Integración de SOAR en ICa

Silvia Sánchez Herranz

Marzo de 2005

A mi padre, por sus sabios consejos, porque sin él todo ésto no hubiera sido posible, porque el tren pasa una sóla vez en la vida...

A mi madre y a mi hermana, por su gran corazón, por su paciencia y por su apoyo incondicional en momentos difíciles

A Miguel Manuel, por su valentía, por creer en mí, por su paciencia, por todo lo que me ha enseñado, por quererme tanto...

A Reichel, por ese "granito" de arena, por estar ahí, por ofrecerme siempre lo que está en su mano, por cuidarme

A Miguel Ángel, por haberme acompañado en este largo viaje, haciendo fácil lo difícil

A mi tía Cristina, por dejarse involucrar en esta "locura"

A Ceci, por los buenos momentos que hicieron todo más fácil

A Carlos, por escucharme y regalarme su ayuda, siempre de forma incondicional

A mis compañeros de ASLab, porque sin ellos todo hubiera sido más difícil

A Joaquín Maroto, Adolfo Lozano y Kurtis Fields, por sus consejos técnicos

A Ricardo, mi tutor, por su calidad como persona, su apoyo en momentos difíciles y porque en estos meses aquí, me ha enseñado una pequeña parte de TODO lo que sabe

GRACIAS

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

Alcanzar altos niveles de inteligencia en un sistema de control no es una tarea trivial. Este es el campo natural del control inteligente, donde se han aplicado multitud de tecnologías diversas que han permitido obtener buenos resultados en determinadas condiciones.

El estado actual del control inteligente es ciertamente confuso y más parece una colección heterogénea de tecnologías que una disciplina estructurada y metódica.

En cierta medida, el control inteligente, mas allá de la definición original de K.S. Fu, permanece como una colección de tecnologías de cómputo blando en la intersección de la inteligencia artificial y los sistemas de control. Algunos ejemplos de estas tecnologias son:

- Redes neuronales
- Sistemas borrosos
- Sistemas expertos
- Algoritmos genéticos
- Aprendizaje bayesiano

En nuestro grupo de investigacion ASLab (Autonomous Systems Laboratory) hemos venido aplicando este tipo de tecnología a multiples aplicaciones

—en procesos continuos y robótica— dando lugar a multiples desarrollos tanto experimentales como industriales (ver por ejemplo la Figura 1.1) y a una perspectiva particular sobre este campo [Sanz et al., 1999b].

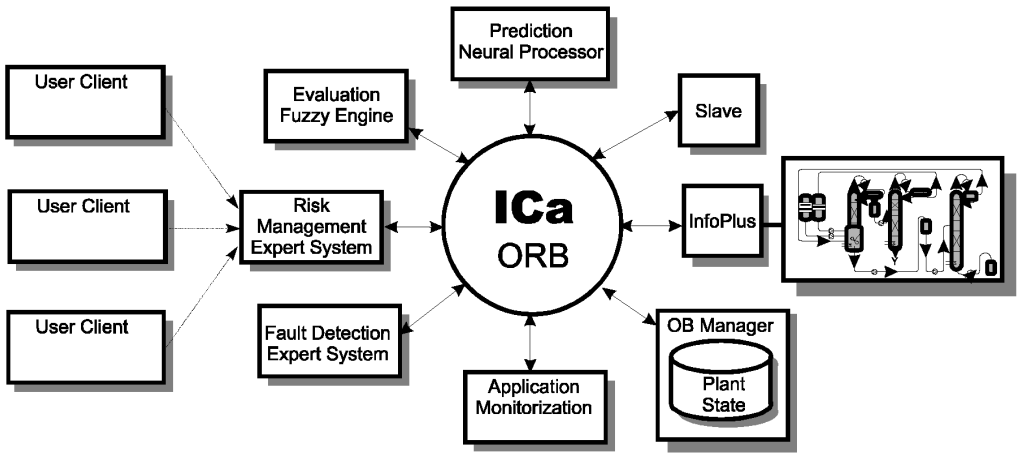

Figura 1.1: Riskman: un sistema de control inteligente de una refinería diseñado para la operación integrada de sistemas inteligentes de control [Sanz et al., 2000].

Al mismo tiempo, en la construcción de sistemas complejos de control intervienen muchos factores que determinan la necesidad de emplear arquitecturas software adecuadas. Esta adecuación se mide por el grado en que dichas arquitecturas permitan obtener tanto *caracterísitcas funcionales* determinadas (como son la satisfaccion de requisitos estrictamente necesarios en el ámbito del control) como *características no funcionales* (que son críticas a la hora de construir un sistema software complejo).

Este proyecto fin de carrera se sitúa en la confluencia de ambas necesidades —inteligencia y arquitectura software— tratando de ofrecer una solución general al problema concreto de la integración modular de agentes inteligentes construidos con tecnologías ajenas a nosotros.

El objetivo de este proyecto fin de carrera es estudiar y facilitar la integración del sistema **SOAR** [Laird et al., 1987] en la arquitectura **ICa**. **ICa** es una metaarquitectura de control integrado basada en el conjunto de estándares CORBA y que está siendo aplicada en la actualidad a la construción de la arquitectura de control reflexivo **SOUL**. **SOAR** es una plataforma para implementar agentes inteligentes, o usando las palabras de sus desarrolladores podemos decir que:

> **SOAR** *is a general cognitive architecture for developing systems that exhibit intelligent behavior. . . . In other words, our intention is for* **SOAR** *to support all the capabilities required of a general intelligent agent.*

Nuestra intención, al tratar de integrar **SOAR** en **ICa** es emplear todos los recursos y conocimientos en torno a esta arquitectura —empleada por múltiples investigdores de todo el mundo— para incorporar módulos con los mas altos niveles de intelgencia en las aplicaciones basada en **ICa**.

Este proyecto permitirá el desarrollo simplificado de componentes de control distribuido basados en el sistema **SOAR**.

1.2. Contexto

Este trabajo se enmarca dentro del proyecto de investigación a largo plazo CS^2 — *Complex Software-intensive Control Systems*. Este es un proyecto de investigacion en tecnologías software para sistemas complejos de control realizado por el *Autonomous Systems Laboratory* de la UPM, dentro del Departamento de Automática de la ETS de Ingenieros Industriales de la Universidad Politécnica de Madrid.

El objetivo de este proyecto investigador es la definición de arquitecturas de control integrado para la construcción de sistemas complejos de control y el desarrollo de tecnologías software para su construcción. Las líneas maestras son simples y guían todo el desarrollo del proyecto:

- Funcional
- Modularidad
- Seguimiento de estándares
- Reutilizabilidad
- Diseño basado en patrones
- Independencia del dominio

En este contexto, se ha definido una plataforma software genérica de base denominada **Integrated Control Architecture** (**ICa**) [Sanz et al., 1999c]

que proporciona los criterios de diseño software centrales. La arquitectura **ICa** es una metaarquitectura software que ofrece un serie importante de ventajas frente a otros enfoques :

Coherente y unificada: La arquitectura **ICa** proporciona integración, vertical y horizontal, total y uniforme; i.e. permite emplear un única tecnología en todos los niveles en la verticalidad del sistema de control (desde la decisión estratégica hasta los dispositivos empotrados de campo) así como la integración horizontal de unidades de negocio o empresas extendidas.

Clara: El modelo empleado en ella es un modelo preciso: el modelo de objetos distribuidos de tiempo real que constituye la base de las plataformas estado del arte en este campo.

Flexible y extensible: Permite su adaptación a distintos contextos de ejecución y distintos dominios al realizar un mínimo de compromisos de diseño; lo que permite el reúso modular de componentes y la incorporación de nuevos componentes en dominios concretos.

Abierta: Permite la interoperabilidad con sistemas ajenos (tanto heredados como futuros).

Portable: Gracias a basarse en estándares internacionales.

Esta arquitectura —o metaarquitectura como debe ser considerada en realidad [Sanz et al., 1999a]— se ha venido desarrollando en nuestro departamento durante los últimos años y, gracias a diferentes proyectos de I+D, se ha aplicado con éxito en multiples ámbitos de control, como por ejemplo:

- Control estratégico de procesos de fabricación de cemento [Sanz et al., 2001].
- Gestión de emergencias en plantas químicas [Sanz et al., 2000].
- Sistemas de monitorización distribuida de tiempo real de produccion y distribución de energía eléctrica[Clavijo et al., 2000].
- Robots móviles cooperantes[Sanz et al., 1999b].
- Bucles de control en red [Sanz, 2003].
- Protección de subestaciones eléctricas [Sanz, 2002].

La característica primaria de **ICa** es su enfoque modular. Los sistemas se construyen por medio de módulos reutilizables sobre *frameworks* de sistemas distribuidos de objetos de tiempo real. La organización concreta de los módulos viene dada por su arquitectura de aplicación, que se deriva de la metaarquitectura por medio del uso de patrones de diseño [Sanz and Zalewski, 2003]. Su despliegue se hace según las necesidades y restricciones de la aplicacion, explotando la reubicabilidad de los componentes (ver Figura 1.2).

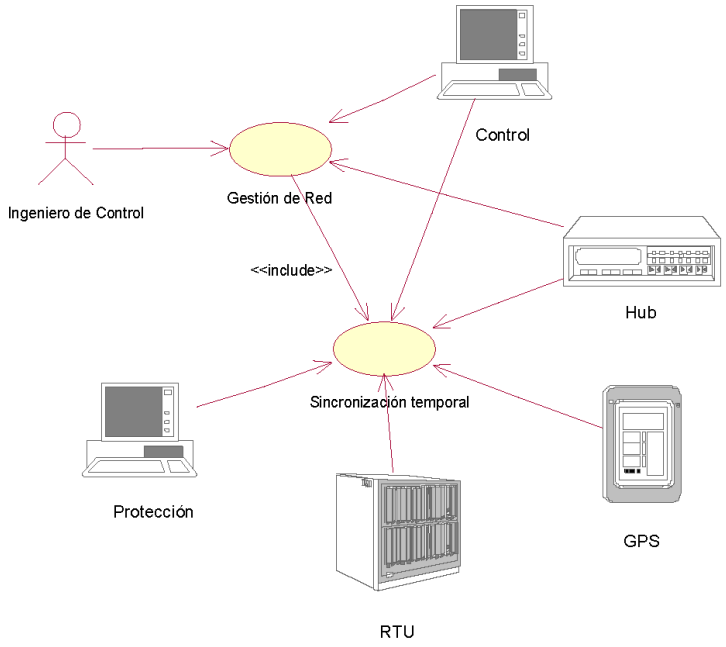

Figura 1.2: Diagrama de caso de uso para un despliegue de una aplicación de control y protección de redes.

En este proyecto fin de carrera se ataca la construcción de un módulo reutilizable específico para una cierta clase de arquitecturas que estamos investigando en la actualidad. La clase de arquitecturas se denomina **SOUL** y el nombre de este módulo es, obviamente, **SOAR**.

1.3. Objetivos del proyecto fin de carrera

Los objetivos de este proyecto son muy simples —que no fáciles de alcanzar— y se describen sumariamente diciendo que persigue la *implementación de un*

módulo reutilizable en el contexto de **ICa** *para la incorporación nativa de programas* **SOAR**.

Gracias a esta implementación se tendrá una comunicación entre **SOAR** y un módulo Pioneer 2AT-8 del contexto **ICa**.

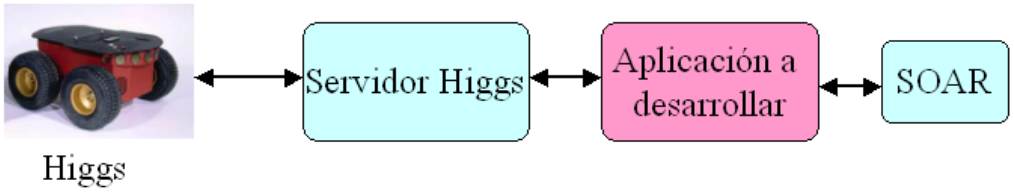

Figura 1.3: Esquema de la aplicación a desarrollar

1.4. Estrategia

Entrando más en detalle, el objetivo del presente proyecto consiste en controlar los estados de un robot Pioneer 2AT-8 perteneciente al laboratorio ASLab a través de **SOAR**, arquitectura general de consciencia, para desarrollar sistemas que presentan un comportamiento inteligente, desarrollado en diversas universidades estadounidenses: Carnegie Mellon Universidad, Universidad de Hertfordshire, Universidad de Michigan, Universidad de Nottingham, Universidad de Southern California y El grupo de agentes inteligentes en la Universidad de Portsmouth.

La estrategia para conseguir tal objetivo, comienza con el estudio independiente de **SOAR** y del robot Pioneer 2AT-8. Gracias al conocimiento de los dos elementos objeto de unión del proyecto, será más sencillo abordar el desarrollo de la solución.

El siguiente paso será establecer la comunicación entre ambos sistemas, para una vez conseguido tal objetivo, tratar de desarrollar pequeños programas de control que sean la base para futuros proyectos.

Para conseguir estos objetivos es necesario conocer el mecanismo de comunicación entre **Higgs** y los computadores del laboratorio, las variables de estado que nos proporcionan los distintos sensores y ultrasonidos de **Higgs**, estudio de **SOAR**, conocer las posibilidades de dicho software y cómo gestionar sus entradas salidas.

1.5. Estructura de la memoria

Esta memoria de Proyecto Fin de Carrera se estructura en siete capítulos y dos apéndices:

Capítulo 1 / Introducción: Este capítulo, en el que se presenta el proyecto y se introducen sus ideas básicas.

Capítulo 2 / SOAR: En el que se describe la arquitectura software **SOAR**, sus principios y estructura básica, la evolución y sus aplicaciones.

Capítulo 3 / Planteamiento del problema: En el que se describe el problema objeto del proyecto, su planteamiento y el enfoque que se le da.

Capítulo 4 / SGIO: En el que se describe el entorno de comunicaciones de **SOAR**, así como las razones por las que se eligió utilizar dicha comunicación.

Capítulo 5 / CORBA: En el que se presenta una de las herramientas principales en el desarrollo del proyecto: CORBA. En dicho capítulo se describe la implementación particular de uno de los brokers de *Open Source* que utiliza CORBA. Se narran los problemas que surgieron durante el desarrollo de esta parte.

Capítulo 6 / Desarrollo de la solución: En el que se describe la solución aplicada.

Capítulo 7 / Conclusiones y líneas futuras: En el que se detallan algunas de las conclusiones de este proyecto así como líneas futuras de investigación.

Apéndice A / SOAR Software License: Donde se detalla la licencia del software utilizado **SOAR**

Apéndice B / Herramientas software utilizadas: Donde se detallan las herramientas que se han utilizado para el desarrollo del proyecto.

1.6. Glosario de términos

Higgs: Nombre del robot Pioneer 2AT-8, objeto de comunicación del proyecto fin de carrera.

Estado (*state*): representación de la situación real en la solución de un problema.

Operador (*operator*): es lo que transforma un estado, hace cambios en la representación.

Goal: es el deseo final en la tarea de resolver un determinado problema.

Memoria de trabajo (*Working Memory*): o memoria a corto plazo, es el lugar donde **SOAR** representa la situación en que se encuentra la resolución de un problema en el momento actual de procesamiento.

Regla de producción (*Production Rule*): es similar al condicionante "*if-then*" en un lenguaje de programación convencional. La parte "*if*" de la producción se llama condición (*condition*) y la parte "*then*" se llama acción (*action*).

Impasse: carencia de conocimiento para poder alcanzar un objetivo, en otras palabras, situación de bloqueo en la que no se puede avanzar más.

Open Source: o *código abierto*, no sólo significa accesibilidad al código, si no que se tiene que cumplir:

- Libre redistribución
- Tiene que incluir el código fuente
- Permitir modificaciones y trabajos derivados, pudiendo ser distribuidos bajo los mismos términos de licencia del software original
- Integridad en el código fuente del autor
- No discriminar personas o grupos
- No discriminar su utilización para el esfuerzo específico de un grupo
- Distribución de la licencia
- Los derechos del programa no deben depender de otros programas con una distribución particular

- La licencia no debe restringir el uso de cualquier otro tipo de software
- La licencia debe ser independiente de la tecnología utilizada

Mapear: es el proceso de aplicar una regla de correspondencia entre dos dominios o conjuntos.

Capítulo 2

SOAR: una arquitectura cognitiva

2.1. ¿Qué es SOAR?

El objetivo principal del proyecto consiste en integrar **SOAR** en **ICa** para que se pueda controlar el comportamiento autónomo de **Higgs**, pero ¿qué es **SOAR**? Una respuesta rápida es que **SOAR** es un software inteligente, esto significa que dicho software es capaz de aprender.

No se va a profundizar en lo que se refiere al concepto de inteligencia, ya que se escapa del alcance del presente proyecto, pero por dar una pequeña definición:

Inteligencia es la capacidad para resolver problemas basándose en la comprensión de éstos y en el conocimiento de las herramientas disponibles para su resolución.

Con esta idea, se puede decir que **SOAR** es una arquitectura software desarrollada por varias universidades estadounidenses con el objetivo de implementar en máquinas para distintas aplicaciones un algoritmo de comportamiento y pensamiento que se acerque lo más posible al humano.

2.1.1. ¿Por qué SOAR?

En el laboratorio se dispone de un robot Pioneer 2AT-8, **Higgs** con el objetivo último de convertirlo en un robot autónomo, principalmente para fines industriales. El desarrollo necesario para alcanzar dicha meta requiere el esfuerzo y la aportación de diversos proyectos y trabajos de doctorado.

Un ente se considera autónomo cuando es capaz de desenvolverse por sí mismo ante una variedad de situaciones saliendo airoso de ellas. Para que ésto sea posible es necesario "algo" que piense dentro de él, que le dote de consciencia, tanto del mundo exterior como de sí mismo, que razone, que sepa administrar adecuadamente todo el conocimiento del que dispone..., en otras palabras, necesita una mente.

Debido a esta inquietud se eligió integrar **SOAR** en **ICa**. Gracias a **SOAR** el robot será consciente de sí mismo y del ambiente que le rodea, será capaz de aprender de su propia experiencia, de utilizar de forma óptima todos sus recursos... Los desarrolladores de **SOAR** han conseguido crear una estructura software capaz de llevar muchas de las tareas previamente mencionadas a cabo.

2.1.2. Origen de SOAR

Muchas ciencias (psicología, lingüística, antropología, biología...) son las que estudian el por qué del conocimiento humano, su naturaleza, la evolución del cerebro, el aprendizaje, las emociones, el procesamiento del lenguaje... Cada una de ellas, trata de dar solución al problema cognitivo desde su perspectiva, así, la ciencia cognitiva se originó con el deseo de integrar hábilmente estas ciencias tradicionalmente separadas.

Allen Newell, un fundador de la inteligencia artificial, considera a estas ciencias *microteorías*, pequeñas piezas de un gran puzzle hechas sin tener en cuenta la restricción de que para formar el puzzle tienen que encajar. Cree que cada una de estas *microteorías* tiene su historia en preguntarse acerca de determinado tipo de cuestiones y aceptar cierto tipo de respuestas, y ésto es tanto una ventaja como un inconveniente.

La ventaja de tener disciplinas independientes, con un objetivo común que es el fenómeno cognitivo, es que cada una ofrece respuestas muy especializadas dentro de su campo de investigación. Pero por otro lado, cuando se quieren unir todas estas disciplinas para dar una respuesta única al fenomeno

cognitivo aparece el inconveniente. Al formar muchas piezas independientes el puzzle, cuando se quiere reconstruir dicho puzzle no se sabe si dichas piezas serán suficientes para completar el gran puzzle o si habrá zonas en que dichas piezas se solapen, mientras que pueden quedar otras zonas sin cubrir.

Con este problema en mente y tratando de ir por delante, Newell y dos de sus estudiantes, alrededor de 1980, tratando de encontrar una solución, una Teoría Unificada Cognitiva (*Unified Theories of Cognition*) comenzaron a trabajar en **SOAR**.

Por tanto, **SOAR** es un candidato para ser la teoría unificada cognitiva (*unified theory of cognition*) emebebida en una arquitectura software, desarrollada por John Laird, Paul Rosenbloom, y Allen Newell a principios de 1982 en la universidad de *Carnegie Mellon*. Su desarrollo continúa con el trabajo de muchos investigadores repartidos por todo el mundo en el área de la Inteligencia Artificial, de la ciencia cognitiva y de la interacción hombre-máquina.

2.1.3. Visión general de la estructura SOAR

Las características principales en el desarrollo de la arquitectura **SOAR** son:

Conocimiento: SOAR representa uniformemente el conocimiento a corto plazo como una red de símbolos activos. El conocimiento a largo plazo es un conjunto de reglas condición-acción. Las condiciones de cada regla forman un patrón que se comparará con la red de símbolos activos. Cuando la condición de la regla es equivalente a dicha red, la regla se ejecuta mediante la realización de sus acciones. Estas acciones pueden implicar la inclusión (o supresión) de símbolos en la estructura de conocimiento a corto plazo. Cuando la condición de una regla se cumple, dicha regla se ejecuta llevando a cabo las tareas que tiene programadas. Estas acciones se darán como resultado la creación (o eliminación) de símbolos en la estructura del conocimiento a corto plazo.

Acción orientada al objetivo: para el control complejo, **SOAR** incluye una jerarquía orientada al objetivo, permitiendo la descomposición sucesiva de problemas en subproblemas. Así mismo, incluye mecanismos para la creación automática de nuevos objetivos como respuesta al conocimiento a largo plazo y a la situación actual.

