 3D nativo. Con esta herramienta se pueden crear representaciones gráficas 3D y representar un estilo de árbol jerárquico de todos los elementos en el mundo virtual.

Cada una de las piezas (dedos y palma) del modelo de la mano del grupo fue construido en Solid Edge y se guardaron como archivos de imagen con el formato (.wrl). Al tratar de cargar estos archivos en *GraspIt!* se deben eliminar primero todas las configuraciones con respecto a la cámara y otros que no correspondan a la forma del objeto, ya que cuando se guardan los diseños en éste formato se guardan también configuraciones que no conciernen a la forma del objeto como por ejemplo la posición y orientación de la cámara o la distancia focal de la misma. Si lo anterior no se hace, cuando *GraspIt!* carga el cuerpo (objeto) este podría salir con algún color diferente o si ya se encuentra algún cuerpo en el mundo de simulación este no será visible al tratar de cargar otro cuerpo.

Cuando ya se tienen los archivos de geometría (falanges y palma), se debe crear un archivo *XML* (.*XML*) por cada uno de ellos, en los cuales se especifica el material, la masa, centro de gravedad y matriz de inercia. El siguiente ejemplo corresponde al archivo *XML* en el que se edita cada una de las propiedades de la falange inferior del dedo pulgar.

<?*XML* version="1.0" ?>
<root>
 <material>**metal**</material>
 <*mass*>**100**</*mass*>
 <*cog*>**0.00 0.0 0.0**</*cog*>
 <inertia_matrix>**3540.1 0 0 0 3350.1 -42.614 0 -42.614 1056.9**</inertia_matrix>
 <geometry*File*>**infpulgar.iv**</geometry*File*>
</root>

En la etiqueta **geometry*File*** del archivo *XML* es en donde de nombra el archivo de geometría al cual se le están destinando las anteriores propiedades. Después de que ya se han editado las propiedades de las falanges de cada dedo, incluyendo las propiedades de la palma, se crea un archivo *XML* en el cual se define un robot. Para ello se necesitan dos cosas: Los archivos *XML* de cada enlace incluyendo la palma, que son parte del robot, más un archivo de configuración general del robot, el cual tiene toda la información cinemática y las referencias apropiadas hacia los archivos de cuerpos que en este caso serán la palma y las falanges.

Cada una de las propiedades esta documentada con mas detalle en el manual de *GraspIt!* que se encuentra en los anexos de este trabajo.

A continuación se explicará como se creo el modelo de la mano del grupo. Para la explicación solo se tomará la palma de la mano y el dedo pulgar de la misma ya que para la configuración de los demás dedos se siguen los mismos pasos.

La configuración general del robot (mano robótica) se debe hacer en un archivo *XML*. La primera etiqueta que se edita es el tipo de robot que se esta creando. Para este ejemplo se utiliza el tipo **Hand,** ya que este robot no es una subclase de otros tipos genéricos tales como la "Barrett" o "Puma560", que tienen propiedades que pueden llegar a ser diferentes, como por ejemplo en el tamaño de las falanges o numero de grados de libertad.

<robot type="Hand">
... (resto del archivo del robot)
</robot>
El siguiente ítem es el nombre del archivo para el enlace palma.
<palm>palma.*XML*</palm>

Palma, es el nombre que se utilizo en el archivo *XML* en donde se configuraron las propiedades del archivo de geometría palma que se mencionó anteriormente.

La siguiente información es para los DOF`s. Para cada grado de libertad se debe editar el tipo de grado de libertad, la velocidad por defecto, la máxima fuerza que cada DOF puede aplicar a la junta, kp y kd que son los coeficientes de un controlador de fuerza PD que se encuentra en cada DOF, y por último la escala del manipulador. Las propiedades para cada DOF que se utilizaron en la mano del grupo fueron las siguientes:

- El tipo de grado de libertad que se utilizó para la mano del grupo fue rígido ya que no tendrá propiedades especiales tal como el rozamiento que se produce en la junta por parte de cada enlace.

- La velocidad por defecto se utiliza en una operación de *auto*-agarre. esta velocidad depende, entre otras cosas, de la maquina en la cual estemos trabajando, por lo tanto con una maquina que tenga mejor procesamiento y capacidad de memoria una operación de *auto*-agarre se verá más rápida y viceversa.

- La máxima fuerza que un enlace puede aplicar, al igual que kd y kp no son constantes, es decir, este valor cambia de acuerdo a la sintonización del PD, por lo tanto estas propiedades están sujetas a modificaciones.

- La escala del manipulador es el que modifica el radio del manipulador de cada DOF, es decir si en el mundo de simulación estamos en modo interacción (anexo: manual de

GraspIt!) y damos clic en una cadena cinemática, aparecerán unos manipuladores en forma de disco, los cuales al arrastrarlos circularmente actuarán como los motores que mueven los DOF`s

Las siguientes etiquetas corresponden a un grado de libertad de los nueve que se utilizaron para crear la mano del grupo. Los demás DOF`s son configurados de la misma manera.

<dof type="**r**">
<defaultVelocity>**1.0**</defaultVelocity>
<maxEffort>**5.0e+9**</maxEffort>
<Kp>**5e+7**</Kp>
<Kd>**9e+7**</Kd>
<draggerScale>**10**</draggerScale>
</dof>

La anterior configuración de DOF se debe repetir 3 veces Debido a que en el dedo pulgar se tienen tres grados de libertad.

La siguiente información se refiere a las cadenas cinemáticas. La primera etiqueta que se encuentra en una cadena cinemática es la posición y orientación de la junta (que es representada por el manipulador entes mencionado) en la palma de la mano. La primera junta que se configura es la que corresponde al enlace que esta junto a la palma (falange inferior del pulgar), por lo tanto, el orden de configuración de cada junta es desde la más cercana a la más lejana de la palma.

La posición y orientación de la junta están de acuerdo al origen de geometría de la palma, por lo que la ubicación de la junta no es una tarea fácil si se hace de forma manual, es decir, si se realiza dando valores manualmente a X, Y, Z. hasta llegar al punto en la palma en donde se colocará la junta y luego rotarla en alguno de los ejes hasta conseguir la orientación deseada.
Para dar solución a lo anterior se realizaron los siguientes pasos:

- En el mismo software en donde se creo la palma de la mano, se creo un disco que contenía el mismo origen geométrico de dicha palma, y se construyó de tal manera que una de sus caras fuera perpendicular a al eje Z de tal origen y luego se guardó en un formato .wrl o .iv (según el software de diseño 3D).

- El anterior archivo de geometría se referenció desde un archivo *XML* de la siguiente manera:

<?*XML* version="1.0" ?>
<root>
<geometry*File*>**disco.iv**</geometry*File*>
</root>

Y se guardó en la carpeta que contiene los enlaces que han sido guardados como archivos *XML*.

- Se cargaron como objetos en *GraspIt!* la palma de la mano y el disco antes mencionado.

- Se Posicionó y orientó el disco al lugar en la palma en el cual habitaría la junta. Se debe recordar que el manipulador de la junta tiene una forma de disco, razón por la cual se creo un archivo de geometría con esa forma.

- Se Guardó la escena como un archivo de mundo *GraspIt!* (.wld).
- Se Dirigió a la ruta en la cual se guardó el anterior archivo de mundo y se abrió en un editor de texto.

En el editor de texto aparecerá la posición y orientación del disco como una etiqueta marcada con ***fullTransform.*** El contenido de esa etiqueta será la posición y orientación del manipulador de la junta.

Después de que ya se tiene la posición y orientación de la primera junta, se deben configurar las propiedades de cada junta. La primera propiedad es el tipo de junta y esta puede ser "Revolute"o"Prismatic". Todas las juntas deben ser de tipo revolución según se expone en el modelo de la mano del grupo.

Los siguientes son los cuatro parámetros Denavit-Hartenberg que definen la transformada entre las juntas de un robot. Cada junta tiene sus ejes alineados con el eje Z de todas las juntas presentes en la cadena cinemática. Los cuatro valores son theta, d, a, y alpha. Las siguientes etiquetas *XML* son tomadas de la primera junta de la primera cadena cinemática (dedo pulgar) del modelo de la mano creado en *GraspIt!*

<transform>
<fullTransform>(-0.000948313 +0.910618 -0.0430904 +0.410995)[+10.16257 +0.663885 -100.443]</fullTransform>
</transform>
<joint type="Revolute">
<theta>d0</theta>
<d>5</d>
<a>44
<alpha>270</alpha>
<minValue>0</minValue>
<maxValue>70</maxValue>
</joint>
theta: Define que DOF pertenece a esa junta. En el modelo de la mano se define que la primera junta pertenece al primer DOF y así sucesivamente. Este parámetro, es además, la rotación sobre el eje x de todo el marco de juntas y es variable solamente para juntas de tipo "Revolute".