Reacción: A diferencia de otros lenguajes de programación, **SOAR** no fuerza un flujo de control en serie. Las acciones tienen lugar en cualquier momento en función del cumplimiento de las condiciones en las reglas. Múltiples reglas pueden dispararse (*fire*) en paralelo, pero **SOAR** provee mecanismos de preferencia y la creación de subobjetivos para hacer frente a los posibles conflictos, si se dan.

Aprendizaje: **SOAR** incluye un mecanismo automático de aprendizaje inspirado en el concepto psicológico de troceado (*chunking*). **SOAR** compila secuencias de acciones en nuevas unidades de conocimiento (trozos) que pueden acortar los pasos de razonamiento cuando el sistema se encuentre ante situaciones similares en un futuro. Los nuevos trozos se unen de foma uniforme en un conjunto de reglas del sistema a largo plazo. Gracias a este mecanismo, **SOAR** puede aprender de forma incremental aprendiendo nuevos detalles sobre el mundo , así como disponer de una representación más eficiente de su conocimiento a largo plazo inicial.

Capacidades: Desde su desarrollo, la arquitectura **SOAR** ha sido utilizada para investigar una gran variedad de sistemas de desarrollo y aplicaciones comerciales.

2.2. Principios básicos

SOAR basa su diseño en una serie de principios que le guían en su intento de aproximarse al comportamiento racional:

- El número de mecanismos arquitecturales debe ser mínimo. En **SOAR**, hay un marco simple para todas las tareas y subtareas (espacio del problema), una representación simple del conocimiento permantente (*producciones*), una representación simple del conocimiento temporal (objetos con atributos y valores), un mecanismo simple de generar objetivos (mecanismo automático de generar subobjetivos) y un simple mecanismo de aprendizaje (*chunking*).
- Todas las decisiones son hechas a través de la combinación de conocimiento relevante en tiempo de ejecución. En **SOAR**, cada decisión está basada en la interpretación real de los datos sensoriales, el contenido de la memoria de trabajo creada por el espacio del problema y algún otro conocimiento obtenido de la memoria permanente.

2.2.1. Objetivo último de SOAR

Una vez analizado todo lo anterior, se puede resumir el objetivo final que se espera de la arquitectura **SOAR**:

- Trabajar con el amplio rango de tareas esperadas en un agente inteligente, desde la mayor rutina hasta la máxima dificultad, problemas de mente abierta.
- Representar y usar formas apropiadas de conocimiento, como puede ser procedural, declarativo, episódico e incluso basado en iconos.
- Poner en marcha todo el posible rango de métodos para la resolución de problemas.
- Interactuar con el ambiente exterior.
- Aprender acerca de todos los aspectos de las tareas y su propia ejecución.

En otras palabras, la intención última de **SOAR** es soportar todas las capacidades requeridas de un agente inteligente en general, simular lo más posible el comportamiento humano: aprender de la experiencia, con entrenamiento, por repetibilidad, prueba fallo-error...

2.3. Concepto de arquitectura

El concepto de arquitectura no es nuevo. Trabajando con computadores, a menudo se describen y comparan arquitecturas de hardware: un conjunto de elecciones del fabricante se hacen para tener un tamaño particular de memoria, comandos, procesadores... Visitando distintas tiendas de computadores, se comprueba que muchas configuraciones hardware son posibles. Las diferencias entre arquitecturas de hardware reflejan, en parte, diseños que no tienen la intención de ser óptimos bajo diferentes supuestos sobre el software que la arquitectura procesará. Así, una vez que las decisiones sobre los componentes de hardware y cómo ellos interactuan están hechas, la arquitectura resultante puede se evaluada (o comparada con otra máquina) simplemente considerando cómo de bien procesa software. En otras palabras, preguntar: ¿La máquina M corre aplicaciones A y B de forma eficiente? Es una pregunta más apropiada que la pregunta: ¿Trabaja la máquina M bien?.

Por hablar sobre cómo un hardware particular procesa aplicaciones software, se hablará de la aplicación particular de software que procesa un conjunto de tareas de alto nivel. Se piensa que es una aplicación software que además tiene una arquitectura. Si se quieren crear documentos u organizar hojas de cálculo (dos tareas comunes de alto nivel), se pueden elegir, entre procesador de texto y aplicaciones de hojas de cálculo. Como en el ejemplo de hardware, hay muchas posibles aplicaciones, con diferentes programas diseñados para ser óptimos bajo diferentes supuestos sobre tareas. La arquitectura particular como aplicación que se elige dicta las subtareas que serán fáciles y eficientes. Si, por ejemplo, el conjunto de tareas de alto nivel está escribiendo los capítulos en un libro de cálculo, se querrá elegir un procesador de texto que tenga funciones y comandos para el tratamiento de ecuaciones matemáticas; mientras que si lo que se quiere hacer es simplemente escribir cartas, entonces, las funciones mencionadas no son importantes.

Hay dos hilos comunes que marcan estos ejemplos de arquitectura.

1. El primero es la idea que para algunos niveles complejos del sistema, se puede hacer distinción entre mezcla de un conjunto de mecanismos y estructuras de la arquitectura por sí misma, y el contenido de estos mecanismos arquitecturales y estructuras de proceso. Así, está mezclado el hardware que procesa el contenido software a un nivel, y mezclas de mecanismos que procesan tareas de alto nivel conteniendo la siguiente. Otra forma de ver esta relación es notar que una arquitectura por sí misma no hace nada; requiere contenido para producir comportamiento. Por ello, hay que tener claro:

$$COMPORTAMIENTO = ARQUITECTURA \cdot CONTENIDO$$

2. El segundo hilo común corriendo a través de los ejemplos es que algunas arquitecturas particulares reflejan supuestos en la parte del diseño sobre características del contenido que la arquitectura procesará. Hay, por ejemplo, muchas arquitecturas hardware posibles que procesan el mismo software, pero, por ejemplo, máquinas con procesadores paralelos ejecutarán software mucho más rápido que máquinas en serie. Por similitud, muchas aplicaciones ejecutarán las mismas tareas de alto nivel, pero algunas darán comandos únicos que alcanzarán todas las subtareas, mientras que otras requerirán al usuario para que construya secuencias de comandos para conseguir el mismo efecto.

En general, el concepto de arquitectura es útil porque permite factorizar algunos aspectos comunes del ancho rango de comportamientos que caracterizan el contenido. Una arquitectura particular, esto es, un particular conjunto mezclado de mecanismos y estructuras, permanece como una teoría de qué es común entre muchos de los comportamientos al nivel superior. Usando esta idea, se puede definir una arquitectura cognitiva como una teoría de mezcla de mecanismos y estructuras que subrayan el fenómeno cognitivo humano.

Analizando lo que tienen en común los comportamientos cognitivos y el fenómeno explicado en las microteorías, parece que producir una teoría cognitiva unificada es un paso significante. Como un ejemplo de arquitectura cognitiva, **SOAR** es una teoría de lo que tienen en común la gran variedad de comportamientos de la inteligencia.

Para construir un modelo de comportamiento en **SOAR** se debe entender primero que aspectos del comportamiento soportará la arquitectura directamente y poner el trabajo preliminar para enlazar la arquitectura con el contenido.

2.3.1. Qué tienen los comportamientos cognitivos en común

Para entender cómo trabajan algunas estructuras computacionales, se necesita usar un modelo con algún comportamiento, como se dijo anteriormente, la arquitectura por sí sola no hace nada. Qué clase de comportamiento se debería modelar con **SOAR**? Una arquitectura cognitiva debe ayudar a producir comportamiento cognitivo. Leer, requiere cierta habilidad cognitiva, así mismo: resolver ecuaciones, hacer la cena, conducir un coche, contar una broma o jugar al baloncesto. De hecho, la mayoría de los comportamientos diarios parecen requerir algún grado de pensamiento que media entre las percepciones y las acciones. Porque cada arquitectura es una teoría sobre qué es común al contenido que procesa; **SOAR** es una teoría acerca de lo que tienen los comportamientos en común. En particular, la teoría de **SOAR** sugiere como principio básico tener al menos las siguientes características:

- Ser orientada al objetivo: a pesar de cómo se siente a veces, no se tropieza en la vida, actuando en modos que no están relacionados con nuestros deseos e intenciones. Si se quiere hacer la cena, se va al lugar apropiado (cocina, por ejemplo), se mezclan los ingredientes, se trocean, se cocinan y sazonan hasta que se produce el resultado deseado. Para

ello, será necesario aprender nuevas acciones: (poner el fuego lento en lugar de una fogata), o aprender el orden correcto de los pasos a realizar (añadir líquidos a sólidos, no viceversa), pero se aprende simplemente por el hecho de actuar varias veces de forma aleatoria.

- Reflejar un rico, complejo y detallado ambiente/medio: Aunque los modos es los que percibimos y actuamos en el mundo son limitados, el mundo que percibimos y en el que actuamos no es simple. Hay un elevadísimo número de objetos, cualidades de objetos, acciones y demás, cada una de las cuales debe ser la llave para entender cómo conseguir nuestros objetivos. Piensen acerca de los rasgos que tiene el paisaje al que respondes cuando vas un lugar nuevo, siguiendo las direcciones que te han dado para llegar a él. De algún modo, se reconoce el lugar real en todo su detalle, a partir, simplemente, de las descripciones que fueron dadas, y reaccionas de manera natural cuando reconoces el lugar al que tratabas de llegar, aunque no se haya estado previamente en dicho lugar.

- Requiere una gran cantidad de conocimiento: Tiene que intentar describir todas las cosas que conoce para saber cómo resolver ecuaciones.

  Algunas de ellas son obvias: poner la variable a resolver en uno de los lados del igual, mover términos constantes mediante suma o resta y coeficientes mediante multiplicación o división. Pero también necesita saber cómo hacer multiplicaciones y restas, nociones básicas de números, cómo leer y escribir números y letras, cómo coger el lápiz y usar una goma, qué hacer si el lápiz se rompe o si la habitación se queda sin luz...

- Requiere el uso de símbolos y abstracciones: Veamos de nuevo el ejemplo de hacer la cena. En frente de usted hay un pavo de cinco kilos, algo que has comido pero nunca has cocinado previamente. ¿Cómo sabe que es un pavo? Ha visto pavos antes, pero nunca antes ha visto *este* pavo y quizá nunca ha cocinado ninguno. De algún modo parte del conocimiento que posee puede ser obtenido por algo más que lo que es en sí la percepción en todo su detalle. Esto se llamará símbolo (o conjunto de símbolos). Porque el hombre representa el mundo internamente usando símbolos, el hombre crea abstracciones. Se puede parar de ver el pavo, pero se puede seguir pensando en él como pavo. Incluso, se puede continuar pensando en el pavo si se decide cenar fuera y salir de la cocina.

- Ser flexible, y función del ambiente: Conducir al colegio por el mismo camino que se hace normalmente, se divisa un atasco, el conductor decidirá girar en la próxima esquina para evitarlo. Bajando por una calle sin movimiento, una pelota aparece en frente del coche. Mientras el conductor frena, inmediatamente mira en la dirección por la que apareció la pelota, buscando a un niño que probablemente vaya a recogerla. Como estos ejemplos muestran, la consciencia del proceso mental humano no es sólo un hecho de pensar por delante, es también un hecho de pensar paso a paso con el mundo.

- Requiere aprendizaje del ambiente y la experiencia.

  La gente no nace sabiendo cómo contar bromas, resolver ecuaciones, jugar baloncesto o hacer la cena. Aún así, la gente llega a ser muy habilidosa (e incluso experta) en una de estas o muchas más actividades y cientos de otras. Efectivamente, quizá la cosa más admirable sobre la gente es cómo aprenden a hacer muchas cosas con poco, pareciendo que nacen sabiendo cómo hacerlo.

- Hay muchas otras propiedades que marcan las capacidades cognitivas (por ejemplo, la cualidad de ser consciente de uno mismo), y hay otros modos de interpretar los mismos comportamientos previamente mencionados.

  ¿Qué significa para **SOAR** como arquitectura reflejar este particular punto de vista acerca de lo que es común en cognición? Significa que los mecanismos y estructuras que se implementan en **SOAR** hará esta visión fácil de implementar, siendo otros puntos de vista más difíciles. Después de haber construido un modelo con esta arquitectura, será sencillo describir el comportamiento del modelo como orientado al objetivo, porque la arquitectura soporta esta visión directamente. Si la teoría es útil, permite modelar comportamientos que se quieren modelar en un modo que parece fácil y natural.

Habiendo identificado las propiedades comunes del comportamiento cognitivo que **SOAR** debe soportar, se deberían motivar las estructuras y mecanismos específicos uniéndolos con estas propiedades.

Recordando la ecuación:

$$COMPORTAMIENTO = ARQUITECTURA \cdot CONTENIDO$$

, es claro que si se quiere ver cómo la arquitectura contribuye al comportamiento, entonces de necesita explorar la arquitectura en términos de un

contenido en particular. Por ejemplo, describiendo cómo **SOAR** soporta la importante característica de *ser orientado al objetivo* no producirá por sí mismo comportamiento. Se necesita ser orientado al objetivo sobre algo.

2.3.2. Comportamiento como movimiento a través del espacio de problema

Cada vez, se dan unas circunstancias diferentes para llevar a cabo un mismo objetivo. Si se intentan dar todas las circunstancias las cuales darán lugar a conseguir el objetivo, nos superarían, sería prácticamente inviable. A pesar de que haya muchas posibilidades, sólo se podrá elegir una en el momento en el que se está en el juego real. Por ello, el sujeto deberá ser capaz de actuar basándose en las elecciones que tienen sentido en ese momento, pero a su vez, teniendo en cuenta las consecuencias que puede haber en el futuro.

La idea de los espacios del problema data a los primeros días de las investigaciones en Inteligencia Artificial y modelos cognitivos usando computadores, y es una de las construcciones computacionales principales en la teoría cognitiva de **SOAR**. Podría ser duro ver el problema de los espacios en el cerebro cuando se focaliza en una masa de neuronas.

2.4. Estructura SOAR

El diseño de **SOAR** está basado en la hipótesis de que todo el comportamiento enfocado a conseguir un objetivo puede ser moldeado como una selección y aplicación de operadores (*operators*) hacia un estado.

Cuando **SOAR** está corriendo, continuamente trata de aplicar el operador actual y seleccionar el próximo, ya que un estado, sólo puede tener un operador activo a la vez, hasta que el *goal* es alcanzado. La selección y aplicación de los operadores se puede ver la la Figura 2.1.

SOAR tiene memorias separadas, con diferentes representaciones, para describir la situación real y su conocimiento a largo plazo. En **SOAR**, la situación actual se compone de los datos recibidos por los sensores, resultados de deducciones intermedias, objetivos activos y los operadores activos que se encuentran en la memoria de trabajo.

La memoria de trabajo está organizada en objetos y los objetos son des-

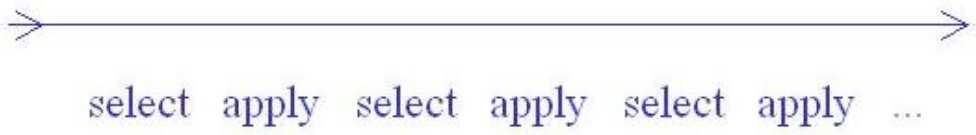

Figura 2.1: Selección y aplicación de los operadores

critos en términos de atributos; el valor de los atributos puede corresponder a sub-objetivos, por tanto, la descripción del estado tiene una organización jerárquica.

El conocimiento a largo plazo, que especifica cómo responder ante distintas situaciones en la memoria de trabajo, se puede decir que es el programa que hay que programar para **SOAR**. Así, una arquitectura **SOAR** no puede solucionar problemas si no dispone del conocimiento a largo plazo. No hay que confundir, por tanto, una arquitectura **SOAR** con un programa **SOAR**;

- Arquitectura **SOAR** hace referencia al sistema estructural de **SOAR**, tipos de memoria, operadores, atributos, *impasses*..., común para cualquier usuario.
- Programa **SOAR** hace referencia al conocimiento que se le ha de añadir a dicha estructura. Un programa contiene el conociemiento que ha de ser usado para solucionar una tarea específica (o un conjunto de tareas), incluyendo información acerca de cómo seleccionar y aplicar operadores para transformar los estados del problema y cómo reconocer cuando el objetivo ha sido alcanzado.

2.4.1. Funciones para la resolución de problemas

Todo el conocimiento a largo plazo está organizado en torno a funciones de operadores de selección y aplicación. Dichas funciones están compuestas por cuatros tipos distintos de conocimiento:

- Conocimiento para seleccionar un operador.
  1. Propuesta de un operador (*Operator Proposal*): conocimiento de que un operador es apropiado para la situación actual.

2. Comparación de operadores (*Operator Comparison*): conocimiento para comparar los distintos candidatos.
3. Selección de operadores (*Operator Selection*): conocimiento para seleccionar un sólo operador basándose en las comparaciones.

- Conocimiento para aplicar un operador.

4. Aplicación de un operador (*Operator Application*): conocimiento sobre cómo un operador específico modifica el estado.

Adicionalmente hay un quinto tipo de conocimiento en **SOAR** que está conectado indirectamente tanto con la selección como con la aplicación de un operador:

5. Conocimiento de inferencias monotónicas que pueden ser hechas sobre el estado (*estate elaboration*). Las elaboraciones del estado, afectan indirectamente a la selección y aplicación de los operadores ya que crean nuevas descripciones de la situación actual que pueden sugerir la selección y aplicación de operadores.

Estas funciones para la resolución del problema son las primeras en la generación de comportamiento en **SOAR**. Cuatro de las funciones requieren la recuperación del conocimiento a largo plazo que es relevante para la situación actual: elaboración del estado, proponer operadores válidos, comparar dichos candidatos y aplicar el operador modificando el estado. Estas funciones son conducidas por el conocimiento codificado en un programa **SOAR**.

SOAR representa dicho conocimiento como reglas de producción (*production rules*). Cuando las condiciones se dan en una situación actual, definidas por la memoria de trabajo, la producción entra en juego y se llevará a cabo, significando que las acciones son ejecutadas, haciendo cambios en la memoria de trabajo.

La otra función, seleccionar el operador actual, implica tomar una decisión una vez se haya recuperado el suficiente conocimiento. Ésto es realizado por el proceso de decisión de **SOAR**, el cual, es un procedimiento mixto que interpreta las preferencias que han sido creadas por las funciones de recuperación del conocimiento. Las funciones que recuperan conocimiento y llevan a cabo el proceso de tomar una decisión se combinan para dar lugar al ciclo de decisión (*decision cycle*) de **SOAR**.

Cuando el conocimiento para llevar a cabo las funciones de resolución de un problema no están disponibles directamente en las producciones, **SOAR** es

incapaz de progresar y alcanza un *impasse*. En **SOAR** hay tres tipos posibles de *impasses*:

1. No se puede seleccionar un operador porque ninguno ha sido propuesto.
2. No se puede seleccionar un operador porque se han propuesto múltiples operadores y las comparaciones disponibles son insuficientes para determinar cuál debe ser seleccionado.
3. Se ha seleccionado un operador, pero no hay suficiente conocimiento para aplicarlo.

En respuesta a un *impasse*, la arquitectura **SOAR** crea un subestado en el cual los operadores pueden ser seleccionados y aplicados para generar o recuperar el conocimiento que no fue disponible directamente; el objetivo en dicho subestado es resolver el *impasse*.

2.4.2. Proposición de operadores válidos

El primer paso antes de seleccionar un operador, es proponer uno o más operadores. Los operadores son propuestos gracias a reglas que analizan las características del estado actual. Cuando se lanza un programa **SOAR** los operadores propuestos se corresponden con las distintas acciones posibles que se pueden llevar a cabo partiendo del estado inicial.

2.4.3. Comparación de operadores válidos: Preferencias

El seguno paso para seleccionar un operador es evaluar o comparar a los operadores candidatos; ésto se hace mediante reglas que testean los operadores propuestos y el estado actual para luego crear preferencias. Por ejemplo, una preferencia podría decir que el operador A es mejor elección que el operador B en un momento determinado, o incluso, decir que el operador A es lo mejor que se puede hacer en dicho momento.

2.4.4. Seleccionar un único operador

SOAR intenta seleccionar un únicop operador basado en las preferencias disponibles para los operadores seleccionados. Se pueden dar cuatro situaciones:

1. Las preferencias disponibles prefieren sin ningún tipo de ambigüedad un único operador.
2. Las preferencias disponibles sugieren múltiples operadores y prefieren un subconjunto de operadores en el cual se puede elegir aleatoriamente cualquiera de los operadores de dicho subconjunto.
3. Las preferencias disponibles sugieren múltiples operadores pero no se da ni el caso 1 ni el 2.
4. Las preferencias disponibles no sugieren ningún tipo de operador.

En el primer caso, dicho operador es seleccionado. En el segundo caso, uno de los operadores pertenecientes al subconjunto es seleccionado aleatoriamente. En el tercer y cuarto caso, **SOAR** alcanza un *impasse* en la resolución del problema y se crea un nuevo subestado.

2.4.5. Aplicación del operador

Cuando un operador es aplicado, produce cambios en el estado; los cambios específicos dependen del operador y del estado actual.

Principalmente, hay dos enfoques para modificar el estado: directa e indirectamente.