Debido a que una cadena cinemática consta de mas de una junta, las propiedades de la primera junta (la mas cercana a la palma) afectan el comportamiento de las demás juntas, por ejemplo, al decir que theta es equivalente a “d0+180” estamos diciendo que la junta actual pertenece al primer DOF (que se configuró en el archivo de la mano) y que este DOF girará 180 grados. Un ejemplo gráfico puede ser visto en la figura C.4.1. En esta figura se muestra la palma de la mano del grupo con la falange inferior del dedo pulgar. Cuando **theta** es configurada como “d0” (ver Figura C.4.1 a)) la falange esta penetrando la palma de la mano, además, no seria coherente que la falange del dedo pulgar tuviera esa posición u orientación, por lo tanto hay que hacer girar la falange 180 grados (ver figura C.4.1 b))

a) **theta** configurada como “d0”. (b) **theta** configurada como “d0 +180”

Figura C.4.1. Diferentes configuraciones de d0

No siempre es necesario cambiar la configuración de theta a 180 grados (o algún otro valor en grados), esto dependerá de cómo aparezca el enlace en el visor de *GraspIt!*

d – Es la traslación del marco de juntas a lo largo del eje Z. Debido a la forma de la falange inferior del dedo pulgar, esta etiqueta tiene un valor de 5 mm. Observar la figura C.4.2.

a- Es la traslación del marco de juntas a lo largo del eje X. Observar la figura C.4.2.

aplha- Es la rotación del marco de juntas alrededor del eje X.

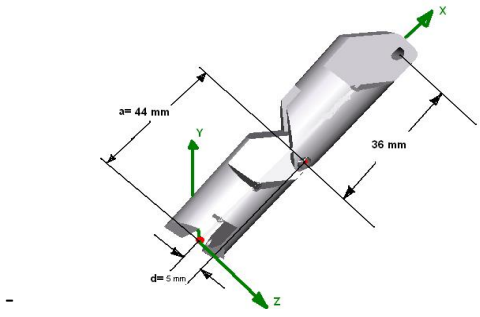


Figura C.4.2. Falanges inferior y media del modelo de la mano del grupo con las respectivas configuraciones de las juntas.

Las etiquetas ***minValue y maxValue***, constituyen, respectivamente, el límite mas bajo y el límite mas alto de la junta actual. Las otras dos juntas son configuradas de la misma manera, pero teniendo en cuenta el modelo de la mano del grupo. Ahora lo que debemos decirle a

GraspIt! es cual archivo de geometría, con sus respectivas propiedades, constituirá una falange determinada. Para ello nos dirigimos a las últimas etiquetas de la cadena cinemática. En ellas se establece el orden de los enlaces, del más cercano al más lejano con respecto a la palma de la mano. A la falange inferior del dedo pulgar se le llamo infpulgar.*XML*, por lo tanto es el primer nombre que debe aparecer en estas etiquetas y así sucesivamente con la falange media y superior del dedo pulgar. Cuando estamos configurando una cadena cinemática lo primero que debemos organizar son las juntas sin importar si los enlaces (falanges) quedan posicionados u orientados en lugares que se consideran inapropiados. Lo anterior sucede es por que cuando *GraspIt!* carga un enlace este no se ajusta al mismo origen de geometría de la palma de la mano ya que trae consigo su propio origen de geometría. Para solucionar este problema lo primero que se hace es abrir el archivo de geometría que se utiliza como la falange inferior y adherirle propiedades de traslación y rotación (de acuerdo a su propio origen de geometría) con el fin de que cuando sea cargado por *GraspIt!* éste se ubique en la posición y orientación correcta, es decir, que cuando se mueva el DOF se produzca un movimiento coherente de la falange. Para trasladar o rotar el archivo de geometría dentro de *GraspIt!* se utilizaron algunas de las propiedades de los nodos que trae el kit de herramientas de Open Inventor. Los nodos que se utilizan son ***Translation*** *y* ***RotationXYZ.*** A continuación se muestra la implementación de estos nodos para la falange inferior del dedo pulgar:

```
Translation
        {
translation -44 1  0
        }
 RotationXYZ    {
          axis     X
          angle   0
    }
RotationXYZ
    {
          axis     Y
          angle   1.5708
    }
RotationXYZ
    {
          axis              Z
          angle   4.7124
    }
```

Para llevar los enlaces a las posiciones correctas, la configuración de los ángulos (en radianes) fue hecha manualmente al igual que las coordenadas utilizadas para trasladar el enlace.

C.5 CREAR UN AMBIENTE DE SIMULACIÓN CON UNA MANO Y UN OBJETO

Un ambiente de simulación, es un ambiente virtual en el cual colocamos objetos y robots que son semejantes al mundo real, no tan solo en su forma, tamaño o material, sino que también en la posición y orientación con respecto al ambiente de simulación. Por lo tanto algunos objetos, al igual que en el mundo real, pueden contener movimiento o simplemente servir como base para contener otros objetos. Para dar un ejemplo de lo anterior se creará un mundo de simulación que constará de una mano robótica (***HumanHand****)* y un Matraz.

Para comenzar se recomienda colocar primero el robot. Para este ejemplo se utilizará la mano robótica ***HumanHand*** que viene con la distribución de *GraspIt!*

1. Ir al menú en ***File →Import Robot …***

2. Dar clic en la carpeta que viene con el nombre del robot en este caso ***HumanHand*****.** Aparecerá una ventana en la cual aparecen, entre otros, 3 archivos *.XML*.

3. Dar clic en el archivo *XML* con el nombre *HumanHand*16DOF. La escena deberá ser parecida a la que aparece en la siguiente figura:

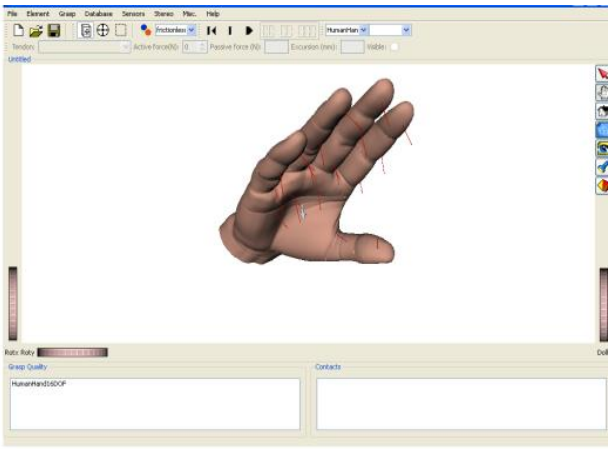


Figura C.5.1. Robot *HumanHand* que viene en la distribución de *GraspIt!*

4. Ir de nuevo al menú en ***File →Import*Object…** y dar clic en ***flask*** la escena se verá de la siguiente manera:

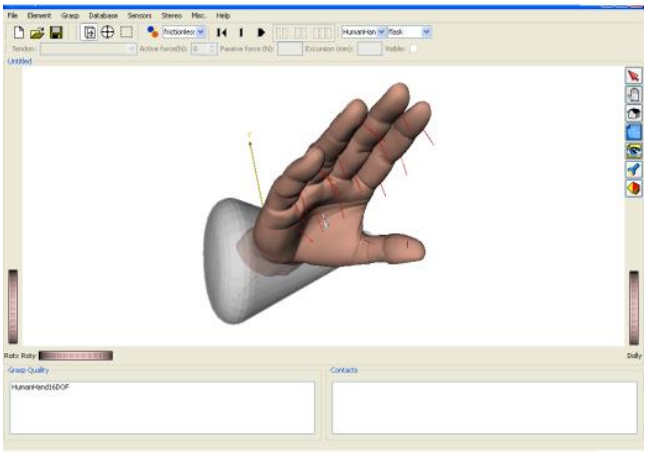


Figura C.5.2. Robot ***HumanHand*** traslapándose con el objeto ***flask*** que viene con la distribución de *GraspIt!*

5. Como puede observarse el Matraz esta traslapándose con la mano por lo que hay que correr ya sea la mano o el Matraz a una posición de agarre. La posición de agarre depende del usuario, por lo tanto la posición y orientación de la mano y el matraz podrían ser configuradas para examinar un tipo particular de agarre.

6. Para este ejemplo solo se configurará la posición del matraz. Para ello se debe ir a la barra de herramientas que se encuentra en la parte derecha del visor de *GraspIt!* y dar clic en . En ese momento se activa el modo de interacción el cual es el único modo en el cual se puede interactuar con *GraspIt!*
7. Se escoge el tipo de interacción de traslación que esta etiquetado con ya que es el que nos permitirá mover el Matraz.

8. Dar clic en el Matraz, en ese momento aparecerá una caja transparente encerrando el objeto.
9. Deshabilitar el sistema de detección de colisión. Al dar clic en deberá verse el botón así . Este paso se hace debido a que cuando se detecta una colisión los elementos que se encuentran en dicha colisión no pueden moverse.

10. Presionar el botón izquierdo del ratón en una de las caras de la caja y arrastrar el Matraz a una posición en la cual la mano pueda agarrar. La siguiente figura muestra una posible posición del Matraz.

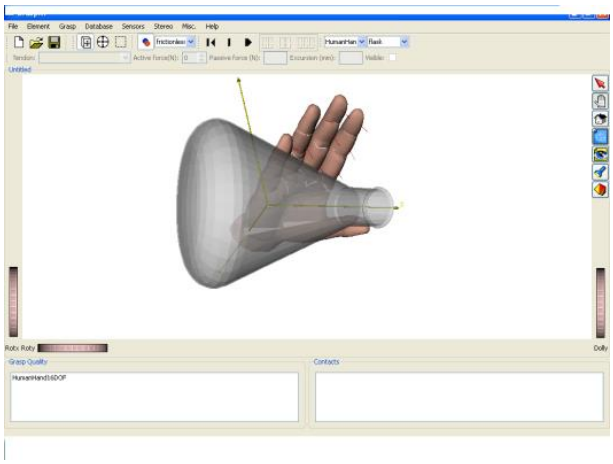


Figura C.5.3. Posible posición del Matraz, con respecto a la mano, en el mundo de simulación

11. Una de las funcionalidades mas sencillas que se implementa en *GraspIt!* es el *auto* agarre el cual consiste en mover cada una de las cadenas cinemáticas (dedos) hasta que uno de sus enlaces detecte colisión, momento en el cual se detiene el movimiento de la cadena cinematica. Para ello se debe habilitar el sistema de detección de colisión (dar clic en) e ir al menú en ***Grasp → Auto Grasp*.** se observará como la mano agarra el Matraz (el agarre puede ser inestable) y aparecen los contactos entre la mano y el Matraz.

C.6 GENERACIÓN DE MOVIMIENTO EN EL MODO ESTÁTICO Y DINÁMICO DESDE MATLAB

Como se ha mencionado antes en el manual de *GraspIt!* existen dos maneras de simular en *GraspIt!* : el modo estático y el modo dinámico. En este apartado se mostrara un ejemplo de simulación utilizando el modo estático de *GraspIt!* controlado desde Matlab.
Para la simulación en modo estático desde Matlab existe un comando de interfaz Matlab-*GraspIt!* moveDOFs(r,D,deltaD) que mueve los DOF(Degree Of Fredom) de un robot a unas posiciones deseadas.Para llamarla se puede utilizar la siguiente línea en Matlab:

newD = moveDOFs(r,D,deltaD) Esta función necesita 3 argumentos de entrada, donde:

r: es el índice del robot al cual se le moverán los DOF.
D: es el vector de posiciones deseadas. El numero de elementos de **D** debe corresponder a el numero de DOF del robot **r**.

deltaD: es un vector que contiene el tamaño de paso que debe tomar cada DOF para moverse a la posición deseada.

El numero de elementos de deltaD debe corresponder a el numero de DOF del robot **r**.La función devuelve una salida, la cual es un vector de posiciones de los DOF, al final de la operación de movimiento.Esta función es usada para controlar cinematicamente las posiciones de los DOF, moviendo los DOF por pasos de tamaño deltaD hasta que los valores deseados sean alcanzados o algún contacto ocurra. El comando no se usa en el modulo dinámico, aunque debido a un error que creemos que es debido a la actualización del servidor de *GraspIt!* podría ser necesario utilizarla en el modulo dinámico controlado desde Matlab.
Las posiciones de los DOF deseados pueden ser alcanzados en un solo paso utilizando lo siguiente:

newD = moveDOFs(r,D,D - getDOFVals(r))

Ahora se mostraran dos ejemplo de movimiento de las articulaciones de un robot, que en este caso es la mano del grupo (*manogrupo.XML*).Para comenzar se debe crear un nuevo mundo en *GraspIt!* y colocar la mano del grupo, luego en Matlab se debe abrir el ejemplo llamado grupoestatica.m .A continuación se muestra la implementación de grupoestatica.m

```
%grupoestatica,   mano del grupo de 9 DOF
%movimiento de la mano en modo estatico

%setPoint =[  pulgar       indice        medio  ]
setPoint =[1.1 0.0 0.0   1.57 0 0      1.57 0  0]';%posicion final de la mano
Di=          [0    0    0      0   0   0      0   0  0]';%posicion inicial de  la mano
r=1;% = robot No 1
deltaD=.01*ones(9,1);%paso de movimiento de 0.01 dado en radianes para cada articulacion
newD= moveDOFs(r,setPoint,deltaD);% movimiento de la articulaciones

pause(5); %tiempo de visualización, antes de que la mano vuelva a su configuracion inicial

moveDOFs(r,Di,deltaD)% orden de movimiento a su configuración inicial
moveDOFs(r,Di,deltaD)
```

El anterior ejemplo funciona de la siguiente manera: en primer lugar se tiene un arreglo de tamaño 9x1 cuyo nombre es setPoint: donde el 1 significa que hay un solo robot y el 9 indica que hay nueve valores para los 9 grados de libertad correspondientes de acuerdo con la configuración del robot. Los primeros tres valores del setPoint se refieren a las tres articulaciones del pulgar, normalmente el último valor de los tres es el que mueve la articulación encargada de mover la punta del dedo; los valores 4,5y 6 se refieren al dedo índice y por ultimo, como no hay mas dedos, los valores 7,8 y 9 se encargan de dar movimiento al dedo medio. Así que si se quiere una configuración de la mano es en el setPoint donde se debe configurar los valores de las posiciones finales de las articulaciones para obtener una configuración deseada.Cada valor del setPoint de debe colocar en radianes, su valor también debe estar entre los limites de las articulaciones configuradas en el archivo *.XML* del robot.

Di: representa la configuración inicial de la mano, en este caso todos sus valores son cero, lo que representa la mano totalmente abierta.Di esta aquí simplemente para volver a una posición inicial desde Matlab para no tener que ir a *GraspIt!* y dar click en *auto*Open.

moveDOFs(r,setPoint,deltaD): mueve los 9 DOF del robot r a un paso de tamaño deltaD=0.01 para cada articulación hasta llegar al setPoint, hasta llegar al limite de las articulaciones o hasta que ocurra un contacto.
newD: es igual a los valores finales de las articulaciones, en este caso después de la ejecución deberían ser los mismos valores del setPoint.En otros casos en que la mano intente agarrar un objeto serán las posiciones finales de las articulaciones para el agarre del objeto.

pause(5): solamente indica el tiempo en que se puede ver la mano con la configuración dada por el setPoint, después de este tiempo=5 segundos la mano vuelve a Di o a su configuración inicial mediante el comando moveDOFs(r,Di,deltaD). moveDOFs Se repite debido a que algunas veces se necesita primero que vuelva un dedo a su configuración inicial para que luego le de paso a otro dedo, por ejemplo el pulgar algunas veces queda encima de el dedo medio, en este caso solo volverá a la configuración inicial el pulgar y el índice, necesitándose así una nueva orden de regreso a la configuración inicial para el dedo medio.