- Indirectamente: son utilizados en programas **SOAR** que interactúan con el ambiente exterior: el programa **SOAR** manda comandos al ambiente exterior y monitoriza el ambiente exterior mediante sus cambios. Los cambios se reflejan en una descripción de datos actualizados provenientes de los sensores.
- Directos: éstos se corresponden cuando **SOAR** resuelve problemas "dentro de sí mismo". Los porgramas **SOAR** que no interactúan con el ambiente exterior sólo pueden hacer cambios directos al estado.

La resolución de problemas externa e interna no debe verse como actividades mutuamente excluyentes en **SOAR**. Los programas **SOAR** que interactúan con el ambiente exterior generalmente tendrán operadores que hagan tango cambios directos como indirectos al estado.

Cuando **SOAR** intenta solucionar un problema internamente, debe saber cómo modificar las descripciones del estado apropiadamente cuando un operador va a ser aplicado. Si está resolviendo el problema en un ambiente exterior, debe saber los posibles comandos motores a los que tiene acceso con la intención de hacer un cambio en el ambiente.

2.4.6. Deducciones del estado

Hacer deducciones sobre el estado es otro de los papeles que el conocimiento a largo plazo puede realizar. Un conocimiento elaborado pude simplificar la codificación de los operadores, ya que añade una serie de características al estado que, por consiguiente, no tienen que ser incluidas explícitamente en la aplicación del operador. En **SOAR**, estas deducciones serán retiradas cuando la situación cambie, como puede suceder con la aplicación de operadores o con cambios procedentes de los datos sensoriales.

2.4.7. Espacios de problema

Si se quisiera construir un sistema **SOAR** que trabajase con un gran número de diferentes tipos de problemas, se necesitaría incluir un gran número de operadores en el programa **SOAR**. Para un problema específico y un determinado escenario en la resolución del problema, sólo se necesitarían un subconjunto de todos los posibles operadores que son realmente relevantes.

Los programas **SOAR** están organizados implícitamente en términos de espacios de problema porque las condiciones para proponer operadores serán estrictas, en el sentido de que un cierto operador será considerado sólo cuando sea verdaderamente relevante.

Típicamente, cuando **SOAR** soluciona un problema en dicho espacio del problema, no genera explícitamente todos los estados, si no que los examina y luego crea una ruta. En cambio, cuando **SOAR** se encuentra en un estado específico en un momento dado, representado en la memoria de trabajo, trata de seleccionar un operador que lo moverá a un nuevo estado. Utiliza cualquier conocimiento que tenga acerca de seleccionar los operadores dados en la

situación actual, y si su conocimiento es suficiente, intentará moverse hasta conseguir el objetivo.

2.4.8. Memoria de trabajo

La memoria de trabajo sustenta el estado actual, operador (así como cualquier subestado y operadores generados por los *impasses*) y un conocimiento a corto plazo, reflejando el conocimiento actual del mundo y del estado en la resolución del problema.

La memoria de trabajo contiene elementos llamados elementos de la memoria de trabajo, (*Working Memory Elements (WME´s)*). A partir de ahora, se hará referencia a dichos elementos de la memoria de trabajo como: WME´s.

Cada WME contiene una pequeña cantidad de información. Por ejemplo, un WME podría decir: "B1 es un bloque". Varios WME´s en conjunto pueden ofrecer más información sobre un mismo objeto. ("B1 se llama A", "B1 está encima de la mesa"...) Estos WME´s están relacionados porque todos ellos contribuyen a la descripción de algo que es conocido internamente en **SOAR** como B1. B1 sería un identificador; un grupo de WMEs que comparten un mismo identificador es denominado objeto en la memoria de trabajo. Cada WME describe un diferente atributo de un objeto; cada atributo tiene un valor asociado. Además, cada WME es un identificador-atributo-valor triple y todos los WME´s con el mismo identificador son parte del mismo objeto.

Los objetos en la memoria de trabajo están unidos a otros objetos: el valor de un WME puede ser un identificador de otro objeto. Todos los objetos en la memoria de trabajo deben estar unidos a un estado, bien directa o indirectamente (a través de otros objetos). Los objetos que no están unidos a un estado serán automáticamente eliminados de la memoria de trabajo por la arquitectura **SOAR**.

WME´s son llamados a veces "aumentos", porque aumentan al objeto, dando más detalle acerca de él. Mientras que estos dos términos pueden resultar algo redundantes, WME es un término que se usa más a menudo para referirse a los contenidos de la memoria de trabajo, mientras que "aumento" es un término que se usa más a menudo para referirse a la descripción de un objeto.

El atributo de un "aumento" es generalmente una constante, como `red`, o un identificador, como `06`. Cuando el valor es un identificador, se refiere a un

objeto en la memoria de trabajo que puede tener una subestructura adicional. En términos semánticos, si un valor es constante se trata de un nodo terminal sin links; si es un identificador, entonces no es un nodo terminal.

La memoria de trabajo es un conjunto, lo que significa que no puede haber nunca dos elementos al mismo tiempo en la memoria de trabajo que tengan el mismo identificador-atributo-valor (ésto lo previene la arquitectura). Aún así, está permitido tener múltiples WME´s que tengan el mismo identificador y atributo pero que cada uno tenga un valor diferente. Cuando ésto pasa, se dice que el atributo es un atributo multivalor, abreviando muchas veces a multi-atributo.

Un objeto es, en cierto modo, definido por sus "aumentos" y no por su identificador. Por consecuente corriendo el mismo programa **SOAR**, podría haber un objeto con exactamente los mismos "aumentos", pero un identificador diferente, y el programa razonará obre el objeto apropiadamente. Los identificadores son marcas internas para **SOAR**, por tanto pueen aparecer en la memoria de trabajo pero nunca aparecerán en producción.

No hay una relación predefinida entre objetos en la memoria de trabajo y los objetos reales en el mundo exterior. Los objetos en la memoria de trabajo se pueden referir a objetos reales, como bloque A; una característica de un objeto, como que sea de color rojo; una relación entre objetos: como estar encima; clases de objetos, como bloques... Los nombres de los atributos y valores no tienen significado para la arquitectura **SOAR** (salvo aquellos que son creados por la arquitectura).

Los elementos en la memoria de trabajo vienen de cuatro fuentes:

1. Las acciones de producción crean la mayoría de los elementos. Las acciones de producciones no deben crear o modificar los WME´s creados por el proceso de decisión o el sistema de entrada-salida.

2. El proceso de decisión crea automáticamente algunos aumentos especiales (tipo, superestado, *impasse*...) cuando un estado se crea. Los estados son creados durante la inicialización (el primer estado) o gracias a un *impasse* (subestado).

3. El proceso de decisión crea el operador aumento del estado basado en las preferencias. Esto toma nota en la selección del operador actual.

4. Las entradas-salidas del sistema crean WME´s en el *link* de entrada para datos sensoriales.

Los elementos en la memoria de trabajo se eliminan de seis maneras:

1. El proceso de decisión elimina automáticamente todos los aumentos de estado que crea cuando el *impasse* que conduce a su creación es resuelto.
2. El proceso de decisión elimina el operador aumento del estado cuando el operador no se selecciona como operador actual.
3. Las acciones de producción que utilizan `reject` preferencias eliminan WME´s.
4. Las entradas de WME´s se eliminan cuando las producciones que crearon no se utilizan más.
5. El sistema de entrada-salida elimina datos sensoriales del *link* de entrada cuando ya no es válido.
6. La arquitectura elimina automáticamente WME´s que no están unidos a un estado (porque otros WME´s han sido eliminados).

Para el resto, el usuario es libre de usar cualquier atributo y valor que pueda ser apropiado para una tarea. De todos modos, los estados tienen especiales aumentos que no pueden ser creados, eliminados o modificados directamente con las reglas. Éstos incluyen los aumentos creados cuando un estado es creado y el aumento operador del estado indica al operador actual.

Las preferencias permanecen el la memoria de preferencia (*preference memory*) donde no pueden ser comprobadas por las producciones; de todos modos, preferencias aceptables pueden pertenecer tanto a la memoria preferente como a la memoria de trabajo. Haciendo preferencias aceptables en la memoria de trabajo, las preferencias aceptables pueden ser testeadas por las producciones permitiendo a los operadores candidatos comparar antes de ser elegidos.

2.4.9. Producciones

SOAR representa el conocimiento persistente como producciones que son almacenadas en la memoria de producción, ilustrado en Figura 2.2

Cada producción tiene un conjunto de condiciones y un conjunto de acciones. Si las condiciones de una producción se dan en la memoria de trabajo, la producción se llevará a cabo y las acciones se realizarán.

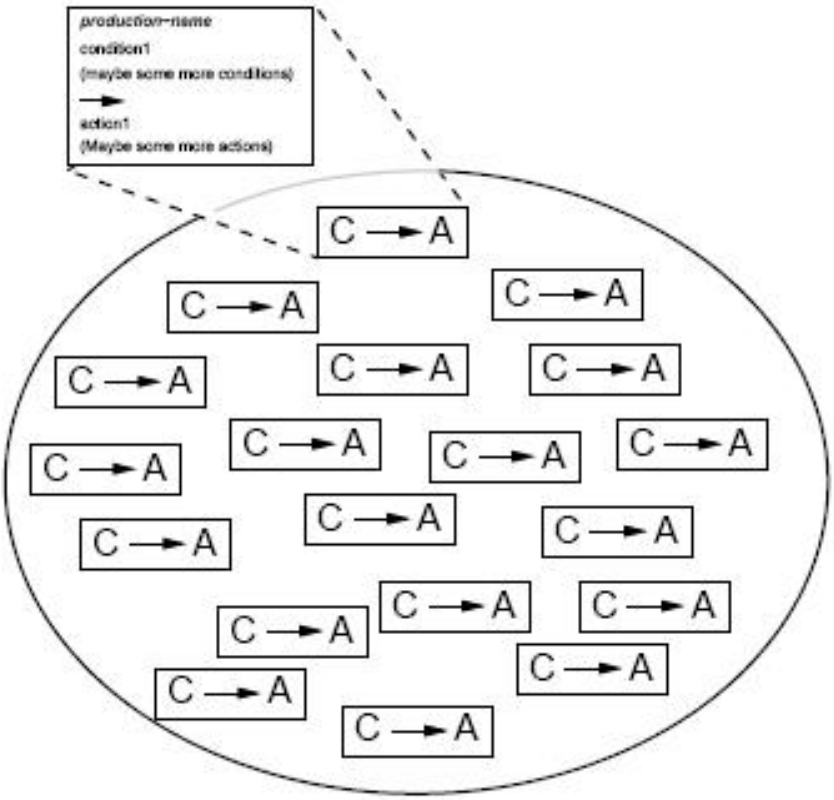

Figura 2.2: Visión abstracta de la memoria de producción.

- Estructura de una producción.

  En la forma más simple de una producción las condiciones y acciones se refieren directamente a la presencia (o ausencia) de objetos en la memoria de trabajo.

  Las condiciones de una producción pueden también especifiar la ausencia de patrones en la memoria de trabajo.

  El orden de las condiciones de una producción no importan en **SOAR**, excepto que la primera condición deba testear directamente el estado. Internamente, **SOAR** reordenará las condiciones para que el proceso de ejecución sea más eficiente. Éste es un detalle mecánico que no interesa a los usuarios. De todas formas, se deben escribir las producciones en la pantalla o salvarlas en un archivo; si no están en el orden que se espera, es porque **SOAR** las ha reordenado.

- Papeles arquitecturales de producciones.

  Las producciones **SOAR** pueden realizar distintos papeles, incluyendo los tres conocimientos recuperados, las funciones para la resolución del problema y el estado de la función de elaboración, como ya se describió:

  1. Propuesta de operadores
  2. Comparación de operadores
  3. (La selección de operadores no es un acto del conocimiento recuperado)
  4. Aplicación de operadores
  5. Elaboración del estado

  Una única producción no podría realizar más de dichos papeles (excepto para proponer un operador y crear una preferencia absoluta para él). Aunque las producciones no son declaradas para ser de un tipo u otro, **SOAR** examina la estructura de cada producción y clasifica las reglas automáticamente basándose en si proponen o comparan operadores, aplican o elaboran el estado.

- Acciones de producción y persistencia.

  Las dos acciones principales de una producción son crear preferencias para una selección de operadores y crear o eliminar WME´s. Para la propuesta de un operador y comparación, una producción crea preferencias para la selección de operadores. Estas preferencias persistirían sólo tanto como la producción que las creó continúa activa. Cuando la producción nunca más se vuelve a dar, la situación ha cambiado y por tanto hace que la preferencia no sea ya relevante. **SOAR** elimina automáticamente las preferencias en los casos mencionados.

  Similarmente, las elaboraciones de estado que son simples deducciones son válidas sólo tanto tiempo como la producción esté en juego. WME´s creados como elaboraciones de estado también tienen I-support y permanecen en la memoria de trabajo tanto tiempo como la producción *instantation* que los creó continua entrando en juego en la memoria de trabajo. Las proposiciones de operadores serán retiradas cuando no se aliquen por más tiempo a la situación actual.

  De todos modos, las acciones de producciones que aplican un operador, bien añadiendo o eliminando elementos de la memoria de trabajo, necesitan resistir incluso después que el operador ya no es seleccionado en más ocasiones y la producción de la aplicación del operador no entra en

juego más. Por ejemplo, si se quiere colocar un bloque encima de otro, la acción de poner el segundo bloque sobre el primero elimina el hecho de que el primer bloque no tiene nada encima, por tanto, la condición nunca será satisfecha.

Así, las producciones de aplicación de operadores no retiran sus acciones, incluso si no van a entrar en juego en la memoria de trabajo. Ésto es llamado O-soporte (soporte de operador). WME´s que participan en la aplicación de operadores son mantenidos a través de la existencia de un estado en el cual el operador es aplicado, a no ser que sea explícitamente eliminado. WME´s son eliminados por una acción de rechazo de una regla de aplicación de operador.

Si un WME recibe O-soporte o I-soporte, es determinado por la estructura de la producción inicial que crea el WME. O-soporte es dado sólo a los WME´s creados por una producción de aplicación de operador.

Una producción de aplicación de operador testea el operador actual de un estado y modifica el estado. Ésto es, un WME recibe O-soporte si es por un aumento del estado actual o una subestructura del estado y las condiciones de la inicialización que lo crearon comprueba aumentos del operador actual.

Cuando las producciones entran en juego, todas las producciones que tienen sus condiciones se ejecutarán creando o eliminando WME´s. También, WME´s y las preferencias que pierden I-soporte son eliminadas de la memoria de trabajo. Ésto es, varios WME´s nuevos y preferencias pueden ser creadas y varios WME´s exitentes y preferencias pueden ser eliminadas al mismo tiempo. Ésto no pasa así literalmente al mismo tiempo, pero el orden de ocurrencia no es importante, desde un punto de vista funcional pasa en paralelo.

2.4.10. Mermoria de preferencia: selección de conocimiento

La selección del operador actual es determinada por las preferencias en la memoria de preferencias. Las preferencias son sugerencias u órdenes sobre el operador actual, o información sobre cómo sugerir ciertos operadores comparados con otros. Las preferencias se refieren a los operadores usando el identificador de un WME que está para el operador. Una vez que las preferencias han sido creadas de un estado, el proceso de decisión las evalúa para seleccionar el operador actual para el estado.

Para que un operador sea seleccionado tendrá que haber por lo menos una preferencia hacia él, específicamente, una preferencia para decir que el valor es un candidato para el atributo del operador de un estado (ésto se hace bien con una preferencia `aceptable` o `requerida`). También pueden ser otros valores como `el mejor`.

Las preferencias permanecen en la memoria de preferencia hasta que son eliminadas por alguna de las razones mencionadas anteriormente.

- Preferencias semánticas.

  Sólo un único valor puede ser seleccionado como el operador actual, todos los valores son mutuamente excluyentes. Adicionalmente, no hay una transición implícita en la semántica de las preferencias. Si A es indiferente a B y B es indiferente a C, A y C no serán indiferentes entre sí a no ser que haya una preferencia que diga que A es indiferente a C.

  Aceptable (+) una preferencia aceptable dice que un valor es un candidato para la selección. Todos los valores, excepto los que requieren preferencias, deben tener una preferencia aceptable si quieren ser seleccionados. Si hay un único valor con una preferencia acpetable (y ninguno con una preferencia requerida), dicho valor será seleccionado a no ser que tenga una preferencia de rechazo o prohibitiva.

  Rechazo (-) esta preferencia dice que el valor no es un candidato para la selección.

  Mejor (¿), Peor (¡) Para los dos valores involucrados, este valor no sería seleccionado si el otro valor es candidato. Mejor y peor permiten la creación de una ordenación parcial entre valores candidatos. Mejor y peor son invertibles entre ellos, si A es mejor que B, entonces B será peor que A.

  El mejor (¿) esta preferencia dice que el valor puede ser mejor que otro valor de la competencia, a no ser que los otros valores también sean "el mejor". Si un valor es el mejor y no es rechazable ni prohibido ni peor que otro, será seleccionado si no hay otro que sea el mejor o requerido. Si ambos valores son los mejores, se analizarán otras preferencias remanentes como indiferencia, peor... para determinar la selección.

  El peor (¡) Una preferencia como ésta sólo sería seleccionada si no hay otras alternativas.

Indiferente (=) No importa qué valor es seleccionado. Cuando se utiliza dicha preferencia, como no importa qué valor sea seleccionado, por defecto, **SOAR** elige aleatoriamente entre todas las alternativas.

Requerida (!) Una preferencia requerida dice que el valor debe ser seleccionado si se quiere alcanzar el objetivo.

Prohibida ( ) el valor no puede ser seleccionado si se quiere alcanzar el objetivo.

Si hay una preferencia aceptable para un valor de un operador y no hay otros valores compitiendo, dicho operador será seleccionado. Si hay múltiples preferencias aceptables para el mismo estado pero con diferentes valores, la preferencia debe avaluar para determinar qué candidato será seleccionado.

Si las preferencias pueden ser evaluadas sin conflicto, el apropiado operador de aumento del estado será añadido a la memoria de trabajo. Ésto puede pasar cuando todos ellos sugieren el mismo operador o cuando un operadros es preferible a los otros que han sido sugeridos. Cuando hay un conflicto de preferencias, **SOAR** alcanza un *impasse*, como se mencionó anteriormente.

Las preferencias pueden ser confusas; por ejemplo, pueden ser sugeridos valores que ambos sean los mejores, por tanto se alcanzaría un *impasse* a no ser que preferencias adicionales resolvieran el conflicto; o puede haber preferencias como: A es mejor que B y otra que diga que B es mejor que A.

2.4.11. El ciclo de ejecución de SOAR: sin subestados

La ejecución de un programa **SOAR** se lleva a cabo a lo largo de un cierto número de ciclos. Cada ciclo tiene cinco fases:

1. Entrada: Nuevos datos sensoriales llegan a la memoria de trabajo.
2. Propuesta: Las producciones se disparan (*fire*), y se retraen para interpretar los nuevos datos, proponen operadores para la situación actual (propuesta de operadores) y comparan los operadores propuestos (comparación de operadores). Todas las acciones de estas productions son *I-supported*. Todas las producciones equivalentes (*matched*) se disparan en paralelo (y todas las retractaciones ocurren en paralelo), y

la búsqueda de equivalencias (*match*) y los disparos continuan hasta que no quedan más equivalencias completas o retractaciones de productions.

3. Decisión: Se selecciona un nuevo operador, o bien se detecta un *impasse* y se crea un nuevo estado.

4. Aplicación: Las productions se disparan para aplicar el operador (aplicación del operador). Las acciones de estas production serán *O-supported*. Debido a los cambios producidos, otras production con acciones I-supported podrán buscar equivalencias o retraerse. Al igual que durante la fase de propuesta. Más producciones se disparan y retraen en paralelo hasta que no quedan más equivalencias completas o retractaciones.

5. Salida: Los comandos de salida se envían al entorno externo.

Los ciclos continuan hasta que el programa **SOAR** invoca la acción *halt* (como resultado de una production) o hasta que **SOAR** es interrumpido por el usuario.

Durante la evolución de estas fases, es posible que las preferencias que produjeron la selección del operador actual cambien. Siempre que las preferencias de operador cambien, las preferencias se reevalúan y si se debe hacer una selección de operador distinta, el enfoque del estado del operador actual (*operator augmentation of the state*) se elimina inmediatamente. No obstante, un operador nuevo no se selecciona hasta la siguiente fase de decisión, cuando todo el conocimiento ha podido ser obtenido.

2.4.12. Impasses y subestados

Cuando el proceso de decisión se aplica para evaluar las preferencias y determinar el enfoque del estado del operador, es posible que las preferencias sean incompletas a inconsistentes. Las preferencias pueden ser incompletas porque no hay sugerido ningún operador aceptable, o porque no hay suficientes preferencias para distinguir entre operadores aceptable. Las preferencias pueden ser inconsistentes si, por ejemplo, el operador A se prefiere al operador B, y el operador B se prefiere al operador A. Puesto que las preferencias se generan de forma independiente a partir de diferentes instancias de production, no hay garantía de que sean consistentes.