El segundo ejemplo que se muestra a continuación es muy parecido al anterior solo que esta ves se utiliza una función creada por nosotros para generar el movimiento de acuerdo con una trayectoria entregada. Para comenzar, en este caso deje el mismo mundo de *GraspIt!* del ejemplo anterior y cargue en Matlab grupo_estatica_trayectoria.m.El ejemplo muestra 4 trayectorias cada una configurada con valores de articulaciones distintos. Dada las trayectorias, la función movtrayectoria se encarga de mover en el mundo de *GraspIt!* las articulaciones a los valores dados por cada trayectoria

```
% generacion del mivimiento por medio de trayectorias
close all
clear all
clc

DOF=9; % grados de libertad

%**********trayectorias de ejemplo*********
t{1}=[.1 0.0 0.0   0.1 0 0       0.1 0  0]';% trayectoria 1
t{2}=[.3 0.0 0.0   0.5 0 0       0.1 0  0]';% trayectoria 2
t{3}=[.5 0.0 0.0   0.7 0 0       0.1 0  0]';
t{4}=[.8 0.0 0.0   1   0 0       0.8 0  0]';

movtrayectoria(t,DOF);% funcion para dar movimiento a las articulaciones de acuerdo con unas trayectorias.
```

La implementación de la función movtrayectoria se basa en la implementación del primer ejemplo solamente que ahora los DOF se mueven a diferentes setPoints .A continuación se muestra la implementación de esta.

```
% funcion para el movimiento de una trayectoria
function movtrayectoria(t,DOF)

for i=1:length(t)

setPoint =t{i};
r=1;% robot No 1
deltaD=.01*ones(DOF,1);%paso de movimiento de 0.01 dado en radianes para cada articulación
newD=moveDOFs(r,setPoint,deltaD);% movimiento de la articulaciones
pause(1);
end
pause(2); %tiempo de visualización, antes de que la mano vuelva a su configuracion inicial

Di=     [0   0   0   0  0  0      0  0  0]';% configuración inicial de la mano
moveDOFs(r,Di,deltaD)%orden de movimiento a su configuración inicial
moveDOFs(r,Di,deltaD)
end
```

Notar que la diferencia con el primer ejemplo solo esta en que ahora el setPoint toma diferentes valores para cada ciclo del for.

1. Uso Del Modo Dinámico

GraspIt! tiene dos maneras de ir a modo dinámico: el modo dinámico normal de *Grasp*It y desde Matlab.

1.1 El modo normal de *GraspIt!*

Para iniciar el motor de dinámicas, tal y como se explica en el manual, hay que presionar el botón *start simulation* para comenzar un simulación dinamica. Al iniciar una simulación dinámica comienzan a actuar diferentes fuerzas en el mundo como la gravedad y las fuerzas que ejercen los controladores de cada DOF.Los controladores de cada DOF son del tipo PD, por lo tanto hay que sintonizarlos modificando Kp y Kd.Para encontrar los valores adecuados de Kp y de Kd se debe de tratar de generar trayectorias con una aceleración suave y per*file*s de velocidad desde una posición actual a una deseada en los DOF.La velocidad de movimiento de los dedos depende de el valor de Kp,Kd y la velocidad por defecto; los valores en promedio utilizados para sintonización son del orden de 1.0e+10 para Kp, 1.0e+7 para Kv y 1 para la velocidad por defecto. Todos los parámetros anteriores se pueden modificar en el archivo de

extensión *XML* encargado de la configuración del robot, a continuación se muestra un ejemplo de la etiqueta <dof> desde el archivo Mano_Grupo.*XML*

```
<dof type="r">
<defaultVelocity>1.0</defaultVelocity>
<maxEffort>5.0e+9</maxEffort>
<Kp>1.0e+10</Kp>
<Kd>1.0e+7</Kd>
<draggerScale>10</draggerScale>
</dof>
```

La etiqueta anterior es solo para la configuración de un DOF, se debe sintonizar para cada DOF que tenga la mano.
Para probar el funcionamiento de los controladores se comienza la simulación y se activa el *auto* agarre. Una ves hecho esto, los dedos comienzan a moverse hasta donde la fuerza motora ejercida por los DOF's los lleven o hasta que haya algún tipo de colisión, contacto entre los mismos dedos, contactos con algún objeto, algún obstáculo detenga el movimiento o por lo menos los frenen permitiéndoseles moverse más lento debido a que ahora los dedos tiene que mover un objeto con masa y peso, esto en el caso de que el objeto sea un objeto dinámico. Si el controlador esta bien sintonizado el agarre será exitoso y se podrá detener la simulación en el momento que usted quiera para que *GraspIt!* muestre las fuerzas de contacto, pares ejercidos sobre el objeto,etc. Si el controlador esta mal sintonizado la simulación se detendrá debido a errores de colisión e interpenetración por causa de fuerzas muy grandes, también puede ocurrir que haya colisión entre los mismos dedos, colisión entre los dedos y el objeto,etc.Debido a lo anterior, *GraspIt!* detiene la simulación y por lo tanto no se obtiene un buen agarre.

En modo de simulación dinámico, *GraspIt!* tiene la cualidad de poder cambiar el tipo de material del objeto y del robot. Todo esto es de gran ayuda para determinar el mejor material para que la mano pueda agarrar la mayoría de objetos con mejor calidad de agarre.

1.2 Simulación Dinámica Utilizando Matlab

En el manual de *GraspIt!* se habla de la interfaz con Matlab donde existen unos comandos para obtener datos de *GraspIt!*, todos estos comandos nos sirven para realizar una simulación dinámica desde Matlab.Si se requiere mas información de estos comandos digite help matlab en la ventana de comandos de Matlab y abra cada .m que aparece allí para mayor información.A continuación se muestra un ejemplo de simulación dinámica para la mano del grupo llamado grupodinamica.m:
%trabajando con la mano del grupo de 9 DOF utilizando los comandos de %interfaz de Matlab-*GraspIt!* y un controlador PD para la simulación dinámica %en *GraspIt!*.

```
%%
close all
```

```
clear all
clc

Kp =[1e+7 1e+7 1e+6   2e+7 2e+6 2e+6   2e+7 2e+6 2e+6]';
Kv =[1e+4 1e+4 1e+3   1e+4 8e+3 1e+3   1e+4 8e+3 1e+3]';
setPoint =[0.8 .5 .5   1.3 1 .5   1.3 1 .5]'; %valores en radianes
lastError=[0 0 0 0 0 0 0 0 0 ]';

r=1; %robot 1
deltaD=.01*ones(9,1); %paso de movimiento de 0.1
D=[0.8 0 0   0 0 0   0 0 0]'; % posición inicial de la mano
moveDOFs(r,D,deltaD); %movimiento de los DOF's hacia D del robot 1 con %pasos de deltaD

%%
while 1

timeStep = moveDynamicBodies;
%**********************controlador*****************************
dofvals = getDOFVals;
error = setPoint - dofvals;
velerror = (error-lastError)/timeStep;
forces = Kp .* error + Kv .* velerror
lastError = error;
%*****************************************************************
setDOFForces(forces);
movdof(forces); % función hecha para actualizar DOF’s en Matlab
computeNewVelocities(timeStep);
render;

end
```

El ejemplo funciona de la siguiente manera: primero se configura una posición inicial de la mano (ver figura C.6.1), para realizar esto usted debe saber que índice del arreglo corresponde a cual dedo o específicamente a cual articulación; en la ventana de comandos de Matlab digite getBodyName para obtener una lista de cuerpos donde usted puede encontrar como están organizadas las cadenas y los enlaces. Normalmente el índice de los arreglos para el setPoint, getDOFVals, D, deltaD, etc.Tienen su correspondencia con los dedos: 1,2 y 3 corresponde a el dedo pulgar; 4,5 y 6 corresponde a el dedo índice; 7,8 y 9 al dedo medio, etc. Una ves dado el setPoint de posición de los DOF’s las articulaciones trataran de llegar a ellos mediante fuerzas motoras aplicadas a ellas, controladas por PD. El setPoint puede ser la posición final de las articulaciones para que los dedos puedan agarrar un objeto. El bucle y los comandos incluidos en el se encargan de realizar la simulación paso a paso, este paso normalmente es 0.0025 segundos (*timeStep*).El controlador se encarga de calcular las fuerzas necesarias para llevar los DOF’s a el setPoint y los demás comandos se encargan de representar los efectos de las

fuerzas en las articulaciones en el mundo de *GraspIt!*; efectos como velocidad de movimiento de los dedos y movimiento de los cuerpos debido a las fuerzas aplicadas.Al correr grupodinamica.m se observa como los dedos comienzan a moverse y a tomar la forma del setPoint o simplemente la forma de la copa(ver figura C.6.2). Dado que se están controlando 9DOF's y al diseño de la mano del grupo es complicado sintonizar todos los 9DOF's correctamente para obtener una simulación completa. Haciendo algunos cambios de material a la mano se logra una simulación mejor y más estable. Cabe mencionar que los controladores todavía les falta sintonización.

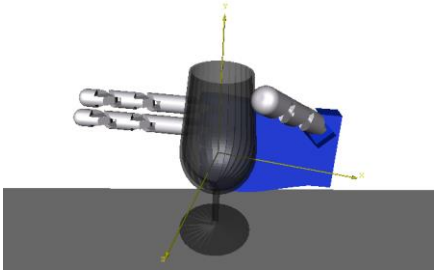


Figura C.6.1. Posición inicial de la mano

En este ejemplo se ha sintonizado los controladores ***PD*** para un mundo en *GraspIt!* especifico, no se puede decir que sea general ya que dependiendo del mundo del *GraspIt!* habrán diferentes fuerzas actuando en distintas direcciones, sin embargo las modificaciones para la nueva sintonización de un nuevo mundo deberían ser sencillas o pequeñas.

La siguiente función mostrada abajo que esta incluida en el ejemplo anterior no es un comando de interfaz de Matlab con *GraspIt!*, esta se ha hecho solamente para tratar de resolver el problema de actualización de los DOF's. Esta función debe ser actualizada a medida que usted va cambiando los valores de Kp principalmente.

```
%funcion realizada para resolver el problema de actualización de los DOF's
%ya que getDOFVals  no entrega valores actualizados si moveDOFs no ha
%movido los DOF's
%nota: esta funcion no es de los comandos de GraspIt!, es realizada con
%algunos de ellos.

function movdof(f)
D=getDOFVals;
r=1;
deltaD=0.1*ones(length(f),1);

for i=1:length(f)
D(i)=D(i)+1.0e-7*f(i)/2.5; % 1e-7 es un valor que va de acuerdo con Kp %maximo
end
moveDOFs(r,D,deltaD);
end
```

La función anterior se puede quitar para que no se controle la fuerza si no que simplemente actúe una fuerza constante sobre cada una de las articulaciones, de esta manera se obtiene un agarre por cierre de fuerza pero sin optimizar.

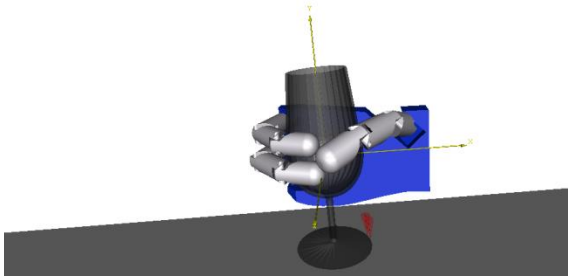


Figura C.6.2. Agarre de una copa de vidrio con la mano del grupo

Realizar una simulación dinámica en *GraspIt!* Depende en gran medida de una buena sintonización, sin embargo no todo depende de esto ya que algunas veces el motor de dinámicas de *GraspIt!* presenta algunas inestabilidades ocasionales. Cuando usted hace una simulación, *GraspIt!* mantiene los datos de simulación para que cuando vuelva a comenzar una segunda simulación se represente la escena muchísimo más rápido, moviéndose los cuerpos más rápido debido a que las fuerzas ya están configuradas.

Fuerzas de contacto y posiciones

En *GraspIt!* Solamente se muestran los valores de contactos de fuerza y par, mientras que en Matlab hay dos comandos de interfaz Matlab-*GraspIt!* para obtener las fuerzas de contacto y posiciones de estas.

Comando Getcontacts

El comando getContacts([vector de cuerpos]) retorna todas las fuerzas de contacto y posiciones para cada cuerpo o cuerpos. Donde el vector de cuerpos corresponde a un vector de índices correspondientes a cada cuerpo o solamente puede ser un número correspondiente a un solo índice de un cuerpo. Una forma adecuada de llamar a la función es como sigue:

[F, P] = getContacts (b)
Donde F=fuerza, P=posición y b=vector de índices de cuerpos.

Si se utiliza getContacts sin ningún argumento, esta retornara la información de los contactos para cada cuerpo. Cada fila de la celda de F contiene un vector 6D(3 columnas de fuerza y 3 columnas de par) de fuerza de contacto para un contacto que esta actuando sobre un solo cuerpo relativo al marco de coordenadas del cuerpo. Cada fila de una celda de P contiene un

vector de posiciones 3D de un contacto sobre un solo cuerpo relativo a el marco de coordenadas del cuerpo. P corresponde a F. Las celdas del arreglo de celdas de salida corresponden a los índices en b o a todos los cuerpos en el mundo, si b no es dado.

Comando Getaveragecontacts

Algunas veces existen varios contactos en un solo eslabón (por ejemplo la punta de un dedo) de un dedo, entonces lo que hace getAverageContacts es sacar el promedio de fuerzas de contacto y posiciones para obtener a la salida solo un contacto promedio que incluye solo un fuerza de contacto y una posición.
La utilización del comando funciona de igual manera que getcontacts.

Utilidad

Ahora veremos la utilidad de estas fuerzas de contacto y posiciones, pero antes veremos como están medidas y que significan.

Las posiciones (*P*) están dadas en 3D y corresponden con el marco de coordenadas del cuerpo, en estas posiciones están actuando las fuerzas de contacto en una dirección y magnitud dada por el vector de fuerza de contactos (F).

Como verificación de lo dicho anteriormente se ha hecho un ejemplo donde se construyo un mundo en *GraspIt!* con una mano y un cubo como objeto para ser agarrado. Luego se reconstruyo, en Matlab, el cubo y todos los contactos que se generan al agarrarlo.

Para revisión del ejemplo abra el mundo de *GraspIt!* llamado pruebacontactos.*XML* y en matlab abra pruebacontactos.m. Genere algunos contactos en *GraspIt!* ya sea con la simulación dinámica o en simulación estática moviendo los dedos de la mano para que toquen el cubo. Luego corra pruebacontactos.m y observara una imagen parecida a la C.6.3.

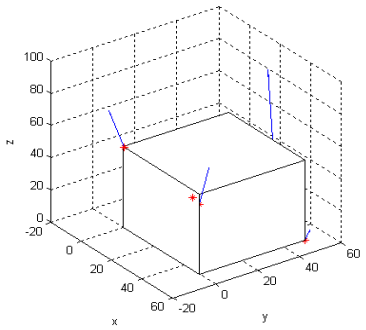


Figura C.6.3. Reconstrucción de las fuerzas de contacto y posiciones para un cubo agarrado en *GraspIt!*

Los contactos se pueden utilizar para conocer el estado de la mano en un agarre, para determinar que cantidad y dirección de fuerza se necesita aplicar para estabilizar un objeto,

para saber que tan buena calidad de agarre tiene una mano, manejar pares en caso de que aparezcan ,etc.

C.7 PRUEBAS EN *GraspIt!* PARA PROBAR LA EFICACIA DEL PARALELISMO ENTRE SUPERFICIES.

A continuación, se expondrá por que el paralelismo, entre la superficie local del objeto y la superficie de la punta del dedo, aumenta la probabilidad de un agarre estable. Lo primero que se hace es denotar matemáticamente, como el incremento de puntos de contacto de un agarre se ve reflejado en un agarre con mayor estabilidad; lo segundo que se hace es obtener por medio del simulador *GraspIt!* las medidas de calidad del agarre de tres objetos. Para cada objeto primero se hace una operación de *auto*-agarre y luego se pone la superficie de la punta de los dedos lo más paralela posible a la superficie local del objeto.

Cuando, físicamente, se tiene un contacto cara-cara los puntos de contacto que pueden aparecer son infinitos. Considerar todos estos puntos de contacto seria, matemáticamente, imposible pero en [23], al igual que en otros trabajos, se han direccionado esta clase de contactos como un conjunto de contactos de punto-punto ubicados en los vértices del plano que esta en contacto. Por lo tanto, un contacto cara-cara aportará más componentes de fuerza y par que un contacto de punto-punto o de borde-borde. El incremento de dichas componentes de fuerza y par se ven directamente reflejados en la suma total de todos los *Wrenches* que se están aplicando al objeto, los cuales, como se describe en el capitulo 6, resultan en la siguiente ecuación:

$$\omega_0 = G_1 f_{c1} + \cdots + G_n f_{cn} = [G_1 \dots G_n].\begin{bmatrix} f_{c1} \\ \vdots \\ f_{cn} \end{bmatrix} = G.\begin{bmatrix} f_{c1} \\ \vdots \\ f_{cn} \end{bmatrix} \quad \text{(C.7.1)}$$

En donde, los mapas de contactoG_i de cada punto de contacto son coleccionados en el mapa de agarre, definido como la matriz $\boldsymbol{G}$:

$$\boldsymbol{w_0 = G.f_c f_c \in FC} \quad \text{(C.7.2)}$$

Un agarre se dice que tiene force-closure, si para cualquier ω_{ext} existe un ω_0 permisible tal que $\omega_0 = -\omega_{ext}$. Por lo tanto, si aumentamos la suma total de *Wrenches* que se están aplicando al objeto, la configuración de agarre actual podrá resistir *Wrenches* de disturbio de mayor magnitud y de diferentes direcciones. Lo anterior se ve reflejado en un incremento en las medidas de calidad del agarre, implementadas en *GraspIt!*, ya que el agarre cuyos contactos sean de cara o planos puede aplicar una mayor variedad de fuerzas y pares en el plano horizontal y vertical del objeto como puede observarse en la determinación cuantitativa de la calidad del agarre de distintos objetos descrita en el capitulo 6.