- Tipos de impasse, hay cuatro tipos:

  - Impasse de empate (*tie impasse*) : Un *impasse* de empate se produce si las prefencias no distinguen entre dos o más operadores con preferencias aceptable. Si dos operadores tienen las preferencias best o worst, causarán un empate a no ser que haya preferencias adicionales que permitan distuinguir entre ambos.
  - Impasse de conflicto (*conflict impasse*): Un impasse de conflicto se produce al menos dos valores tienen su preferencias comparativas en conflicto (como A es mejor que B y B es mejor que A) para un mismo operador, y ninguna es rechazada, prohibida, o definida como *requerido*.
  - Impasse de falta de cambio (*no-change impasse*): Un *impasse* de falta de cambio se produce si no se selecciona un nuevo operador durante el proceso de decisión. Hay dos tipos:
    - *Impasse* de falta de cambio de estado: ocurre cuando no hay preferencias acceptable (o require) para indicar operadores para el estado actual (o todos los valores aceptable han sido rechazados). El proceso de decisión no puede seleccionar un nuevo operador.
    - *Impasse* de falta de cambio de operador: ocurre cuando un nuevo operador se selecciona para el estado actual pero no hay más equivalencias de productions durante la fase de aplicación, o bien cuando no se selecciona un nuevo operador durante la siguiente fase de decisión.

- Creación de nuevos estados.

  SOAR trata estas inconsistencias creando un nuevo estado en el que el objetivo es resolver el impasse. Así , en el subastado, se seleccionarán y aplicarán operadores en un intento de descubrir cual de los operadores empatados debe ser seleccionado, o se aplicará el operador seleccionado pieza a pieza. El subestado se llama a menudo subobjetivo (*subgoal*) porque existe para resolver el impasse, pero a veces se llama subestado debido a la representación del subobjetivo en **SOAR** como un estado.

  El estado inicial en el subobjetivo contiene una descripción completa de la causa del impasse, incluyendo los operadores entre los que no se pudo decidir (o exponiendo que no había operadores propuestos) y el estado en el que surgió el impasse. Desde el punto de vista del nuevo estado, este último es el superestado. Así, el superestado es parte de

la subestructura de cada estado, representado por la arquitectura de **SOAR** utilizando el atributo superstate. (El estado inicial, creado en el ciclo de decisión 0-ésimo, contiene un atributo superstate con el valor nil. El estado de nivel más alto no tiene superestado.)

El conocimiento para resolver el impasse se puede obtener de cualquier tipo de resolución de problemas, desde la búsqueda al descubrimiento de las implicaciones de distintas decisiones o la petición de consejo a un agente exterior. No hay ninguna restricción a priori en el proceso, salvo que conlleva aplicar operadores a estados.

En el subastado, los operadores se pueden seleccionar y aplicar mientras **SOAR** trata de resolver el subobjetivo. (Los operadores propuestos para resolver el subobjetivo pueden ser similares a los operadores en el superestado, o pueden ser completamente diferentes.) Mientras se realiza la resolución de problema en el subobjetivo, se puede llegar a nuevos impasses, lo que produce nuevos subobjetivos. Por tanto, es posible que **SOAR** tenga una pila de subobjetivos, representados como estados: cada estado tiene un solo superestado (salvo el estado inicial) y cada estado puede tener como mucho un subastado. Los nuevos subobjetivos se consideran añadidos en la parte baja de la pila, por tanto el primer estado es el estado de nivel más alto (*top-level state*).

- El ciclo de **SOAR** con subestados.

  Cuando hay múltiples subastados, el ciclo de **SOAR** se mantiene en esencia, si bien sufre unos pocos cambios menores.

  El primer cambio es que durante el proceso de decisión, **SOAR** detectará impasses y creará nuevos subastados. Por ejemplo, tras la fase de propuesta. La fase de decisión detectará si no se puede realizar una decisión dadas las preferencias actuales. Si se produce un impasse, se crea un nuevo subastado y se añade a la working memory

  El segundo cambio es que en cada fase, **SOAR** se mueve a través de los subestados desde el más antiguo (el de más alto nivel) hacia el más reciente (el de más bajo nivel), completando todo el procesado necesario a ese nivel y para esa fase antes de hacer nada en el siguiente subestado. Cuando dispara productions para las fases de propuesta o de aplicación, **SOAR** procesa el disparo (y retractación) de reglas, comenzando por aquella que hacen equivaler el subestado más antiguo al más reciente. Cuando una production dispara o se retracta, lo cambios se realizan en la working memory y la preferente memory, lo que posiblemente cambiará que productions equivaldrán en niveles más bajos (las

productions que se disparan en un nivel dado lo hacen en paralelo). Las productions que se disparan en niveles más altos pueden resolver impasses y así eliminar estados de más bajo nivel antes de que las productions de ese nivel se disparen. Así, siempre que se alcanza un nivel en la pila de estados, las actividad de todas las productions tiene la garantía de ser consistente con cualquier procesado que se haya llevado a cabo en niveles más altos.

2.4.13. Aprendizaje

Cuando un *impasse* de operador se resuelve, significa que **SOAR** ha conseguido conocimiento que no estaba disponible anteriormente mediante la resolución de problemas. Por tanto, cuando se resuelve un problema **SOAR** tiene la oprtunidad de aprender, resumiendo y generalizando el procesamiento llevado a cabo en el subastado.

El mecanismo de aprendizaje de **SOAR** es denominado troceado (*chunking*): intenta crear una nueva production, llamada trozo (*chunk*). Las condiciones del trozo son los elementos del estado que permitieron la resolución del impasse (mediante la concatenación del disparo de varias production); la acción de la producción es el elemento de la working memory o la preferencia que resolvió el impasse (el resultado del *impasse*). Las condiciones y la acción son parametrizadaos de modo que esta nueva production pueda tener una equivalencia en una situación similar en el futuro y prevenir la aparición de un impasse.

Los trozos son muy similares a las justificaciones (*justifications*) puesto que ambos se forman a través del proceso de retroseguimiento (*back-tracing process*) y crean un resultado en sus acciones. No obstane, hay algunas diferencias importantes:

- Los trozos son productions y se añaden a la memoria de producción. Las justificaciones no aparecen en la production memory.
- 2. Las justificaciones desaparecen en cuanto que elemento de la memoria de trabajo o la preferencia correspondiente es eliminado
- 3. Los trozos contienen variables de modo que pueden buscar equivalencias en la working memory en otras situaciones, las justificaciones son similares a un trozo instanciado.

2.5. Evolución

2.5.1. Antecedentes

- Teoría Lógica (LT) 1955: Primer solucionador de problemas simbólicos de forma eurística.
- GPS (1958): Análisis significado-fin y objetivos recursivos.
- Espacios de problema (1965): Estructura de tareas uniforme.
- Sistemas de producción (1967): Uniforme, incremental, contexto sensitivo de la representación del conocimiento.
- Conocimiento débil (1969): Organización del control general de conocimiento.

2.5.2. Historia de SOAR

- **SOAR** 1 - 1982
  - Desarrolla una arquitectura que soporta muchos métodos.
  - Desarrolla una arquitectura de Inteligencia Artificial que soporta espacios de problema.
  - Usa un sistema paralelo de producción para la memoria permanente.
  - Los logros en esta etapa son: unificación de los espacios de problema y los sistemas de producción, fijación de las bases para el método universal débil (*Weak Method*), representación separada del conocimiento de control y toma de decisiones basadas en la integración dinámica del conocimiento, construcción de sistemas como simples puzzles y tareas de juego, y es implementado en XAPS2 (*Rosenbloom´s eXperimental Activation Production System*).
  - Número de usuarios: 1
  - Las carencias: No contempla los objetivos de segundo nivel, ni el razonamiento meta modelado.
- **SOAR** 2 - 1983

- El sistema puede generar cualquier tipo y todo tipo de objetivos.
- Mecanismo simple para la generación de todo tipo de objetivos.
- Todos los objetivos de segundo nivel se generan automáticamente.
- Las preferencias a la hora de ejecutar una tarea u otra están basadas en el proceso de decisión.
- Los logros de esta etapa son: Aparece la representación simbólica: aceptable, rechazar, mejor, peor..., detección de la incapacidad para tomar decisiones automáticamente: conflictos, ningún cambio en el proceso de decisión... creación de objetivos de segundo nivel para resolver dichos niveles de conflicto y la memoria de trabajo llega a ser una estructura gráfica con origen en los objetivos a conseguir.
- Carencias: No aprendizaje y debe resolver el mismo comflicto muchas veces durante una tarea.

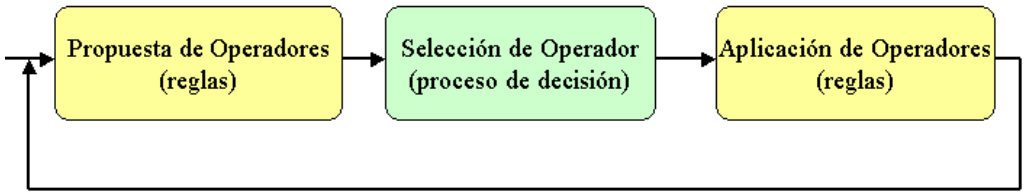

Figura 2.3: Ciclo de trabajo

- **SOAR** 3 - 1984 La pregunta es: ¿cómo puede ser el aprendizaje integrado con la resolución del problema?

  Los logros en esta etapa son: método general de aprendizaje integrado con la resolución del problema:

  - Aprende de forma incremental y continua en todas las tareas.
  - Aprende tanto del éxito como del fracaso.
  - Aprende una gran variedad de tipos de conocimiento
  - Convierte automáticamente el conocimiento profundo a una sombra de conocimiento.
  - Número de usuarios: 8

- **SOAR** 4 - 1986
  - Primera versión para comunicarse con el ambiente exterior.
  - Aumenta el número de aplicaciones desarrolladas y el número de usuarios: más de 50 en 1988.
  - Aumentan los tipos de aprendizaje y métodos de resolución de problemas.
  - Newell propone la Teoría Unificada del Conocimiento. (*Unified Theory of Cognition*)
  - Se crean sistemas basados en conocimientos reales: Diseño de algoritmos, diagnosis médica (Red-SOAR), modelado de procesos químicos, entendimiento del lenguaje natural (NL-SOAR)...
  - Carencias: Interacción limitada con el ambiente exterior y demasiada copia de estructuras de estado durante la aplicación del proceso.
- **SOAR** 5 - 1989
  - Añade mayor interacción con el ambiente exterior.
  - Mayor eficiencia al eliminar muchas de las copias de estructura de estado durante la aplicación del proyecto.
  - Integra interacción, reacción, planeamiento y aprendizaje.
  - Cuatro niveles de arquitectura.
  - Usado en muchos lugares del mundo, alrededor de unos 15 sitios y 100 usuarios.
  - Carencias: Software rústico, el código tiene ya más de diez años y es difícil de mantener y extender, problemas significativos con grandes sistemas y largo tiempo de ejecución.
- **SOAR** 6 - 1992 Mejoras notables en eficiencia, portabilidad y mantenimiento corregido; es ocho o diez veces más rápido que **SOAR** 5 para tareas de dificultad media (100 reglas), de 20 hasta 80 veces más rápido en tareas grandes y de elevado tiempo de ejecución.

  Carencias: No trabaja en tiempo de ejecución con el ambiente y dificultades de integrar **SOAR** con otras aplicaciones.
- **SOAR** 7 - 1996
  - Hace más fácil: Crear herramientas en tiempo de ejecución, conexión con el ambiente exterior y correr muchos agentes simultáneamente.

  - Se integra **SOAR** con Tcl/Tk.
  - Modifica el modelo del espacio de problema computacional
  - Dos compañías usan **SOAR** con fin comercial: ERS (*Explore Reasoning Systems*) y **SOAR** Technology, Inc.
  - Carencias: Inconsistencia en el razonamiento a través posibles objetivos de segundo orden, el aprendizaje puede crear ciertas reglas que nunca se llegarán a ejecutar y las salidas pueden ser inconsistentes con ciertor operadores seleccionados.

- **SOAR** 8 - 1999 Mezcla de problemas con la interacción con el ambiente exteriior. Se desarrolla SGIO: librería C++ para la comunicación con el ambiente exterior.

2.6. Aplicaciones

- 1992
  - Modelo cognitivo, aprendizaje: juego Super Marios (HI-SOAR), resolución de problemas físicos electromagnéticos (EFH-SOAR), Procesamiento del lenguaje natural (NL-SOAR [1]), diagnóstico médico (Red-SOAR), control del tráfico aéreo (ATC).
  - Sistemas de aprendizaje: aprendiendo a través de instrucciones (Instructo-SOAR), aprendiendo de la interacción con el ambiente exterior (IMPROV), aprendizaje simbólico (SCA-2).
  - Sistemas de interpretación: simulador de vuelo (Air-SOAR), primera versión de simulación de agentes militares (TacAir-SOAR [2], RWA-SOAR, Debrief), trabajo en equipo (STEAM), entrenamiento e instrucción (STEVE).
- 1996
  - Aprendizaje multitarea (EPIC-SOAR).
  - Modelado de sistemas de control del tráfico aéreo.
  - Procesos de visión (Vision-SOAR).
  - Más juegos: PACMAN, Quake-SOAR.

[1] Subsistema que permite a los programas aprender e interpretar textos en inglés

[2] Piloto sintético de combate que lleva a cabo misiones aéreas de combate en tiempo real distribuido en ambientes de simulación

- Modelos de organización

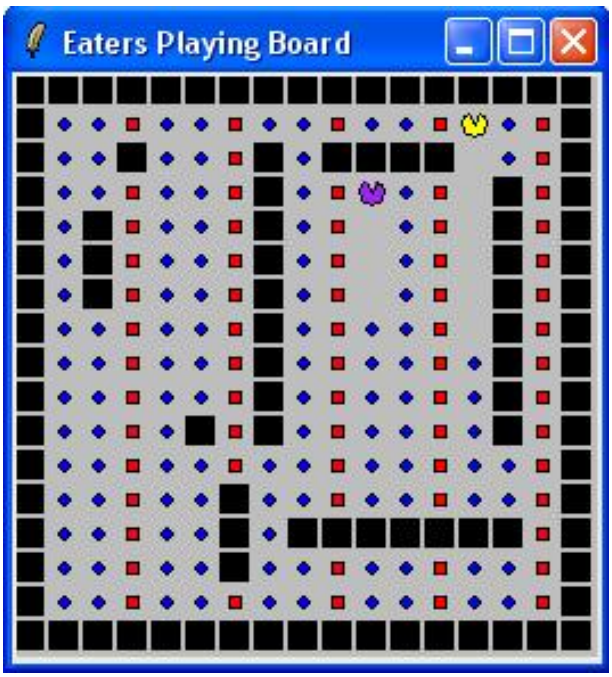

Figura 2.4: Comecocos inteligente-SOAR: *Eaters*

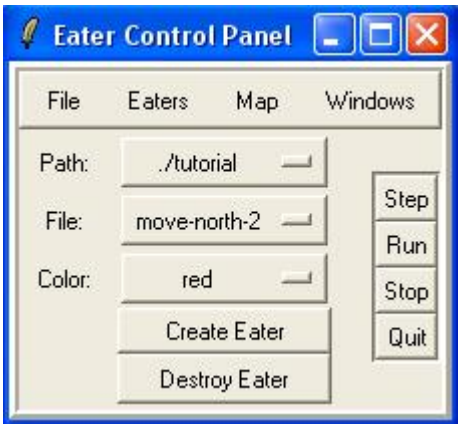

Figura 2.5: Panel de control de *Eaters*

- 1999

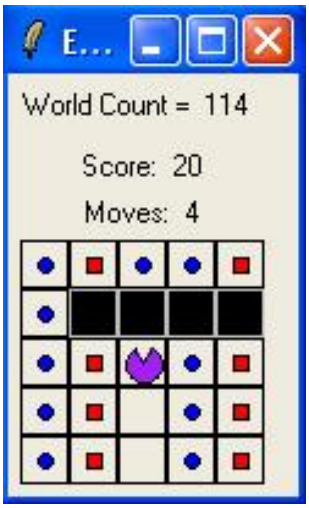

Figura 2.6: Panel de información de *Eaters*

- Interacción con el ambiente exterior: adversarios para simulaciones militares (MOUTBot), control de vehículos aéreos no tripulados (UAV).
- Agentes sociales de interacción.

En las Figuras siguientes: 2.4, 2.5, 2.6 se puede ver la interfaz de usuario de una de las aplicaciones programadas con **SOAR**.

2.7. Instalación

La última versión hasta la fecha de **SOAR** es la 8.5.2, está disponible para plataformas:

- Windows
- Linux
- Mac OSX

A partir de la versión 8.5.0 los desarrolladores de **SOAR** han facilitado la instalación. A partir de dicha versión, en lugar de tener que descargar e

instalar cada uno de los componentes necesarios para el funcionamiento de **SOAR** individualmente, se dispone de un paquete de instalación en el cual se encuentran todos los componentes requeridos.

`http://sitemaker.umich.edu/soar/SOAR_software_downloads`

Una vez descargado el paquete habrá que descomprimirlo e instalarlo en la forma adecuada en la plataforma en cuestión. En el caso de linux sí será necesaria la descarga e instalación de una aplicación adicional: Tcl/Tk.

Capítulo 3

Planteamiento del problema

El principal objetivo del presente proyecto es integrar el software **SOAR** en **ICa**, generando un flujo de datos entre **SOAR** y **Higgs** para un futuro control inteligente de **Higgs** con **SOAR**. Ver Figura 3.1

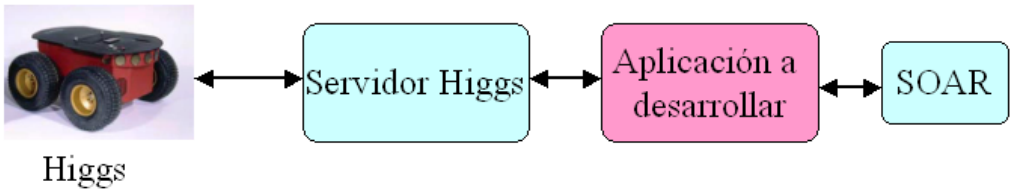

Figura 3.1: Esquema de la aplicación a desarrollar

Tal y como se vio en el Capítulo 2, **SOAR** es una tecnología muy potente y reciente , por tanto muy poco conocida en lugares ajenos a sus desarrolladores. Este factor es uno de los principales obstáculos del presente proyecto fin de carrera.

3.1. Componentes del sistema

3.1.1. Robot Pioneer 2AT-8

En el laboratorio se dispone de un robot Pioneer 2AT-8, **Higgs**, al cual se le quiere dotar de la máxima autonomía.

Higgs es uno de los robots móviles de ActivMedia Robotics [1]. Esta empresa diseña y construye robots móviles así como sistemas de navegación, control y soporte a la percepción sensorial de los mismos. El Pioneer 2AT-8 es una plataforma robusta que incorpora los elementos necesarios para la implementación de un sistema de navegación y control del robot en entornos del mundo real.

Higgs puede establecer una comunicación inalámbrica con cualquier ordenador de la red ASLab, gracias al desarrollo de un proyecto fin de carrera realizado en este mismo departamento por un miembro de ASLab, [Pareja, 2004].

En dicho proyecto se desarrolló un software a partir del cual, en cualquier ordenador de la red ASLab se obtienen los estados sensoriales proporcionados por **Higgs**. La aplicación implementa un servant CORBA que encapsula la interfase proporcionada por el fabricante del robot (Aria). Aria es un paquete de clases con el que se puede comunicar y controlar el robot desde aplicaciones cliente.

En este punto hay que señalar que la solución dada por dicho proyecto va siendo modificada adaptándose a las nuevas necesidades del laboratorio.

3.1.2. SOAR

3.1.3. Problemática

Una vez vistas las dos secciones anteriores se puede ver la estructura general del sistema a solucionar: se tiene un robot, **Higgs**, que es capaz de establecer comunicación con cualquier ordenador de la red ASLab, que se quiere comunicar con un software *inteligente*: **SOAR**. Ver Figura 3.2

3.2. Análisis del problema

El problema principal del proyecto consiste en mapear los estados proporcionados con **Higgs** en un programa **SOAR**, para realizar un control sencillo del comportamiento del robot.

[1] `www.activmedia.com`

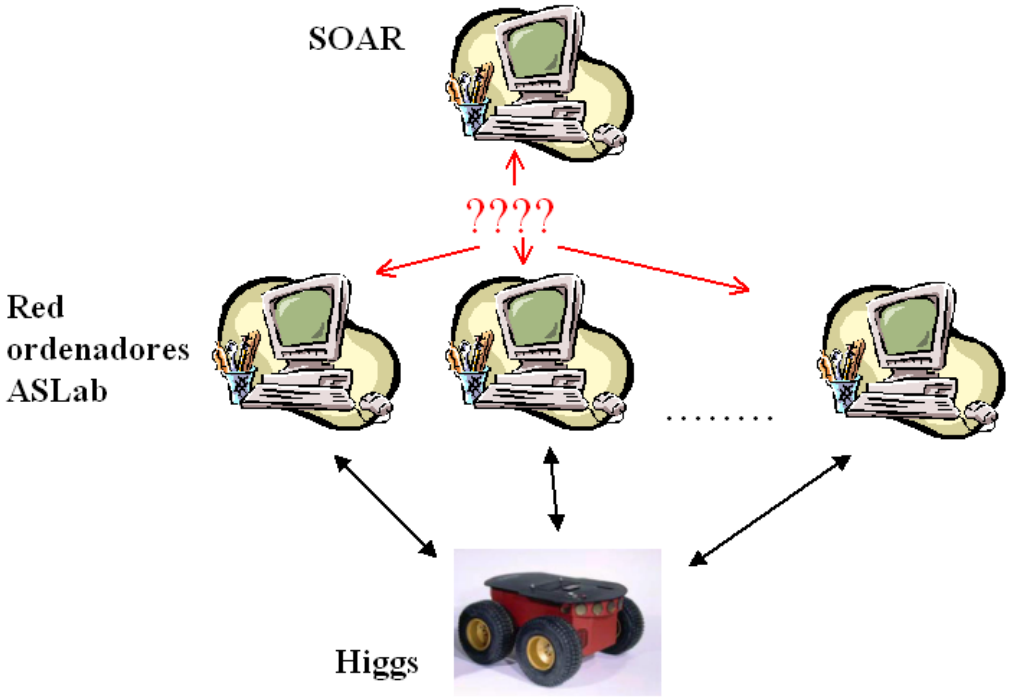

Figura 3.2: Planteamiento del problema

Ya que el servidor **Higgs** es un objeto CORBA, en el presente proyecto se desarrollará un objeto distribuido CORBA que se comunique con el servidor, se tendrá por tanto, de una estructura Cliente/Servidor.