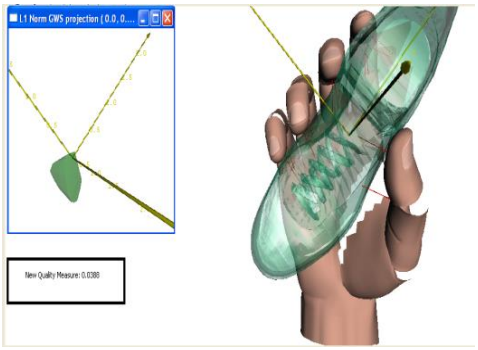
En [24] se implementó un sistema de control tal que: 1) los dedos no colisionaran con otro objeto y 2) ellos cerraran simultáneamente, formando una configuración de agarre simétrico alrededor del objeto. Esta habilidad es principalmente utilizada en agarre de objetos en donde su geometría es desconocida.

La arquitectura de control jerárquica usada se compone de dos elementos de control reactivo que se ejecutan asíncronamente y que controla aspectos separados del proceso de agarre.
En el nivel inferior, el controlador de distancia de los dedos asegura que éstos se mantengan a una distancia aproximadamente igual con relación a la superficie del objeto. En la parte superior un controlador de condicionamiento cinemático controla que el brazo lleve la mano robótica al centro del objeto y que la punta de los dedos quede paralela a la superficie local del objeto. Lo último se realiza con el fin de que la configuración final de agarre corresponda a la orientación del objeto aumentando de esta manera la probabilidad de un agarre estable. Para probar que el paralelismo entre la superficie local del objeto y la superficie de la punta de los dedos aumenta la probabilidad de un agarre estable se simularon en *GraspIt!* diferentes agarres sobre objetos de diferente forma. El primer agarre que se realizó sobre dichos objetos fue el "*auto*-agarre", que esta implementado en *GraspIt!*, el cual dependiendo de la configuración inicial de los dedos, realiza solo movimientos de flexión hasta que se detecte un contacto con el objeto. El segundo agarre que se simuló sobre cada objeto fue tratar de colocar, por medio de movimientos de abducción-aducción, la punta de los dedos lo mas paralela posible a la superficie local del objeto.

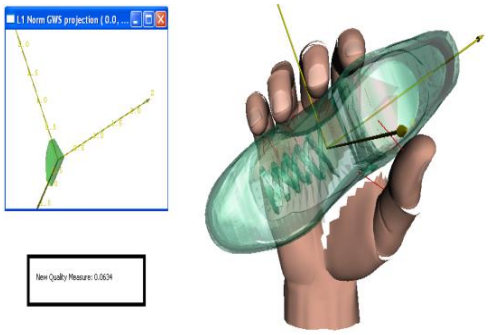
Los objetos que debían es*cog*erse para simular las diferentes configuraciones de agarre deberían corresponder a los encontrados comúnmente en una casa y que fueran de una forma no tan regular, ya que la idea general es implementar el enfoque de planificación de agarre en una mano robótica, que pueda reemplazar la mano humana. Aunque son muchas las formas de objetos que pueden encontrase en una casa, solo se tomaron como referencia dos de ellos (zapato y un vaso de café) ya que estos se encontraban en la distribución de *GraspIt!*, y no era necesario tener un software de diseño 3D para crearlos.

La mano que se utilizó para simular los diferentes agarres es la *HumanHand*, la cual se encuentra en los modelos de robots que trae consigo la distribución de *GraspIt!* Este robot tiene 20 grados de libertad (DOF´s). Cada dedo utiliza tres DOF´s para realizar movimientos de flexión y extensión y un DOF para movimientos de abducción y aducción. El dedo pulgar utiliza los 4 DOF`s restantes para realizar los movimientos de oposición, flexión-extensión y aducción-abducción.

En las siguientes graficas se muestra cada uno de los agarres realizados sobre cada objeto. En la parte izquierda-arriba de cada grafica se crea una proyección del Espacio *Wrench* Del Agarre y en la parte izquierda-abajo se muestra la medida de calidad obtenida de la configuración de agarre actual.

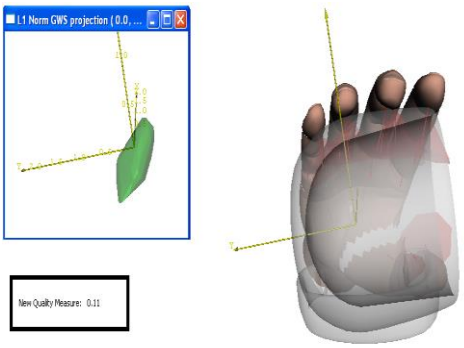


a) Operación de *auto*-agarre de un de un zapato

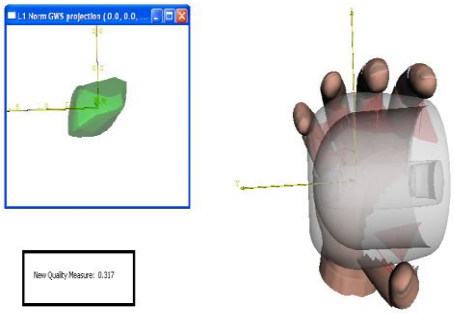


b) Posición de la punta de los dedos lo mas paralela posible a la superficie

Figura C.7.1. Diferentes configuraciones de agarre de un zapato.

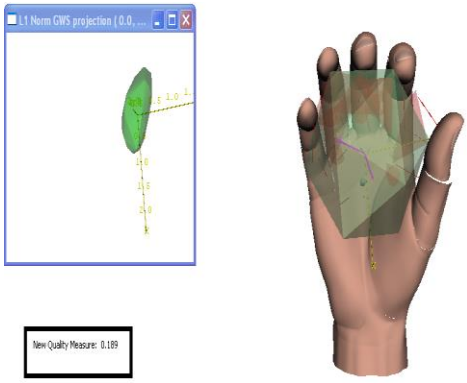


a) Operación de *auto*-agarre de un de un vaso de café

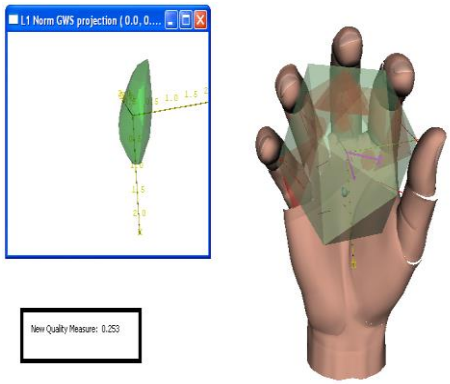


b) Posición de la punta de los dedos lo mas paralela posible a la superficie

Figura C.7.2. Diferentes configuraciones de agarre de un vaso de café.



a) Operación de *auto*-agarre de un de un icosaedro



b) Posición de la punta de los dedos lo mas paralela posible a la superficie

Figura C.7.3. Diferentes configuraciones de agarre de un icosaedro.

En las figuras C.7.1 y C.7.2 se puede observar el robot *HumanHand* agarrando un zapato y un vaso de café respectivamente. En las figuras C.7.1 a) y C.7.2 b), se realiza una operación de *auto*-agarre, en ellas se observa como la mano robótica agarra el objeto de tal manera que la superficie de la punta de los dedos no es totalmente paralela a la superficie local del objeto. En las figuras C.7.1 b) y C.7.2 b) se observa un incremento en la medida de calidad de agarre debido a que la punta de los dedos, por medio de movimientos de aducción y abducción, quedo lo mas paralela posible a la superficie local del objeto, haciendo con ello una configuración de agarre mas simétrica lo cual aumenta las capacidades de fuerza de contacto en el agarre.

En la figura C.7.3 a) se muestra el agarre de un icosaedro[5], en ella se observa como en una operación de *auto*-agarre, la punta del dedo anular contacta en uno de los vértices de los triángulos que conforman el icosaedro, y los dedos índice y meñique no están totalmente paralelos a la superficie. En la figura C.7.3 b) se muestra el mismo objeto pero esta ves con una configuración de agarre mas simétrica, en donde la superficie de la punta de los dedos queda lo mas paralela posible a la superficie local del objeto dando como resultado un notable incremento en la medida de calidad en el agarre.

[5]El icosaedro, es un solido limitado por 20 caras iguales, las cuales son triángulos equiláteros. Este objeto fue utilizado para mostrar de forma mas clara el aumento en la medida de calidad cuando la punta de los dedos queda lo mas paralela posible a la superficie local del objeto.

ANEXO D: APLICACIÓN DE FUERZAS EN EL AGARRE

Para estos experimentos utilizamos la definición de agarre estable de [25] ellos definen el agarre como un movimiento en el que la mano de un robot o de un humano continua sosteniendo un objeto de forma estable. En nuestros experimentos decimos que un agarre es estable si la mano continua sosteniendo el objeto después de que se le quita la base que sostiene el objeto, por ej: podemos quitar la mesa o cualquier obstáculo donde se coloque el objeto. Esto es algo semejante a levantar el objeto y mantenerlo agarrado.

La mano robotica es la robonaut configurada con 12 GDL como se propone en este trabajo. En el momento de simulación siempre se la reconfigura por un material de goma (rubber) los contactos que genera son punto a punto. No se le coloco sof-finger debido a que esta parte de *Grasp*It no esta optimizada para un buen desempeño en el modo Dinámico. Se toman como medidas de calida las dos medidas que ofrece *GraspIt!*, ver sección 6.4. Ahora procedemos a realizar 5 experimentos para agarrar 5 objetos distintos. Cada experimento se repite 10 veces para poder obtener una probabilidad de agarrar estable del objeto con la mano propuesta.

Experimento No 1:

Mundo en *GraspIt!*: ver figura D.1

Nombre del mundo en *GraspIt!*: *flask.XML*

Objeto: Matraz; **Material**: stone; **masa:** 300 g; **Tipo de agarre:** cilindrico

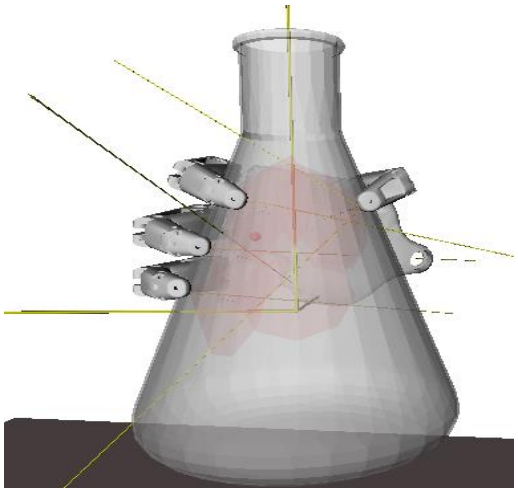


Figura D.1. Mano robotica agarrando un matraz

Tabla 10. Registro del primer expeimento.

No simulación	Epsilon	Volumen	Agarre estable
1	0.206	1.39	si
2	0.194	1.23	si
3	0.235	1.55	si
4	0.185	0.733	si
5	0.179	0.917	si
6	0.151	0.658	si
7	0.143	0.804	si
8	0.229	1.57	si
9	0.204	1.31	si
10			no

4 fallas de simulador de15 experimentos

Resultados experimento:

La probabilidad de agarre estable = 9/10=0.9. El 90% de los agarres fue estable.

El promedio de la medida de calidad *Epsilon=0.1726*

El promedio de la medida de calidad Volumen=*1.0162*

Experimento No 2:

Mundo en *GraspIt!*: ver figura D.2

Nombre del mundo: sphere.*XML*

Objeto: Esfera; **Material**: plástico; **masa:** 300 g; **Tipo de agarre:** esférico

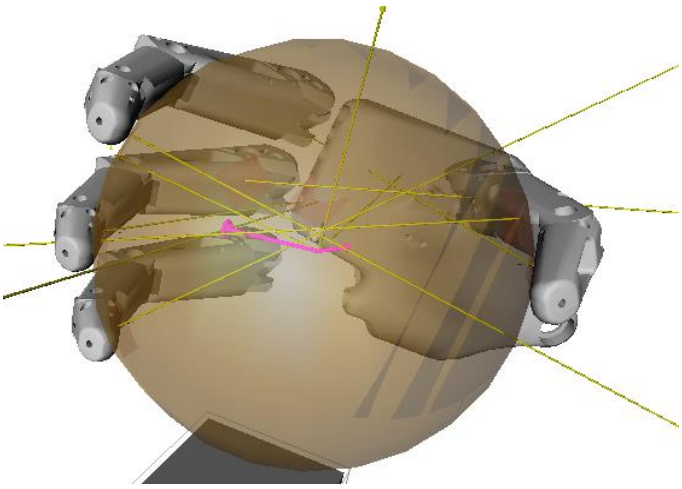


Figura D.2. Mano robotica agarrando una esfera.

Tabla 11. Registro del segundo expeimento.

No simulación	Epsilon	Volumen	Agarre estable
1	0.45	5.1	si
2	0.388	5.52	si
3	0.334	4.17	si
4	0.388	5.1	si
5	0.386	5.53	si
6			no
7	0.452	5.71	si
8			Fallo del simulador
9	0.415	5.54	si
10	0.418	5.15	si

Resultados experimento:

La probabilidad de agarre estable = 8/10=0.8. El 80% de los agarres fue estable.

El promedio de la medida de calidad *Epsilon=0.3231*

El promedio de la medida de calidad Volumen*=4.182*

Experimento No 3:

Mundo en *GraspIt!*: ver figura D.3

Nombre del mundo: book.*XML*

Objeto: libro; **Material**: plástico; **masa:** 440 g; **Tipo de agarre:** palmar

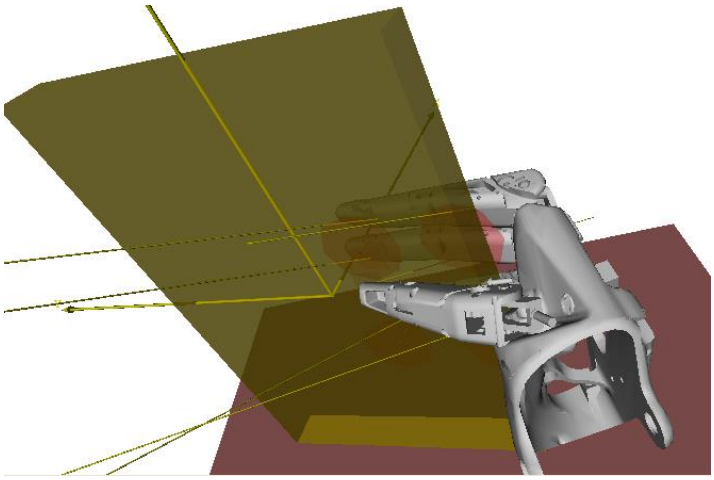


Figura D.3. Mano robótica agarrando un libro

Tabla 12. Registro del tercer expeimento.

No simulación	Epsilon	Volumen	Agarre estable
1	0.0101	0.0991	si
2			Fallo del simulador
3	0.0141	0.102	si
4	0.00272	0.0925	si
5	0.0061	0.1	si
6	0.0174	0.104	si
7	0.00124	0.0912	si
8	0.00325	0.0938	si
9			Fallo del simulador
10	0.0042	0.0925	si

Resultados experimento:

La probabilidad de agarre estable = 8/10=0.8. El 80% de los agarres fue estable.

El promedio de la medida de calidad *Epsilon=0.0061*

El promedio de la medida de calidad Volumen=*0.0755*

Experimento No 4:

Vista del mundo en *GraspIt!*: ver figura D.4

Nombre del mundo: glass.*XML*

Objeto: copa; **Material**: vidrio; **masa:** 200 g; **Tipo de agarre:** cilindrico

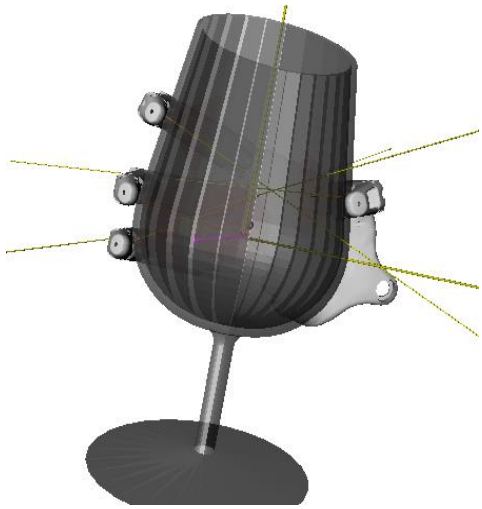


Figura D.4. Mano robótica agarrando una copa

Tabla 13. Registro del cuarto expeimento.