Como se ha visto, el problema no se resuelve simplemente con un objeto que reciba los estados de **Higgs**, sino que dichos estados han de ser procesados por el software a integrar: **SOAR**.

SOAR es un software con el que se pueden programar agentes inteligentes, ésto es, gracias a las aplicaciones **SOAR** es posible resolver problemas de alto nivel, como por ejemplo el famoso problema de las *Torres de Hanoi*.

SOAR no dispone de un entorno directo de comunicación con el mundo exterior, sino una serie de librerías (SGIO), que serán útiles para el desarrollo de la aplicación.

3.3. Conclusiones

Una vez analizado el problema se puede concluir diciendo que el proyecto tiene dos partes iniciales a desarrollar bien diferenciadas, una primera consistente en un objeto CORBA que se comunique con **Higgs**, ver Figura 3.3, y una segunda que aborde el problema de establecer una comunicación con

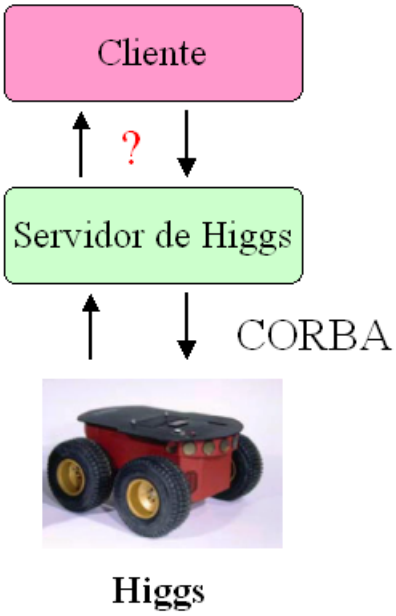

Figura 3.3: Problema 1

SOAR, ver Figura 3.4.

Una vez desarrollados ambos bloques, será necesario integrarlos para que exista una única aplicación que establezca la comunicación entre **Higgs** y **SOAR**: Objetivo del proyecto.

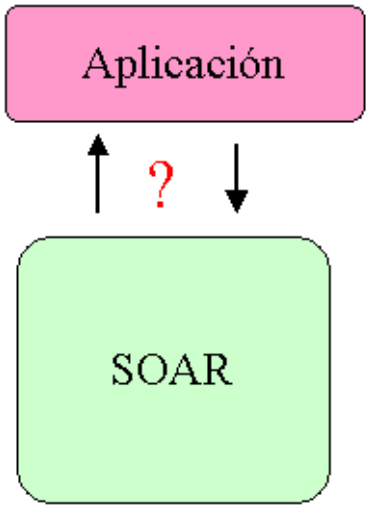

Figura 3.4: Problema 2

Capítulo 4

SGIO

4.1. Introducción

La comunicación de **SOAR** con el ambiente exterior es relativamente reciente. **SOAR** dispone de entradas y salidas, pero para que dichas conexiones puedan interactuar con el ambiente exterior, el grupo de desarrollo de **SOAR** ha escrito una serie de librerías para poder llevar a cabo de una manera más asequible la comunicación de un programa **SOAR** con el ambiente exterior.

En este capítulo se da una idea general acerca de SGIO. Gracias a SGIO ha sido posible programar la comunicación entre **SOAR** y el ambiente exterior: **Higgs**.

4.2. Qué es SGIO

SGIO es una librería de C++ que permite comunicar el ambiente o mundo exterior con **SOAR**. Los objetivos de diseño de este proyecto son:

- Proveer una comunicación eficiente con **SOAR**
- Integrar fácilmente **SOAR** con el mundo exterior
- Ser capaces de embeber **SOAR** con el ambiente
- Hacer posible una comunicación fácil entre distintos modos de comunicación con **SOAR**.

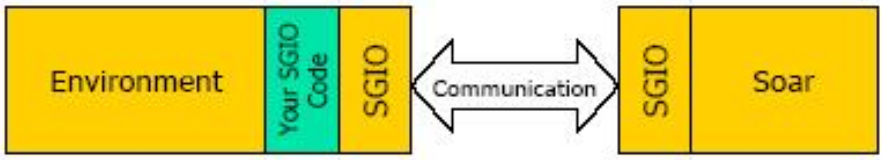

Figura 4.1: SGIO Marco de trabajo

4.2.1. Componentes de SGIO

Simside es una librería que se compila con el sistema externo y es donde se originan las llamadas a **SOAR**.

Shared contiene clases que usan tanto **SOAR**side como Simside, como son los mensajes.

Soarside una herramienta que permite comunicar el ambiente exterior con **SOAR** y tener acceso a él por medio de sockets. Es un ejecutable que crea instancias de **SOAR** y de agentes, manejando y dando respuesta a las peticiones hechas por el sistema exterior a través de Simside. Estas llamadas pueden ser síncronas (la aplicación exterior espera a una respuesta dada por Soarside antes de continuar con la aplicaición) o asíncronas (la aplicación exterior continua lo que está haciendo y atiende a la respuesta de Soarside más tarde). Ver Figura 4.2

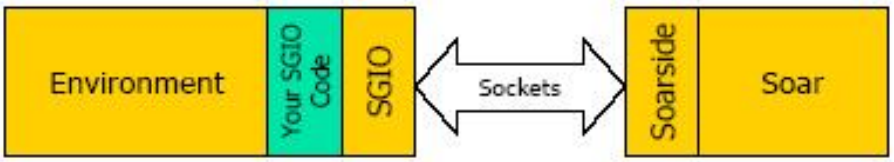

Figura 4.2: Conexión a SOAR con Soarside

Soarside toma dos parámetros opcionales en la línea de comandos: El puerto y un archivo de inicialización (`soarside-init.tcl`).

El número de puerto es el puerto a través del cual Soarside se comunicará, de hecho, Soarside espera que Simside se comunique vía dicho puerto. Si no se dice nada, el puerto por defecto es el 6969.

El archivo de inicialización es un archivo Tcl que especifica la localización de **SOAR**, el directorio donde se encuentran los agentes y arran-

ca **SOAR**. Si no se especifica ningún archivo en concreto, se toma por defecto `soarside-init.tcl`. Si el archivo dado no existe, Soarside lo detecta y aborta inmediatamente.

Se observa que el puerto puede ser especificado de manera individual, mientras que si se quiere especificar un archivo de inicialización determinado, será necesario también especificar el puerto, debido a que Soarside espera los argumentos en un orden particular.

```
--COMIENZO ARCHIVO--

# Especifica la localización de la versión SOAR
que se quiere utilizar
cd ..\\..\\soar-8.4.5\\

# Comienza el código SOAR start-soar.tcl

# Especifica el directorio de Agentes
que se quiere utilizar

cd ..\\sgio-1.0.5\\examples\\simple\\agents\\

--FIN ARCHIVO--
```

4.2.2. Comunicación SGIO

El ambiente (vía SGIO) inicia toda la comunicación con **SOAR**:

- El ambiente inicializa variables, pone datos en el link de entrada (*input-link*) y lee del link de salida (*output-link*).
- **SOAR** no hace llamadas al ambiente; simplemente responde a sus peticiones: El ambiente tanto escribe como lee de los comandos de entrada/salida de los WMEs, elementos de la memoria de trabajo de **SOAR**.

4.2.3. Clases principales

El diseño de SGIO se centra en tres clases principales implementadas en C++:

SOAR La clase **SOAR** representa una conexión particular a **SOAR**. Dicha clase, por sí misma, es solo una interfaz de procedimiento que todas las clases derivadas de **SOAR** tienen que implementar.

Estas clases derivadas incluyen:

- LogSoar: una interfaz básica registrada, útil para depurar el mundo exterior independiente de **SOAR**
- SIOSoar: una interfaz que comunica con un proceso remoto **SOAR**
- APISoar: una interfaz que utiliza el núcleo API de **SOAR** para una ejecución mejor en una única máquina

El papel de cada uno de los constructores es inicializar algunos modos de comunicación con **SOAR**; el destructor para cada clase cierra la conexión.

Agent La clase Agent representa un agente particular corriendo en una conexión **SOAR**. Cada vez que se crea un agente, hay que adjudicarle un nombre. Los agentes son creados y destruidos a través de la interfaz **SOAR**.

WorkingMemory La clase WorkingMemory es una clase de utilidad diseñada para ayudar a los usuarios a tener cuidado de las copias asociadas con la *memoria de trabajo* de los elementos. Para el usuario hay cuatro elementos diferentes de memoria: StringElements, IntElements, FloatElements y SoarIds. Cada uno de estos elementos es creado con sus correspondientes llamadas a función:

- `IntElement* CreateIntWME(SoarId* inParentId, const char* inAttribute, int inValue, int inToler = 0);`
- `FloatElement* CreateFloatWME(SoarId* inParentId, const char* inAttribute, double inValue, double inToler = 0.0);`
- `SoarId* CreateIdWME( SoarId* inParentId, const char*inAttribute);`

Todos ellos, pueden ser eliminados de la memoria de trabajo (*Working Memory*) del agente y su correspondiente memoria con:

`void DestroyWME(Element* element).`

Cuando se elimina un SoarId, los *hijos* de este identificador también se eliminan. Es un error eliminar el elemento *padre* y luego el *hijo*.

Los IntElements, FloatElements y StringElements pueden ser actualizados con nuevos valores utilizando:

- void Update(IntElement* element,int newValue);
- void Update(FloatElement* element,double newValue);
- void Update(StringElement* element,const char* newValue);

Para hacer cambios a la *memoria de trabajo*, se llama a la función Commit(), que lo único que hace es hacer una llamada a CommitWMEChanges en la clase agente.

Una vez descrita la esencia de SGIO, el procedimiento natural para programar en SGIO consiste en mapear las entradas y salidas que se quieren utilizar del ambiente, en una estructura interna de **SOAR** de entras y salidas.

Lo que se suele hacer como regla general es que un identificador en **SOAR** se corresponde con un objeto en C++ y los valores en **SOAR** se corresponden con miembros de la clase en C++.

4.3. Otras alternativas

Existen otras dos alternativas para abordar el problema de la comunicación con **SOAR**, son:

- Comunicación vía Tcl.

  Tcl es un lenguaje de comandos, cuyo intérprete recibe el nombre de tclsh, que tiene como una de sus principales características la gran facilidad con la que se pueden implementar funciones en C/C++ que pasan a ser nuevas instrucciones del intérprete. Es decir, se pueden embeber aplicaciones en C/C++ dentro del propio intérprete de Tcl; de esta forma es posible obtener nuevas versiones de Tcl, denominadas extensiones, que no dejan de ser otra cosa que intérpretes que aunan a los comandos originales de Tcl nuevos comandos escritos en C/C++.

- Comunicación directa con el *kernel* de **SOAR**.

  Lo cual supone una gestión directa de las entradas y salidas del sistema.

4.3.1. Conclusiones

Analizando las alternativas mencionadas anteriormente, se vio que:

- Comunicarse vía Tcl : utilizando esta alternativa la comunicación es más lenta, requiere un elevado conocimiento de Tcl y es necesario hacer referencia a objetos Tcl dentro del código C++ (lenguaje utilizado en el presente proyecto) o utilizar alguna librería compilada como SWIG.
- Comunicarse directamente con el *kernel* de **SOAR**: se necesita un enorme mantenimiento, como puede ser el control de todas las etiquetas de tiempo, y el mapeo de las entradas y salidas directamente puede ser un trabajo tedioso.

Por tanto, debido a los inconvenientes mencionados, se utilizó SGIO gracias a que:

- Se evita el trabajo directo con el *kernel* de **SOAR**.
- El mapeo de las entradas y salidas es más sencillo.
- Puede utilizar TSI para la depuración del código.
- Se puede evitar la utilización de Tcl si no hay que depurar el código, más rápido.
- Se puede lanzar en múltiples máquinas.
- Se puede conectar fácilmente entre depuraciones y modos de funciones de alto nivel.

4.4. Visual Soar

Para el desarrollo de los programas **SOAR** ha sido necesario utilizar Visual Soar como entorno de desarrollo. Gracias a esta herramienta las engorrosas tareas de programar en **SOAR** se hacen más amigables y visualizables.

4.4.1. ¿Qué es Visual SOAR?

Visual Soar es uno de los componentes que vienen en el paquete de instalación de **SOAR**.

Visual Soar es un ambiente de desarrolo escrito en Java para facilitar y ayudar a la creación de agentes para ser utilizados en aplicaciones **SOAR**. Al estar escrito en Java, es necesario tener la herramienta *Java Runtime Environment* instalada.

4.4.2. ¿Qué hace Visual Soar?

Visual Soar contiene herramientas que soportan directamente muchas tareas de programación relacionadas con **SOAR**. Por ejemplo, la creación de la jerarquía de un operador, el mantenimiento del código en árbol.... Adicionalmente, Visual Soar, además de suministrar información sobre lo que se espera que esté en la memoria de trabajo (*Working Memory*), Visual Soar puede chequear para cerciorarse de que las producciones (*Productions*) que se programan son consistentes y en algunos casos completa atributos o valores para el usuario.

4.4.3. Componentes clave de Visual Soar

Los tres componentes claves de Visual Soar son:

La ventana de operación muestra la jerarquía de los operadores que se tienen en el sistema; en ella es donde se define la estructura del sistema en cuestión. Soporta directamente objetivos de segundo orden y la jerarquía de los operadores. Aquí es donde se puede manipular directamente la estructura del sistema. Ver Figura 4.3

Las entidades existentes en un sistema son:

- La raíz la base del proyecto, tendrá el mismo nombre que el nombre que se le haya dado al proyecto. Operadores de segundo nivel pueden ser añadidos a la raíz y acceder a la vista que proporciona el mapa de datos directamente desde la raíz.
- Los operadores hay de tres tipos:
  - Alto nivel producen un subestado y tienen suboperadores.

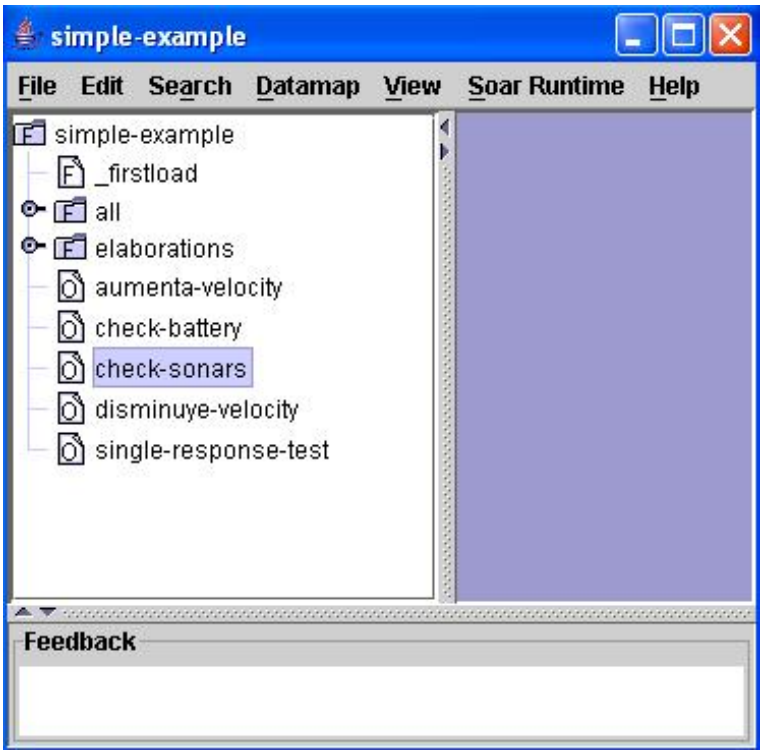

Figura 4.3: Ventana de operación

- Bajo nivel no producen subestados.
- Vinculados son operadores con el mismo nombre que otro de alto nivel en el sistema y producen el mismo subestado.

Cada tipo de operador tiene asociado un archivo de reglas donde las *producciones* asociadas con el operadore se guardan. Los operadores de alto nivel también tienen asociada una vista del mapa de datos.

- Archivos que ayudan a la organización. Los archivos es donde se dejan los datos que debe leer el sistema pero que no son *producciones*; *producciones* que no están asociadas con ningún operador y que se relacionan más con el estado que con los operadores, elaboraciones (*elaborations*) entre otras. El único lugar donde el usuario puede añadir archivos es en la carpeta de elaboraciones (*elaborations*).
- Carpetas son ayudas para la organización. Siempre hay dos:
  - All el lugar para poner operadores que no pertenecen a ningún

lugar en particular de la jerarquía porque pueden ejecutarse en todos los estados.

- Elaborations carpeta que contiene archivos con *producciones* que relacionan el estado superior, se ejecutan en todos los posibles estados y otras *producciones* que no encajan bien en la jerarquía.

El editor de reglas permite editar y manipular las producciones asociadas con un operador o un archivo. Ofrece facilidades básicas para la edición del texto y permite la insercción de plantillas, con la facilidad de autocompletar con el tabulador llamada: "soar complete", la cual completará algunos atributos o valores al usuario. Figura 4.4.

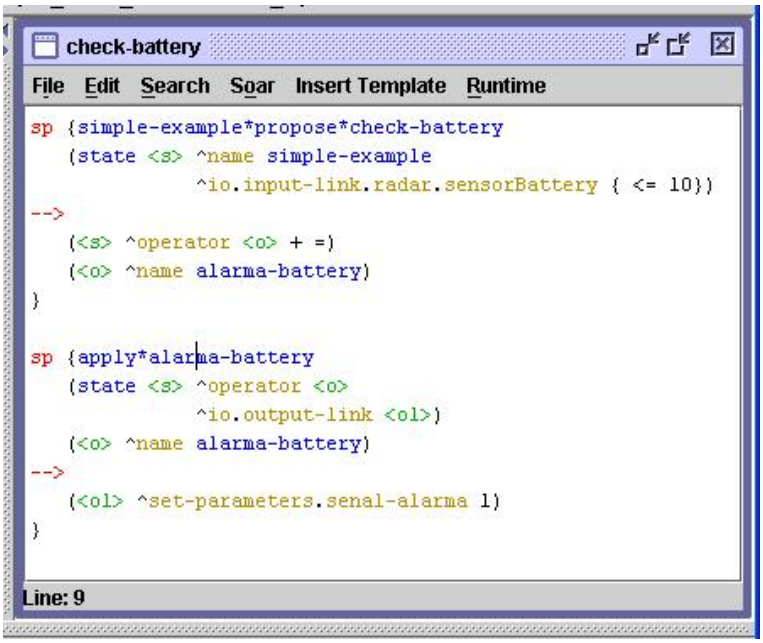

Figura 4.4: Editor de reglas

El mapa de datos muestra todas las posibles estructuras que podrían existir en la memoria de trabajo (*Working Memory*) cuando se cumplen y ejecutan las diferentes reglas. Se supone la representación de un súper conjunto de todos los elementos que pueden estar en la *Working Memory*. Ver Figura 4.5.

Visual Soar utiliza esta información para cerciorarse que las producciones son consistentes. El mapa de datos también utiliza dicha infor-

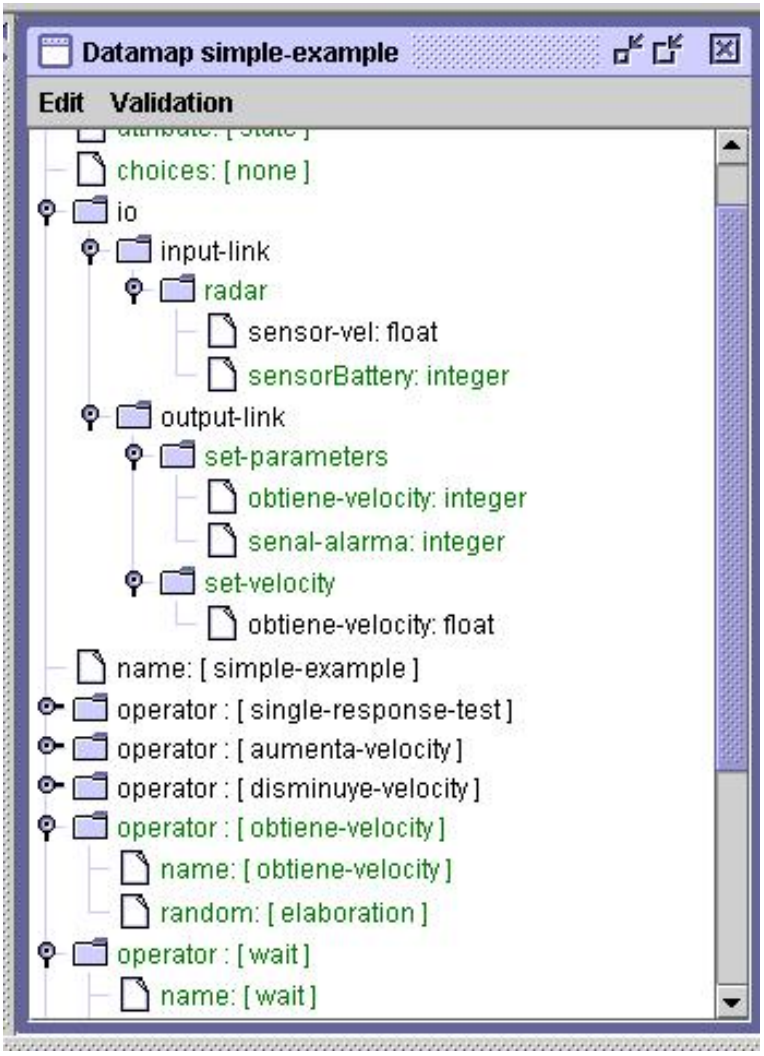

Figura 4.5: Mapa de datos

mación para asegurarse de que los valores tienen el tipo correcto de dato.

Los diferentes tipos de valores que puede crear el mapa de datos son:

- Identificadores: un identificador es equivalente a un identificador **SOAR** y puede tomar un valor.
- Enumeraciones: una enumeración es, generalmente, una enumeración de strings que representa posibles valores para un atributo dado.
- Enteros: un dato entero que puede pertenecer a un rango o no.

- Floats: un dato tipo float que puede pertenecer a un rango o no.
- Strings: un string puede ser básicamente algo además de un identificador.

Nótese que algunos atributos tienen tipos mezclados de valores.

El mapa de datos es una herramienta muy útil, a partir de él se puede estudiar la estructura general de cualquier proyecto **SOAR**, se pueden modificar nombres de identificadores, cambiar el tipo de dato y borrar atributos que no interesen, y todo ello visto desde una perspectiva general, sin tener que navegar por los distintos archivos que contienen dichos atributos.

- ¿Cómo se crea un mapa de datos?

  Un mapa de datos no necesita ser creado explícitamente ya que es generado por Visual Soar cuando un proyecto es creado.

  Para generarlo, hay que desplegar el menú principal: *Datamap-¿Generate de Datamap from de Current Operator Hierarchy.*

  Una vez generado, para abrirlo y poder editarlo, hay que posicionarse con el ratón sobre el nombre del proyecto **SOAR** y con el botón derecho, seleccionar: *Open Datamap.* En la parte derecha aparecerá el mapa de datos.

4.4.4. Observaciones

Gracias a la utilización de dicho entorno de visualización y desarrollo se facilitó la programación de agentes **SOAR**.

Capítulo 5

CORBA

5.1. El Object Management Group

5.1.1. Descripción y objetivos

El *Object Management Group* (OMG) es una organización sin ánimo de lucro creada en 1989, con el objetivo de desarrollar especificaciones de *software* independientes que se basan en arquitecturas orientadas a objetos, y crear un nicho de mercado para estas tecnologías. El OMG pretende con ello reducir la complejidad, costes y esfuerzos que implica la introducción en el mercado y las empresas de nuevas aplicaciones de *software*. Cualquier organización que utilice la tecnología orientada a objetos del OMG, está reduciendo futuros costes de desarrollo e implantación de nuevas aplicaciones como pueden ser los sistemas de control.

El OMG es una organización de estandarización de carácter neutral e internacional, ampliamente reconocida y con un funcionamiento significativamente rápido en lo que se refiere a la creación de nuevos estándares. Hoy en día son miembros del OMG alrededor de mil distribuidores de *software*, desarrolladores y usuarios que trabajan en diferentes campos, incluyendo universidades e instituciones gubernamentales. Además, el OMG mantiene estrechas relaciones con otras organizaciones como ISO, W3C, etc. Los estándares del OMG facilitan la interoperabilidad y portabilidad de aplicaciones orientadas a los objetos distribuidos. El OMG no produce implementaciones de *software*, sólo especificaciones de *software* que son fruto de la recopilación y elaboración de las ideas propuestas por los miembros del OMG a través de las respues-

tas planteadas a los RFI (*Request For Information*) y a los RFP (*Request For Proposals*), documentos mediante los cuales alguno de los miembros del grupo da a conocer su interés para desarrollar una especificación para un campo de aplicación determinado.

5.1.2. Estructura y actividades

Las actividades del OMG se organizan en torno a tres secciones principales:

- *Platform Technology Comitee* (PTC): responsable de la tecnología del núcleo de CORBA.
- *Domain Technology Comitee* (DTC): responsable de las especificaciones para dominios verticales.
- *Architecture Board* (AB): responsable de la arquitectura OMA y la verificación de especificaciones para comprobar que son compatibles con ella.

El trabajo es realizado por grupos de trabajos organizados en diferentes áreas, correspondientes a temas desde el núcleo de CORBA (por ejemplo los protocolos de interoperabilidad) hasta las especificaciones de dominios concretos como pueden ser la adquisición de datos o la seguridad en las operaciones financieras. El proceso de elaboración de especificaciones no es realizado por un comité concreto, sino por un grupo de miembros del OMG, guiados por sus propios criterios y experiencia. Si hay varias propuestas para realizar especificaciones, se trata de llegar a un consenso y elaborar una versión unificada y útil para todos.

5.1.3. Resumen de especificaciones

La siguiente lista muestra las principales áreas donde se realizan especificaciones:

***Common Object Request Broker Architecture* (CORBA).** Especificación que se basa en la interoperabilidad de aplicaciones independiente

de la plataforma, sistema operativo, lenguaje de programación y protocolos de comunicación. Contiene una serie de especificaciones particulares, incluyendo el lenguaje IDL, protocolos de red como GIOP e IIOP, infraestructura para el desarrollo de servidores escalables y portables (POA), y el modelo de componentes de CORBA (CCM).

***Object Management Architecture* (OMA).** Define una serie de interfaces estándar, mediante el uso de IDL, para objetos que soportan aplicaciones CORBA. Incluye los servicios proporcionados por CORBA (CORBAServices, CORBAFacilities y DomainFacilities).

***Unified Modelling Language* (UML).** Estandariza la representación de objetos y el análisis y diseño de sistemas basados en ellos. Es un lenguaje gráfico que incluye diagramas de casos de uso y actividades, diagramas de clases y objetos, diagramas de despliegue, etc.

***MetaObject Facility* (MOF).** Estandariza un metamodelo utilizado para análisis y diseño de objetos.

***Common Warehouse Metamodel* (CWM).** Estandariza una base para el modelado de datos dentro de una empresa, para su uso en bases y almacenes de datos.

***XML Metadata Interchange* (XMI).** Permite el intercambio de datos a través de XML, de modelos basados en MOF y CWM.

***Model Driven Architecture* (MDA).** Unifica los espacios de modelado y desarrollo de aplicaciones, abarcando todo su ciclo de vida, desde el análisis y diseño, pasando por la implementación, hasta su despliegue, mantenimiento y evolución.

Para el modelado de sistemas de control, que es lo que nos atañe, son de especial utilidad las especificaciones de CORBA, UML, MOF y MDA.

5.2. Especificaciones del OMG

Las especificaciones del OMG van más allá de la especificación de CORBA, pero casi todas ellas están relacionadas de alguna manera con dicha especificación o con el modelo de objetos OMA. Algunas de estas especificaciones

están incluidas dentro del documento de especificación de CORBA, "Common Object Request Broker Architecture and Specification", que puede encontrarse en la página web del OMG. El OMG proporciona extensiones y perfiles adicionales en forma de documentos separados. Son de especial importancia para el control las especificaciones de CORBA "mínimo", CORBA de tiempo real, y CORBA con tolerancia a fallos.Las especificaciones pueden agruparse en las siguientes categorías:

- Modelado
- CORBA/IIOP
- Lenguaje IDL y mapeo del mismo
- Especificaciones especiales
- CORBA embebido
- Servicios CORBA
- Facilidades CORBA
- Especificaciones de dominio
- Seguridad

5.2.1. Object Management Architecture (OMA)

La arquitectura para gestión de objetos (OMA) es una de las mayores contribuciones del OMG. Está orientada a la construcción de sistemas distribuidos que utilizan un *middleware* y una serie de servicios comunes predefinidos. Esta arquitectura consiste en cuatro componentes principales divididos en dos partes: los componentes orientados al sistema, y los componentes orientados a la aplicación.

***Object Request Broker* (ORB):** es el núcleo de la arquitectura y permite la integración en tiempo de ejecución de los objetos CORBA que componen la aplicación. Se encarga de gestionar los mensajes e invocaciones que se envían entre dichos objetos.

***Common Object Services (CORBAServices)*:** proporciona una serie de servicios comunes y básicos, cercanos al nivel de sistema, como por ejemplo el servicio de nombres o el servicio de notificación de eventos.

***CORBAFacilities*:** son servicios que no están relacionados con un nicho específico, pero que son de un nivel demasiado alto como para ser incluidos en los servicios anteriores. Por ejemplo, servicios de impresión o de seguridad.

***Domain facilities*:** proporcionan servicios estándar utilizados en campos de aplicación particulares, como puede ser el transporte o la salud.

***Application Objects*:** objetos de la aplicación en desarrollo, que proporcionan servicios concretos para usuarios concretos.

El ORB gestiona toda la comunicación entre los componentes descritos. Permite que interaccionen en ambientes heterogéneos y distribuidos, con independencia de la plataforma en la que cada uno de ellos se ejecuta y de la forma en que fueron desarrollados. Los servicios CORBA están muy relacionados con el funcionamiento del ORB, dada su naturaleza; las "facilidades" y los servicios de dominio son más próximos al nivel de usuario que al de funcionamiento del sistema distribuido.

5.2.2. Especificaciones de CORBA/IIOP

CORBA//IIOP. Esta especificación en concreto describe el *middleware* llamado *Object Request Broker* (ORB) que es la base de la interoperabilidad de CORBA. Es el núcleo de la especificación CORBA.

Interoperabilidad Segura (CSIv2). Indica los requerimientos necesarios para una interoperabilidad segura basada en la autentificación, delegación y privilegios.

Modelo de Componentes CORBA (CCM). Se especifica una extensión del lenguaje IDL llamada CIDL (*Component Implementation Definition Language*); la semántica del modelo de componentes, un marco de integración de componentes (CIF, *Component Integration Framework*), que define el modelo de programación de implementaciones de componentes; un modelo de programación para contenedores de componentes; procedimientos de empaquetamiento y despliegue de componentes, etc.

Tolerancia a fallos. Proporciona un soporte para tolerancia a fallos que sea requerido por aplicaciones críticas.

Actualización en línea. Permite la actualización de los objetos de un sistema de manera portable, a través de sistemas heterogéneos. De esta forma puede cambiarse la implementación de objetos individuales sin que por ello se vea afectada su interfaz externa; se puede detener la ejecución de un objeto para su actualización; retener el estado antes de la actualización; y las capacidades necesarias para la gestión y mantenimiento de objetos sin que sea necesario afectar el funcionamiento global de los sistemas distribuidos.

5.2.3. Lenguaje IDL

El lenguaje IDL debe ser mapeado a diferentes lenguajes de programación para poder implementar los objetos CORBA en dichos lenguajes. Los lenguajes de programación soportados en la actualidad son:

- Ada
- C
- C++
- COBOL
- IDL to Java, Java to IDL
- Lisp
- PL/1
- Python
- Smalltalk
- XML

5.2.4. Especificaciones especiales

Estas especificaciones aportan perfiles y extensiones orientadas a aplicaciones y nichos de tecnología particulares, siendo adecuadas para sistemas embebidos, sistemas de tiempo real, y otros sistemas con funcionalidades especiales.

Proceso de datos en paralelo. Útil para computación que requiera un gran rendimiento, esta especificación define una arquitectura para la programación paralela en CORBA.

Planificación dinámica. Este tipo de planificación se emplea a menudo en sistemas de tiempo real y sistemas distribuidos.

CORBA Mínimo. Una especificación de un subconjunto de CORBA, diseñado para funcionar en sistemas con recursos limitados.

CORBA de tiempo real. Interfaz estándar que satisface los requisitos de tiempo real, ya que proporciona un método para asegurar la predictabilidad de las operaciones en un sistema, además de soporte para la gestión de recursos.

5.2.5. Servicios

Los servicios proporcionan una serie de funcionalidades preconstruidas, útiles para el desarrollo y funcionamiento de las aplicaciones CORBA. La mayoría son útiles en el campo de los sistemas de control:

- Colecciones
- Concurrencia
- Visión Avanzada del Tiempo
- Servicio de eventos
- Externalización
- Servicio de licencias
- Ciclo de vida
- Servicio de nombres
- Servicio de notificaciones
- Persistencia de estado
- Servicio de propiedades
- Servicio de consultas

- Servicio de relaciones
- Seguridad
- Servicio de tiempos
- Intercambio de objetos
- Servicio de transacciones

5.2.6. Facilidades

Son funcionalidades similares a los servicios, pero describen infraestructuras orientadas a usos muy específicos, y no de aplicación general:

- Internacionalización y Tiempo
- Agentes móviles

5.2.7. Especificaciones de dominio

Hay muchas especificaciones de dominio que son más o menos importantes en lo que a sistemas de control se refiere:

- Control de tráfico aéreo
- Flujo de datos audiovisuales
- Consultas bibliográficas
- Análisis de secuencias biomoleculares
- Imágenes clínicas
- Observaciones clínicas
- Sistemas CAD
- Adquisición de datos en entornos industriales (DAIS)
- Simulación distribuida
- Mapas genéticos
- Control de instrumentación de laboratorio (LECIS)
- Identificación de personas

- Gestión de datos de producto
- Gestión de claves públicas
- Acceso a recursos
- Servicios de telecomunicaciones

5.2.8. Especificaciones de seguridad

Otro subconjunto de especificaciones de relativa importancia son aquellas relativas a la seguridad de los sistemas de *software*:

- Capa de autorización de servicios (ATLAS)
- Interoperabilidad segura (CSIv2)
- Servicio de seguridad
- Acceso a recursos
- Gestión de miembros en dominios de seguridad (SDMM)

5.3. La Tecnología CORBA

En esta sección haremos una descripción general de la arquitectura CORBA y su funcionamiento, aunque para conocerlos en profundidad es recomendable estudiar los documentos de especificación del OMG que pueden encontrarse en el *Catálogo de especificaciones CORBA/IIOP*, en el enlace siguiente:

`http://www.omg.org/technology/documents/corba_spec_catalog.htm`

El documento utilizado en este proyecto corresponde a la versión 3.0.2.

5.3.1. Common Object Request Broker Architecture (CORBA)

CORBA especifica un sistema que proporciona interoperabilidad entre sistemas que funcionan en entornos heterogéneos y distribuidos, de forma transparente para el programador. Su diseño se basa en el *modelo de objetos del OMG* donde se definen las características externas que deben poseer

los objetos para que puedan operar de forma independiente de la implementación.

Las características más destacables de CORBA son las siguientes:

- Es una tecnología no propietaria
- Una base fundamental de la especificación son los requisitos reales de la industria
- Es una arquitectura extensible
- Compatible con diversas plataformas
- Independiente del lenguaje de programación
- Proporciona múltiples servicios de aplicación
- Proveedores y usuarios de servicios (servidores y clientes) pueden desarrollarse de manera totalmente independiente.

En una aplicación desarrollada en CORBA, los objetos distribuidos se utilizan de la misma forma en que serían utilizados si fueran objetos locales, esto es, la distribución y heterogeneidad del sistema queda oculta al programador (el proceso de comunicación entre objetos es totalmente transparente).

5.3.2. Arquitectura general

Un objeto CORBA es una entidad que proporciona uno o más servicios a través de una interfaz conocida por los clientes que requieren dichos servicios. No todas las entidades que integran un sistema basado en CORBA son objetos CORBA propiamente dichos, lo son técnicamente cuando proporcionan algún servicio determinado.

La Figura 5.1 muestra los principales componentes de la arquitectura y su interacción cuando se realiza una petición de servicio desde un cliente a un objeto servidor. El componente central de CORBA es el ORB (*Object Request Broker*), el cual proporciona la infraestructura necesaria para identificar y localizar objetos, gestionar las conexiones y transportar los datos a través de la red de comunicación. En general el ORB está formado por la unión de varios componentes, aunque es posible interaccionar con él como si fuera una única entidad gracias a sus interfaces. El núcleo del ORB (*ORB Core*) es la

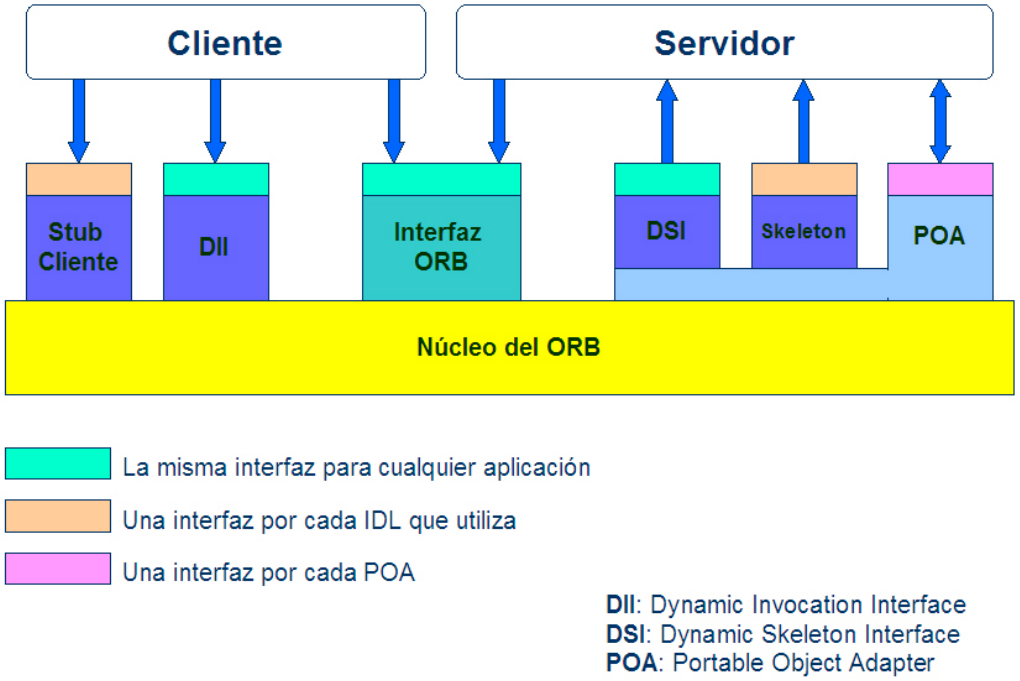

Figura 5.1: Componentes de la arquitectura

parte fundamental de este componente, siendo el responsable de la gestión de las peticiones de servicio. La funcionalidad básica del ORB consiste en transmitir las peticiones desde los clientes hasta las implementaciones de los objetos servidores.

Los clientes realizan *peticiones de servicio* a los objetos, también llamados servidores, a través de interfaces de comunicación bien definidas. Las peticiones de servicio son eventos que transportan información relativa a los objetos o entidades implicadas en dicho servicio, información sobre la operación a realizar, parámetros, etc. En la información enviada se incluye una *referencia de objeto* del objeto proveedor del servicio; esta referencia es un nombre complejo que identifica de forma unívoca al objeto en cuestión dentro del sistema.

Para realizar una petición un cliente se comunica primero con el ORB a través de uno de los dos mecanismos existentes a su disposición: el *stub*, o la interfaz de invocación dinámica (DII, *Dynamic Invocation Interface*). El *stub* es un fragmento de código encargado de mapear o traducir las peticiones del cliente, que está implementado en un lenguaje determinado, y transmitírselas al ORB. Es gracias a los *stubs* que los clientes pueden ser programados en di-

ferentes lenguajes de programación. En la figura 5.2 puede verse un esquema empleando el primer mecanismo, llamado *invocación estática*.

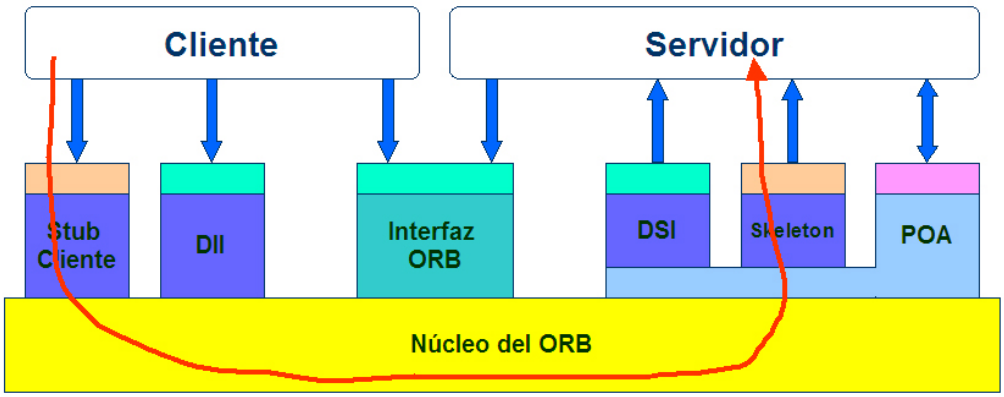

Figura 5.2: Invocación estática

La segunda forma de realizar una petición, el mecanismo de invocación dinámica, se basa en el uso de la DII. Permite realizar invocaciones sin necesidad de que el cliente sepa cierta información acerca del servidor, que sería necesaria en caso de utilizar el *stub*. En la figura 5.3 puede verse el esquema de la invocación dinámica.