No simulación	Epsilon	Volumen	Agarre estable
1	0.172	0.431	si
2	0.148	0.243	si
3			Fallo del simulador
4			Fallo del simulador
5	0.126	0.227	si
6	0.127	0.222	si
7			Fallo del simulador
8	0.127	0.234	si
9			Fallo del simulador
10	0.13	0.244	si

Resultados experimento:

En este experimento el simulador presento bastante inestabilidad, por lo que en algunos casos no nos deja saber si el agarre será estable o inestable. De todas maneras se tomo el peor de los casos que es considerar la inestabilidad, aunque no siempre sea asi.

La probabilidad de agarre estable = 6/10=0.6. El 60% de los agarres fue estable.

El promedio de la medida de calidad *Epsilon=0.083*

El promedio de la medida de calidad Volumen=*0.1601*

Experimento No 5:

Vista del mundo en *GraspIt!*: ver figura D.5

Nombre del mundo: icosaedro.*XML*

Objeto: icosaedro; **Material**: madera; **masa:** 300 g; **Tipo de agarre:** cilindrico

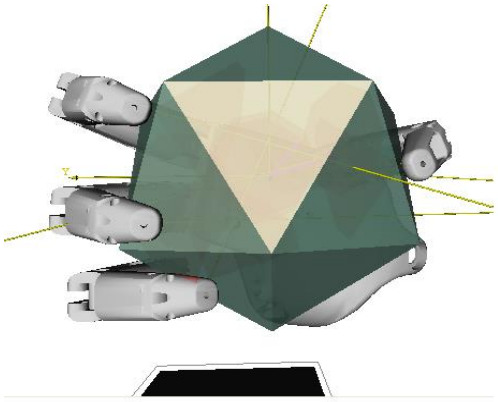


Figura D.5 Mano robótica agarrando un icosaedro

Tabla 14. Registro del quinto expeimento.

No simulación	Epsilon	Volumen	Agarre estable
1	0.319	4.68	si
2	0.25	3.28	si
3	0.15	2.61	si
4	0.306	4.3	si
5	0.218	3.14	si
6	0.312	4.62	si
7			fallo del simulador
8			no
9	0.293	2.72	si
10			Fallo del simulador

Resultados experimento :

La probabilidad de agarre estable = 7/10=0.7. El 70% de los agarres fue estable.

El promedio de la medida de calidad *Epsilon=0.1848*

El promedio de la medida de calidad Volumen=*2.35*

Resultados generales:

Probabilidad de que la mano agarre estable un objeto: 76%.

Masa promedio de objetos: 308 gramos.

Calidad Épsilon promedio para agarres de tipo potencia: *0.1908*

Calidad Volumen promedio para agarres de tipo potencia: *1.9227*

Calidad Épsilon para el agarre de tipo precisión: *0.0061*

Calidad Volumen para el agarre de tipo precisión: *0.0755*

En los resultados de calidad se puede ver como los agarres de potencias son más estables mientras que los agarres de precisión tienden a ser inestables por la poca calidad del agarre que tienen.

BIBLIOGRAFÍA

[1] J. Gómez, *Metodologías para el desarrollo de sistemas multi-agente*. Madrid: Departamento de Sistemas Informáticos y Programación Facultad de Informática, Universidad Complutense, 2003.

[2] E. Milgrom, P. Chainho, Y. Deville, R. Evans, P. Kearney and P. Massonet, “MESSAGE - Methodology for Engineering Systems of Software,” European Institute for Research and Strategic Studies in Telecommunications GmbH, Germany, Tech. rep. EURESCOM, 2001.

[3] F. Zambonelli, *Developing Multiagent Systems: The Gaia Methodology.* Italy: Universit`a di Modena e Reggio Emilia, 2004.

[4] C. Iglesias, M. Garijo, J. Gonzalez y J. Velasco, *Analysis and design of multiagent systems using MAS-CommonKADS*. Berlin: Springer-Verlag, 1998

[5] L. Padgham and M. Winik*OFF*, *Developing Intelligent Agent Systems: A practical guide*. Australia, RMIT University: John Wiley & Sons, Ltd, 2004.

[6] A. Miller, “Graspit!: A Versatile Simulator for Robotic Grasping," PhD. Thesis, Columbia University, Columbia (USA), 2001.

[7] M. Muñoz, *Diseño de una prótesis de mano en un ambiente de diseño asistido por computador*. Popayán, Colombia: Universidad Del Cauca, 2007 pp. 25-30.

[8] P. Shilane, P. Min, M. Kazhdan and T. Funkhouser, “The princeton shape benchmark,” In *Shape Modeling and Applications,* Genoa, Italy, 2004.

[9] J. García y D. Vallejos, “Entorno Gráfico De Un Entrenador Virtual De Prótesis De Mano,” Tesis de Pre-grado, Universidad Del Cauca Facultad De Ingeniería Electrónica Y Telecomunicaciones, Colombia, 2010, pp. 79.

[10] J. Delgado, “Detección de Colisiones Mediante Recubrimientos Simpliciales,” Tesis Doctoral, Universidad de Granada, España, 2006.

[11] E. Larsen, S. Gottschalk; M. Lin and D. Manocha, “Fast proximity queries with swept sphere volumes,” Dept. of Computer Science, University of North Carolina, Tech. Rep TR99-018,.1999.

[12] J. Pang and J. Trinkle, “Stability Characterizations of Fixtured Rigid Bodies with Coulomb Friction,” in *IEEE International Conference on Robotics and Automation*, San Francisco, CA, 2000, pp. 361-368.

[13] A. Miller and P. Allen, “Examples of 3D *Grasp* Quality Computations,” In *Proceedings IEEE International Conference on Robotics and Automation*. Detroit, U.S.A, 1999.

[14] A. Miller, S. Knoop, H. Christensen and P. Allen, “Automatic grasp planning using shape primitives,” in *Proceedings of the International Conference on Robotics and Automation IEEE*, 2003, pp. 1824–1829.

[15] D. Osswald, J. Martin, C. Burghart, W. Ralf, B. Heinz and Georg, *Integrating a Flexible Anthropomorphic Robot Hand into the Control System of a Humanoid Robot*. Germany: Department of Computer Science, University of Karlsruhe, 2003
[
[16] M. Ciocarlie, C. Goldfeder and P. Allen, “Dexterous *Grasp*ing via eigen*Grasp*s: A low-dimensional approach to a high-complexity problem,” in *Robotics: Science and Systems Manipulation Workshop - Sensing and Adapting to the Real World*, 2007.

[17] L. Ingber, *Very fast simulated re-annealing.* Monterey, CA: Physics Department, Naval Postgraduate School, 1989, pp 967–973.

[18] M. Santello, M. Flanders and J. Soechting. “Postural hand synergies for tool use*,” Journal of Neuroscience*, vol. 18, no. 23, pp. 10 105–10 115, 1998.

[19] C. Brown and H. Asada, ” Inter-finger coordination and postural synergies in robot hands via mechanical implementation of principal components analysis,” in *IEEE RAS Intl. Conference on Intelligent Robots and Systems*, 2007.

[20] M. Ciocarlie, C. Goldfeder, and P. Allen, "Dimensionality reduction for hand-independent dexterous robotic grasping," in *Proc. of the IEEE/RSJ Conf. on Intelligent Robots and Systems*, 2007, pp. 3270- 3275.

[21] T. Matei, M. Ciocarlie and P. Allen, “On-Line Interactive Dexterous *Grasp*ing,” In *Eurohaptics*. Madrid, España, 2008.

[22] D. Stewart and J. Trinkle, “An implicit time-stepping scheme for rigid body dynamics with coulomb friction,” In *Proc. of the 2000 IEEE Intl. Conf. on Robotics and Automation*, 2000, pp 162–169.

[23] C. Ferrari and J. Canny, “Planning optimal grasps” In *Proc. of the 1992 IEEE Intl. Conf. on Robotics and Automation*, 1992, pp. 2290–2295.

[24] K. Hsiao, P. Nangeroni, M. Huber, A. Saxena, and A. Y. Ng, “Reactive grasping using optical proximity sensors,” in *Conference on Robotics and Automation (ICRA)*, Kobe, Japan, 2009.

[25] I. kazuto and M. Takashi, “Hierarchical Control Method for Manipulating/Grasping Tasks using Multi-fingered Robot Hand,” In *Conference on Intelligent Robots and Systems*, Las Vegas, Nevada, 2003.

[26] J. Cabrera, “Conjunto normalizado de Interfaces para los componentes de un simulador virtual de Prótesis de mano,” Tesis de maestría, Universidad del Cauca, Colombia, 2009.