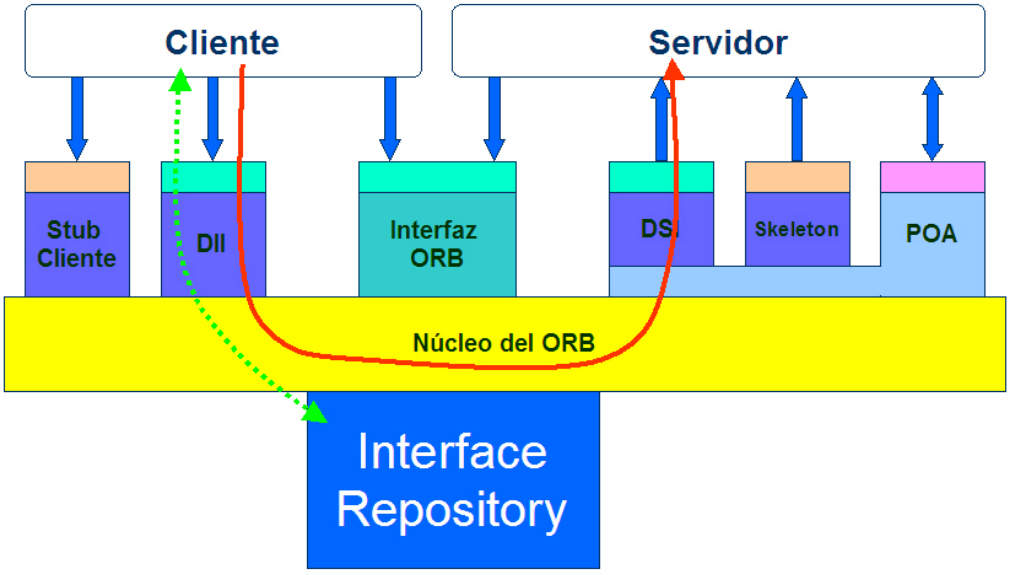

Figura 5.3: Invocación dinámica

Una vez la petición llega al núcleo del ORB es transmitida hasta el lado del servidor, donde se sigue un proceso inverso de nuevo a través de dos mecanismos alternativos: el *skeleton* del servidor, análogo al *stub* del cliente, o la *DSI (Dynamic Skeleton Interface)*, contrapartida de la DII. Los servicios que proporciona un servidor CORBA residen en la implementación del objeto, escrita en un lenguaje de programación determinado. La comunicación entre el ORB y dicha implementación la realiza el adaptador de objetos (*Object Adapter*, OA), el cual proporciona servicios como los listados a continuación:

- Generación e interpretación de referencias a objetos
- Invocación de métodos de las implementaciones
- Mantenimiento de la seguridad en las interacciones
- Activación o desactivación de objetos
- Mapeo de referencias a objetos
- Registro de implementaciones

Existen adaptadores de objetos más complejos, que proporcionan servicios adicionales, y que son utilizados para aplicaciones específicas (por ejemplo, bases de datos). Dos adaptadores de objetos básicos son el BOA (*Basic Object Adapter*) y el POA (*Portable Object Adapter*). El primero ha quedado ya obsoleto, y suele utilizarse el segundo adaptador. El ORB proporciona además un POA preconfigurado que puede usarse si no se desea crear un POA específico para una aplicación, llamado *RootPOA*. El OMG especifica tres políticas de actuación para los adaptadores de objetos:

- **Servidor compartido**: múltiples objetos pueden ser implementados en el mismo programa servidor.
- **Servidor privado**: se ejecuta un nuevo servidor por cada petición de servicio.
- **Servidor persistente**: la implementación del servicio permanece siempre activa.

El adaptador de objetos necesita conocer cierta información acerca de la implementación del objeto y del sistema operativo en el que se ejecuta. Existe

una base de datos que almacena esta información, llamada *Interface Repository* (IR), que es otro componente estándar de la arquitectura CORBA. El IR puede contener otra información relativa a la implementación como por ejemplo datos de depurado, versiones, información administrativa, etc.

Las interfaces de los objetos servidores pueden especificarse de dos maneras: o bien utilizando el lenguaje IDL, o bien almacenando la información necesaria en el IR. La interfaz DII que mencionamos anteriormente accede al IR en busca de ésta información en concreto cuando un cliente la utiliza para realizar una petición. Este mecanismo de invocación ya descrito hace posible que un cliente pueda invocar un servicio sin necesidad de conocer la descripción IDL de la interfaz del objeto servidor.

Para utilizar la DII el cliente debe construir una estructura de datos (común a todas las implementaciones de la especificación) que incluye la referencia al objeto servidor, la operación a realizar y una lista de parámetros. La información acerca de los objetos disponibles y los servicios que proporcionan se obtiene del IR. En el lado del servidor la DSI funciona de manera análoga; si se emplea el mecanismo DII/DSI para invocar un servicio, no se llama al método correspondiente de la implementación a través del *skeleton*, sino que se accede a él a través de una estructura de datos que le identifica y que contiene los parámetros necesarios. De nuevo, en caso necesario, puede obtenerse información relativa al método desde el *Interface Repository*.

La potencia del mecanismo de invocación DII/DSI reside en que no es necesario tener conocimiento de la interfaz de un objeto determinado en tiempo de compilación (éste es el caso del mecanismo *stub/skeleton*), sino que la información sobre dicha interfaz se obtiene en tiempo de ejecución desde el *Interface Repository*. Este mecanismo también puede ser utilizado para conseguir la interoperabilidad entre diferentes implementaciones de ORBs.

Por último, en el lado del servidor CORBA, los objetos que se encargan en última instancia de atender a las peticiones de servicio son los *servants*. Estos objetos contienen la implementación de las operaciones asociadas a cada servicio o método de la interfaz del objeto. No son visibles desde el lado del cliente; éste sólo ve una entidad única a la que conoce como servidor. Dentro del servidor se crean y gestionan *servants* que atienden las peticiones que llegan al mismo. El adaptador de objetos es el encargado de hacer llegar dichas peticiones a los *servants*, crearlos o destruirlos, etc. Cada *servant* lleva asociado un identificador llamado *object ID*, valor que es utilizado por el adaptador de objetos para gestionar los *servants*.

5.3.3. Interoprerabilidad entre ORBs

Existen muchas implementaciones diferentes de ORBs en la actualidad, lo cual es una ventaja ya que cada desarrollador puede emplear aquella que mejor satisface sus necesidades. Pero esto crea también la necesidad de un mecanismo que permita a dos implementaciones diferentes comunicarse entre sí. También es necesario en determinadas situaciones proporcionar una infraestructura que permita a aplicaciones no basadas en CORBA comunicarse con aplicaciones sí basadas en esta arquitectura. Para satisfacer todos estos requisitos existe una especificación para lograr la interoperabilidad entre ORBs.

Las diferencias en la implementación del *broker* no son la única barrera que separa a objetos que funcionan en distintos ORBs, también existen otras dificultades como infraestructuras de seguridad o requisitos específicos de sistemas en vías de desarrollo. Debido a esto, los objetos que funcionan en diferentes dominios (ORBs y entornos de los mismos) necesitan un mecanismo que haga de puente entre ellos. Este mecanismo debe ser suficientemente flexible para que no sea necesaria una cantidad de operaciones de "traducción" inmanejable, ya que la eficiencia es uno de los objetivos de la especificación de CORBA. Esta característica es crítica en sistemas de control, donde suele ser necesario alcanzar unos niveles determinados de seguridad, predecibilidad y seguridad. En la Figura 5.7 puede verse un esquema explicativo de la interoperabilidad. En dicho esquema existen dos ORB diferentes, que bien podrían estar funcionando en dos sistemas de muy distinta naturaleza y localización.

La interoperabilidad puede alcanzarse a través muchos procedimientos, los cuales pueden clasificarse en dos tipos principales: inmediatos e intermediados (*immediate/mediated bridging*).

- En los procedimientos intermediados los elementos de un dominio que interaccionan con los de otro dominio distinto, son transformados a un formato interno acordado por ambos dominios de antemano, y bajo este formato la información pasa de un dominio al otro. Este formato interno de la información puede basarse en una especificación estándar, como en el caso del IIOP del OMG, o bien puede ser un formato acordado de forma privada.
- Los procedimientos inmediatos se basan en traducir directamente la información desde el formato utilizado por un dominio, al formato utilizado por el otro, sin pasar por estructuras intermedias de datos. Esta

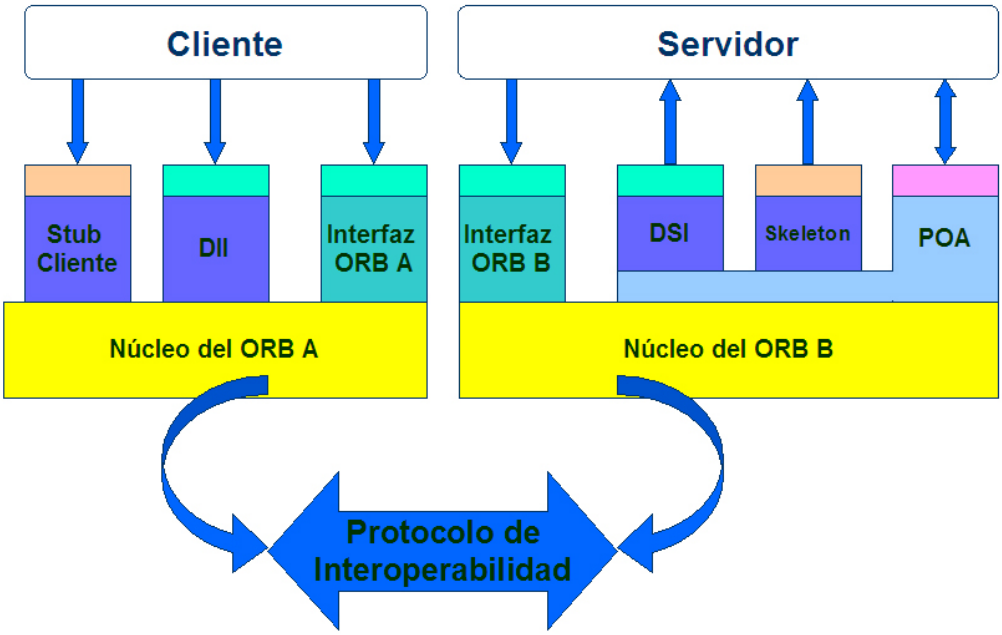

Figura 5.4: Interoperabilidad entre ORBs

solución es mucho más rápida, pero es menos general.

La especificación CORBA define a GIOP (Protocolo Inter-ORB General) como su marco básico de interoperabilidad. GIOP no es un protocolo concreto que se pueda utilizar directamente para la comunicadión entre ORBs. En su lugar, describe cómo se pueden crear protocolos específicos para cumplir con el marco GIOP. IIOP (Inter-ORB de Internet) es una realización concreta de GIOP. La especificación del GIOP consta de los siguientes elementos:

- Suposiciones sobre el transporte: GIOP hace varias suposiciones sobre el nivel del transporte subyacente que se utilice para las implementaciones del protocolo GIOP.
- Representación de datos común (*Common Data Representation, CDR*).

  GIOP define un formato para el envío por la red de cada tipo de datos IDL, de forma que el emisor y el receptor estén de acuerdo con el formato binario de los datos.
- Formatos de los mensajes.

> GIOP define ocho tipos de mensajes que utilizan los clientes y los servidores para comunicarse. Sólo dos de estos mensajes son necesarios para conseguir la semántica básica de las llamadas a procedimientos remotos de CORBA. El resto son mensajes de control o mensajes que ofrecen ciertas optimizaciones.

GIOP especifica la mayor parte de los detalles del protocolo que necesitan los clientes y servidores para comunicarse. GIOP es independiente del nivel de transporte concreto y es, por tanto, un protocolo abstracto, mientras que IIOP es específio de TCP/IP y es, por tanto, una implementación concreta de GIOP. Para convertir GIOP en un protocolo concreto IIOP necesta de tres componentes principales:

- El ID de repositorio
- La información de destino
- La clave del objeto

IIOP sólo especifica la forma en al que se codifica la información de direccionamiento TCP/IP dentro de un IOR, para que el cliente pueda establecer una conexión con el servidor para enviar una petición.

IIOP es el principal protocolo interoperable utilizado por CORBA, y cada ORB que dice ofrecer interoperabilidad debe soportar IIOP.

En el presente proyecto dicha interoperabilidad entre ORB´s se pone de manifiesto.

El servidor que se encarga de comunicar a Higgs con la red de ordenadores del laboratorio está programado en C++, bajo Linux y utiliza como broker el de MICO.

Por otro lado, la comunicación que se tiene que establecer con SOAR, hace que el cliente objeto del presente proyecto se tenga que programar en C++ pero bajo Windows, ya que las librerías del protocolo de comunicación vía SGIO están sólo disponibles para Windows.

Así pues, se dispone de un cliente programado en C++, que trabaja bajo Windows utilizando el broker de omniORB, y un servidor programado en C++, bajo Linux utilizando el broker de MICO; quedando de manifiesto la interoperabilidad entre ORB´s previamente mencionada. Con ésto, no sólo se pone de manifiesto la interoperabilidad entre distintos ORBs si no también la comunicación entre las plataformas de Linux y Windows.

5.3.4. CORBA IDL

Como ya hemos mencionado, el lenguaje IDL (*Interface Definition Language*) es un lenguaje utilizado para especificar interfaces CORBA. A través de un compilador específico que procesa las definiciones escritas en IDL, se genera código fuente en el lenguaje de programación deseado en cada caso, código que es utilizado en las aplicaciones para crear servidores CORBA ya que proporciona la infraestructura de *skeletons* y *stubs* necesaria para que estos objetos puedan comunicarse con el ORB. IDL es un estándar ISO, y tiene las siguientes características principales:

- Su sintaxis es muy similar a la de C++
- Soporta herencia múltiple de interfaces
- Independiente de cualquier lenguaje de programación y/o compilador; puede mapearse a muchos lenguajes de programación.
- Permite independizar el diseño de la interfaz de la implementación del objeto CORBA en cuestión. La implementación puede cambiarse por otra distinta, manteniendo la misma interfaz, de forma que desde el "exterior" el objeto sigue siendo el mismo y continúa ofreciendo idénticos servicios.

IDL no es un lenguaje de programación propiamente dicho, ya que no pueden implementarse con él estructuras de control ni variables. Es únicamente un lenguaje declarativo. Tiene tres elementos principales: operaciones (métodos), interfaces(conjuntos de operaciones) y módulos (conjuntos de interfaces).

Para el lenguaje C++, IDL sigue la siguiente tabla de conversión o *mapping*:

Tipo de datos IDL	*Tipo de datos C++*	*typedef*
`short`	`CORBA::Short`	`short int`
`long`	`CORBA::Long`	`long int`
`unsigned short`	`CORBA::UShort`	`unsigned short`
`unsigned long`	`CORBA::ULong`	`unsigned long`
`float`	`CORBA::Float`	`float`
`double`	`CORBA::Double`	`double`
`char`	`CORBA::Char`	`char`
`boolean`	`CORBA::Boolean`	`unsigned char`
`octet`	`CORBA::Octet`	`unsigned char`
`enum`	`enum`	`enum`

5.3.5. Bases de la construcción de aplicaciones CORBA

Para desarrollar un objeto CORBA se siguen, en general, los siguientes pasos:

1. **Diseño**: determinación de los servicios que debe proporcionar el objeto, e implementación de la/las interfaces IDL correspondientes.
2. **Compilación de IDL**: mediante el compilador de IDL se procesan las definiciones de interfaces y se generan los *skeletons* y *stubs* correspondientes, en un lenguaje de programación determinado.
3. **Implementación de servants**: utilizando las clases generadas por el compilador de IDL, se crean los *servants* de la aplicación, que contienen la implementación de las operaciones del objeto CORBA.
4. **Implementación del servidor**: el objeto CORBA debe proporcionar la infraestructura base para que la comunicación con el ORB sea posible; debe crear un adaptador de objetos para sus *servants*, etc.
5. **Compilación**: se procede a la compilación de los archivos de código fuente. Debe incluirse el código generado automáticamente por el compilador de IDL (la parte correspondiente a los *skeletons*).

El proceso se reduce básicamente a tres fases: generación e integración de código correspondiente a las interfaces, implementación de la funcionalidad del objeto CORBA y por último, compilación. Estos pasos son los correspondientes para el desarrollo de un objeto CORBA, es decir, un servidor. Para

el desarrollo de un cliente, el proceso se reduce a la integración del código fuente generado correspondiente a los *stubs*.

5.4. Servicios CORBA

5.4.1. Servicio de nombres

El Servicio de Nombres del OMG es el más sencillo y básico de los servicios estandarizados de CORBA. Ofrece una correspondencia entre nombres y referencias a objetos: dado un nombre, el servicio devuelve una referencia al objeto almacenado con ese nombre. Similar al Servicio de Nombres de Internet (DNS).

El Servicio de Nombres proporciona varias ventajas:

- Los clientes pueden utilizar nombres con significado para los objetos, en lugar de tener que tratar con referencias como cadenas de caracteres.
- Si cambia el valor de una referencia publicada con un nombre, se pueden obtener clientes que usen una implementación diferente de una interfaz sin tener que cambiar el código fuente. Los clientes usan el mismo nombre, pero obtienen referencias distintas.
- El Servicio de Nombres se puede utilizar para resolver el problema de cómo los componentes de una aplicación acceden a las referencias iniciales para una aplicación. Publicar estas referencias en el Servicio de Nombres elimina la necesidad de almacenarlas como referencias como cadenas de caracteres en archivos.

5.4.2. Servicio de eventos

El Servicio de Eventos del OMG proporciona soporte para comunicaciones desacopladas entre objetos. Hasta ahora, todas las peticiones eran síncronas, un cliente activo invoca operaciones en servidores pasivos; después de enviar una petición, el cliente se bloquea esperando una respuesta. Los clientes son conscientes de los destinos de las peticiones, porque ellos mantienen las referencias a los objetos destino y cada petición tiene un único destino denotado por el objeto usado para llamar.

Por tanto, este Servicio permite a los proveedores enviar mensajes a uno o más consumidores con una única llamada. Los proveedores que usan una implementación del Servicio de Eventos no necesitan ser conscientes de ninguno de los consumidores de sus mensajes: el Servicio de Eventos actúa como un mediador que desacopla los proveedores de los consumidores. Una implementación del Servicio de Eventos también blida a los proveedroes frente a las excepciones que resulten debido a que cualquiera de los objetos consumidores pueda ser inalcanzable o a que su rendimiento sea muy pobre.

Fundamentos del Servicio: En este modelo, los proveedores (*suppliers*) producen eventos y los consumidores (*consumers*) los reciben. Tanto los proveedores como los consumidores se conectan a un canal de eventos (*event channel*). Un canal de eventos conduce los eventos de los proveedores a los consumidores sin que los proveedores tengan que saber nada de los consumidores o viceversa. El canal de eventos juega un papel central en el Servicio de Eventos. Es responsable del registro del proveedor y del consumidor, y de la entrega fiable a tiempo a todos los consumidores registrados y de gestionar los errores asociados con los consumidores que no respondan.

Existen dos modelos para entregar eventos:

- El modelo de inyección (*push*).
- El modelo de extracción (*pull*).

Modelo de Inyección: Los proveedores inyectan eventos al canal de eventos y el canal de eventos inyecta eventos a los consumidores. En la figura 5.5 se puede ver el esquema.

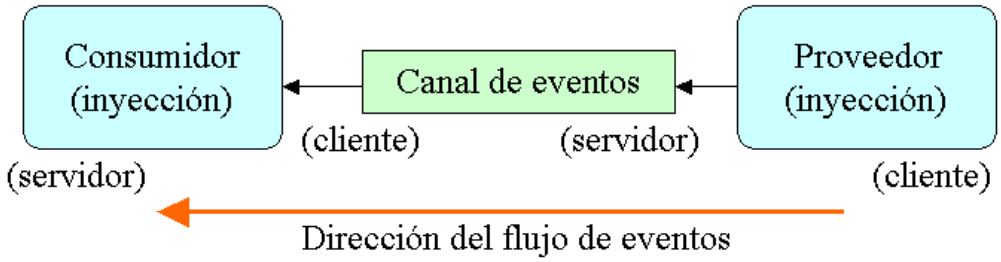

Figura 5.5: Modelo de entrega de eventos por inyección

Modelo de extracción: Las acciones que causan la existencia del flujo de eventos ocurren en la dirección opuesta: los consumidores extraen los eventos del canal de eventos y el canal de eventos extrae los eventos de los proveedores. En la figura 5.6 se puede ver el esquema.

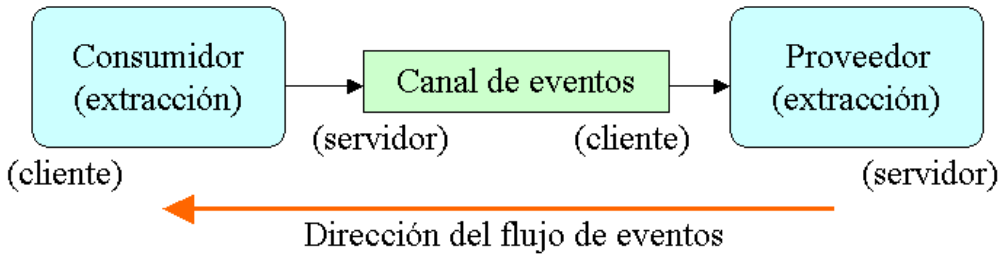

Figura 5.6: Modelo de entrega de eventos por extracción

5.5. UML

El lenguaje unificado de modelado UML (*Unified Modelling Language*, es un lenguaje gráfico y textual que se utiliza para formalizar y diseñar todo tipo de sistemas. Puede usarse tanto para modelos de negocio, como para diseño de *software*, modelos de organización, y todo tipo de aplicaciones.

El UML tiene sus orígenes en una metodología denominada *Object Oriented Analysis and Design*, utilizada en el desarrollo de *software*. Las ventajas de utilizar esta metodología (tanto para el diseño de *software* como para otros dominios) son las siguientes:

- Si hay que realizar algún cambio en un diseño, dicho cambio suele ser muy localizado y se evitan interacciones o modificaciones inesperadas en otras entidades del diseño distintas de la modificada.
- La herencia y el polimorfismo hacen a los programas orientados a objetos más extensibles, permitiendo un desarrollo más rápido.
- El diseño basado en objetos es válido para el desarrollo de sistemas distribuidos, sistemas de funcionamiento en paralelo o sistemas de funcionamiento secuencial.

- Los objetos pueden ser asociados a conceptos del mundo real con mayor facilidad, permitiendo realizar diseños más claros y entendibles por cualquiera.
- Las estructuras de datos compartidas se encapsulan, evitando posibles modificaciones inesperadas u otras anomalías.

Los autores de UML proponen un proceso de desarrollo incremental, basado en casos de uso y centrado en la arquitectura del sistema. Deben respetarse siempre los dos principios fundamentales de la orientación a objetos: encapsulamiento y herencia.

Para el modelado con UML se dispone de nueve tipos de diagramas, que permiten diseñar los diferentes aspectos de una aplicación:

Diagramas de clases. Describen la estructura estática del sistema, y organizan los elementos en grupos (paquetes).

Diagramas de objetos. Describen la estructura estática del sistema para una instancia determinada del mismo.

Diagramas de casos de uso. Modelan las funcionalidades del sistema utilizando actores y casos de uso.

Diagramas de secuencia. Describen las interacciones entre las clases en términos de intercambio de mensajes a lo largo del tiempo.

Diagramas de colaboración. Representan interacciones entre objetos como series ordenadas de mensajes, describiendo tanto la estructura estática del sistema como su comportamiento dinámico.

Diagramas de estado. Describen el comportamiento dinámico del sistema en respuesta a estímulos externos.

Diagramas de actividad. Modelan el flujo de las operaciones del sistema.

Diagramas de componentes. Describen la organización física de los componentes de *software*, incluyendo librerías, ejecutables, etc.

Diagramas de despliegue. Describen los recursos físicos de un sistema, incluyendo nodos, conexiones y componentes.

Una de las mayores ventajas de UML es que permite su extensión con elementos adicionales, en caso necesario. Las extensiones suelen hacerse diseñando nuevos *perfiles de UML*, que proporcionan nuevos elementos específicos de un dominio particular.

5.6. omniORB

5.6.1. Introducción

OmniORB es un *Object Request Broker* (ORB) que implementa la especificación 2.6 de CORBA (*Common Object Request Broker Architecture*). Ha pasado adecuadamente el test para el (*Open Group*) CORBA y fue uno de los tres ORBs al que le fue otorgada la marca CORBA en 1999.

5.6.2. Características

Multihilo: OmniORB es completamente multihilo. Para disminuir la sobrecarga debida a las diferentes llamadas, las innecesarias llamadas multiplexadas han sido eliminadas. Con las políticas por defecto, hay por lo menos una llamada en cada canal de comunicación entre dos espacios de direcciones en cualquier momento. Para hacer ésto sin limitar el nivel de concurrencia, cuando hay demanda, se crean nuevos canales conectando dos espacios de direcciones y alijo cuando hay llamadas concurrentes en proceso.

Cada canal se sirve de un hilo dedicado. Este arreglo proporciona la coincidencia máxima y elimina cualquier hilo que se pone en marcha en cualquiera de los espacios de dirección para tratar una llamada.

En la versión 4.0, omniORB también soporta una política flexible de hilos y puede mandar múltiples llamadas en una sóla conexión. Esta política conduce a un montón de llamadas adicionales, comparadas con el hilo por defecto en el modelo de conexión, pero permite a omniORB escalar al extremo un gran número de clientes concurrentes.

Portabilidad: OmniORB ha sido siempre diseñado para ser portable. Corre sobre muchas variedades de Unix, Windows, varios sistemas operativos embebidos y relativamente oscurecen sistemas como OpenVMS, Jujitsu-Siemens BS2000. Ha sido diseñado para portarlo fácilmente a

nuevas plataformas. El mapeo IDL a C++ para todas las plataformas es el mismo.

OmniORB usa excepciones reales de C++ y clases anidadas. Ésto conserva el estándar de la especificación CORBA mapeando tanto como se pueda y no utiliza mapeos alternativos para dialectos de C++. La única excepción el mapeo de módulos IDL, los cuales pueden usar bien espacios de nombres o clases anidadas.

Parte técnica: Técnicamente hay que destacar que omniORB soporta:

- C++ y Python como lenguajes vinculantes
- Cumple la especificación CORBA 2.6
- Soporte para GIOP e IIOP 1.0, 1.1 y 1.2
- Completamente multihilo en tiempo de ejecución
- *Type Code* y *Type Any*
- Interfaces CORBA 2.6 *DynAny*
- Invocación dinámica e interfaces dinámicas de *skeletons*
- Servicio de nombres (*NameService-¿omniNames*)
- Soporte para *wchar, wstring y code set negotiation*
- Tipos de dato: long long, long double y apoyo de punto fijo
- PortableServer::Current
- Dominio Unix transporte mediante sockets
- GIOP bidireccional
- Transporte de capa interoperable socket seguro
- Dirección de hilo flexible
- Interceptores
- Soporta las siguientes plataformas (algunas de ellas han sido sólo probadas con versiones recientes):
  - Windows NT / XP / 9x with Visual C++ version 5.0 y superiores
  - GNU/Linux / EGCS 2.91 or GCC 2.95 y superiores
  - Solaris 2.5,6,7,8 / Sun C++ version 4.2 and above, or GCC 2.95y superiores
  - HPUX 11.00/ aC++
  - SGI Irix 6.x/ SGI C++ compiler 7.2

- Digital Unix 4.0D/ DEC C++ compiler version 6.0
- IBM AIX 4.2/ IBM C Set++ 3.1.4 and xlC 5.0 (Visual Age C++ 5.0)
- IBM AIX 4.3/ IBM C Set++ 3.6.6 and xlC 5.0 (Visual Age C++ 5.0)
- HPUX 10.20/ aC++ (B3910 A.01.04)
- OpenVMS Alpha 6.2/ DEC C++ compiler 6.2/5.5 (UCX 4.1 ECO 8)
- OpenVMS Vax 6.1/ DEC C++ compiler 5.5 (UCX 4.0 ECO 1)
- NextStep 3.3/ gcc-2.7.2
- Reliant Unix 5.43/CDS++
- Phar Lap's Real Time ETS Kernel
- SCO Unixware 7
- Mac OS X
- Fujitsu Siemens BS2000 (con parche en la distribución)

▪ Completa interoperabilidad con otros CORBA ORBs

Inconvenientes: OmniORB todavía no es una implementación completa del núcleo CORBA 2.6. Las siguientes características no son proporcionadas:

▪ No tiene su propio *Interface Repository* (Repositorio de interfaz).
▪ No soporta objetos por valor (IDL *valuetype*).
▪ No soporta interfaces locales.
▪ OmniORB soporta interceptores, pero no los interceptores portables estandar API.

5.6.3. Instalación de omniORB

En el proyecto fin de carrera la plataforma utilizada es Windows XP, pudiéndose utilizar cualquiera de los siguientes paquetes:

`omniORB-4.0.5-win32-vc6.zip o omniORB-4.0.5.tar.gz`

▪ `omniORB.zip` contiene directamente los binarios. Dichos binarios han sido compilados con Visual C++ 6.0, por tanto, si se utiliza otra versión superior de Visual C++, dichos binarios no funcionarán.

- `omniORB.tar.gz` contine el código fuente, por tanto habrá que construir los binarios.

Una vez descargado uno de los dos paquetes, se descomprime en el sistema, preferiblemente en C: donde una vez extraídos los archivos se generará la siguiente carpeta: C:\omniORB-4.0.5. Dentro de este directorio se encuentran una serie de carpetas: bin, config, contrib, doc, etc, idl, include, lib, man, mk, patches, readmes, src y una serie de archivos.

Para lanzar aplicaciones de omniORB en Windows hay dos alternativas:

1. Utilizar la versión de binarios y utilizar por tanto Visual C++ 6.0.
2. Construir omniORB en Windows compilando los archivos fuente.

A continuación se describen ambas posibilidades:

Para ambos casos será necesario actualizar el path del sistema incluyendo:

```
C:\omniORB-4.0.5\bin\x86\_win32
```

Utilizando la versión de binarios: Configuración de Visual C++ 6.0:
Para trabajar directamente con los binarios será necesario:

- Incluir los archivos generados al compilar la interfaz .idl con el compilador idl, (*omniidl*).
- Actualizar el path de búsqueda de las librerías y archivos a incluir:
  1. En el menú: Tools-¿Options, seleccionar la pestaña *Directories*
  2. En *Show directories for* seleccionar *include files*
  3. añadir el directorio en el que se haya instalado.
  4. En *Show directories for* seleccionar *library files*
  5. Añadir el directorio en el que se haya instalado.
- Actualizar las macros y las librerías:
  1. En el menú *Proyect/settings* seleccionar la pestaña *C/C++*
  2. En el cuadro *Category* seleccionar *C++ Language* y seleccionar *Enable exception handling*
  3. En el cuadro *Category* seleccionar *Code Generation*, en el cuadro *Use run-time library* seleccionar *Multithreaded DLL*

4. En el cuadro *Category* seleccionar *Preprocessor*, en el cuadro *preprocessor* añadir las siguientes macros:
   `__WIN32__,__x86__,_WIN32_WINNT=0x0400,__NT__`
   `y __OSVERSION__=4`
5. Seleccionar la pestaña *link*
6. En el cuadro *category box* seleccionar *input* y añadir las siguientes librerías:
   `ws2_32.lib, mswsock.lib, advapi32.lib,`
   `omniORB405_rt.lib, omniDynamic405_rt.lib,`
   `omnithread30_rt.lib,omniORB405_rtd.lib,`
   `omniDynamic405_rtd.lib y omnithread30_rtd.lib`

Una vez hecha esta configuración, Visual C++ ya está preparado para compilar proyectos de omniORB.

Construcción de omniORB en Windows Se construirá omniORB a partir del código fuente. Los pasos necesarios serán:

- Descargarse Cygwin para poder construir el código fuente. No es necesaria la versión completa, se puede utilizar una más pequeña:

  `\texttt{http://www.uk.research.att.com/pub/`
  `omniORB/gnu\-\win32\-lite\.\zip}`

  `http://www.uk.research.att.com/pub/`
  `omniORB/gnu-win32-lite.zip`

  Una vez que ha sido bajada, se descomprimirá en C:\ y:

  1. En una consola tendremos que tener las variables de entorno activadas, para ello ejecutaremos el comando: VCVARS32.BAT, que se encuentra en el directorio de Microsoft Visual Studio 6.0. Ésto será necesario cada vez que abramos por primera vez un símbolo del sistem.
  2. Añadiremos al path del sistema: C:cygwin32bin (Cuando el path se modifica, será necesario reiniciar el sistema para que los cambios tengan efecto)
  3. Lanzar este script, sólo una vez:
     `C:\gnuwin32\bin\checkmounts C:\gnuwin32\`
     Lo que hace este script es decir a gnuwin32 cómo traducir un path /bin/sh al path real de Windows. Si todo va bien, aparecerá el siguiente mensaje:

```
C:\> C:\gnuwin32\bin\checkmounts C:\gnuwin32 no
/bin/sh.exe,mounting c:\gnuwin32\bin
 as /bin Completed successfully.
```

- Instalar una pequeña versión de Python

  `http://omniorb.sourceforge.net/download.html`

  1. Una vez descargada, se descomprimirá en el directorio raíz de omniORB.
  2. Se editará el archivo config.mk dentro de la carpeta config descomentando la línea:
     `platform = x86_nt_4.0`
  3. Se editará el archivo:
     `<top>\mk\platforms\x86_nt_4.0.mk`
     Y como se está usando la versión light de Python, en dicho archivo se descomentará la siguiente línea:
     `PYTHON = $(ABSTOP)/$(BINDIR)/omnipython`

- Finalmente, desde la consola, dentro de la carpeta /src, ejecutaremos la siguiente sentencia:

  `make export`

Ya estará construido omniORB.

5.7. MICO

MICO implementa el estandard de CORBA. Desde su fundación en Diciembre de 1996, MICO ha llegado a ser muy estable, maduro y una completa y obediente implementación del estandard de CORBA.

MICO está disponible como código abierto bajor el copyright GNU y es ampliamente utilizado para diferentes fines. En Junio de 1999 MICO consiguió la marca de CORBA, lo que significa que ha sido probado y certificado para CORBA 2.1 por el *OpenGroup*.

MICO surgió como un resultado académido en la Universidad de Frankfurt, la implementación MICO tiene un diseño claro y modular. Sólo confía

en C++, el estandar API de Unix y librerías no propietarias, es decir, MICO utiliza una gran variedad de herramientas disponibles sin que éstas sean propiedad.

La versión actual de MICO incluye:

- IDL para el mapeo en C++
- *Dynamic Invocation Interface (DII)*
- *Dynamic Skeleton Interface (DSI)*
- Un buscador gráfico de *Interface Repository* que permite la invocación arbitraria de métodos en interfaces arbitrarias
- *Interface Repository (IR)*
- IIOP como protocolo nativo (Su ORB soporta multiprotocolo)
- *Portable Object Adapter (POA)*
- Objetos por valor
- CORBA componentes (CCM)
- *Dynamic Any*
- CORBA services:
  - Servicio interoperable de nombres
  - Servicio de *Trading*
  - Servicio de eventos
  - Servicio de relaciones
  - Servicio de propiedad
  - Servicio de tiempo
  - Servicio de seguridad

Gracias a esta implementación de CORBA, se reescribió el servidor de **Higgs**, programado previamente utilizando el **ICa** broker, [Pareja, 2004].

5.8. Aplicación al cliente CORBA

El proyecto consiste en desarrollar un objeto CORBA que interaccione con **Higgs**, como se vio en el Capítulo 3 El servidor que gobierna la comunicación directa con **Higgs** y proporciona todos los estados posibles fue desarrollado por un miembro de ASLab [Pareja, 2004]. Dicho proyecto está programado en C++, bajo GNU/Linux, utilizando el **ICa** Broker. En este proyecto se utiliza una versión actualizada del mismo, debido a las actualizaciones que se realizan en el laboratorio adaptadas a las necesidades del momento.

Así mismo, se vio que las librerías para facilitar la comunicación con SOAR están preparadas para Windows, debido a esta limitación, el cliente CORBA a desarrollar tendrá que correr bajo Windows.

Por tanto, en este punto se pone de manifiesto la interoperabilidad entre ORBs, como se vio en la Sección 5.3.3, así como la independencia en de la plataforma en que se desarrollen los objetos CORBA. Ver Figura 5.7.

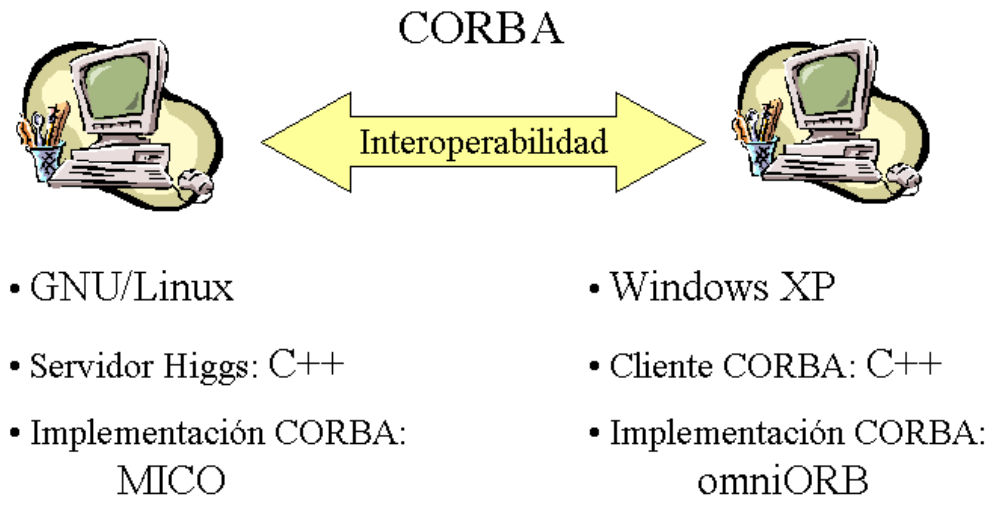

Figura 5.7: Manifestación de la interoperabilidad

5.8.1. Cliente CORBA

Para abordar la comunicación con **Higgs** se desarrolló un cliente prototipo, en este momento se encontraron problemas.

- Problema: a la hora de abordar esta situación hubo algún problema con dicha *teoría* de la interoperabilidad. Se realizaron las siguientes pruebas:

  Pruebas Se lanzó el servidor de nombres de ICA y se intentó que un cliente utilizando MICO u omniORB se conectaran a él. No se consiguió.

  Se lanzaron otros servidores de nombres, el de MICO desde Linux y el de omniORB desde Windows, y se intentó que el servidor en ICA se conectará a ellos. No se consiguió.

  Después de las pruebas fallidas se observó que el servidor de **Higgs** utilizando el **ICa** broker tenía algún problema con la conexión a otro Servidor de Nombres que no fuese el de **ICa**, así como al intentar lanzar el Servidor de Nombres de ICA y que cualquier otro servidor/cliente que no estuviesen programados en **ICa** se conectaran a él.

  Se constató que la arquirectura **ICa** tiene que depurar la parte que se refiere al servicio de nombres.

- Solución: se tomaron varias decisiones:

  - Utilizar el broker de omniORB bajo Windows.
  - Reescribir el servidor de **Higgs** utilizando C++, bajo GNU/Linux pero utilizando el broker de MICO.
  - Lanzar, bien el servidor de nombres de omniORB (`omniNames`) bajo Windows, bien el servidor de nombres de MICO (`./mico.sh`) bajo GNU/Linux.

Una vez adoptadas dichas medidas, la comunicación entre el servidor de **Higgs** (GNU/Linux) y el cliente (Windows) funciona correctamente. Ver Figura 5.8

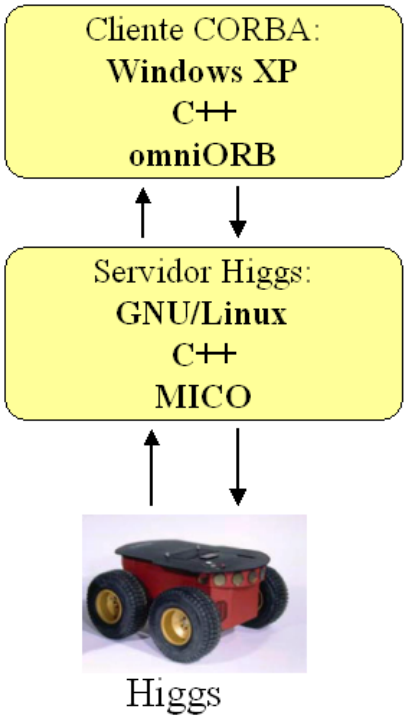

Figura 5.8: Interoperabilidad entre distintos ORBs y distinta plataforma

Capítulo 6

Resolución del problema

6.1. Introducción

El principal objetivo del presente proyecto es la integración de **SOAR** en **ICa**. Una vez que se haya conseguido dicha integración, se tendrá una comunicación activa entre **Higgs** y el software **SOAR**.

SOAR es un software muy potente a partir del cual se pueden programar agentes inteligentes, agentes que son capaces de aprender de la experiencia, de utilizar adecuadamente su conocimiento en la resolución de nuevas situaciones...

El objetivo a largo plazo es controlar el comportamiento autónomo de Higgs gracias a la tecnología **SOAR**, en otras palabras, dotar a **Higgs** de una mente más desarrollada.

Uno de los requisitos previos para alcanzar dicho objetivo es conseguir la comunicación entre Higgs y **SOAR**, de aquí, la importancia del éxito del presente proyecto.

Para la realización del proyecto ha sido necesario el conocimiento y la utilización de distintas herramientas, como son:

- Software **SOAR**: qué es **SOAR**, programación básica de **SOAR**, gestión de sus entradas y salidas, comunicación con el ambiente exterior, entornos propios de desarrollo (VisualSoar).
- Tecnología CORBA: qué es CORBA, utilidades de CORBA, progra-

mación CORBA, diferentes implementaciones de CORBA.

- Documentación de Higgs y su comunicación con la red de ordenadores ASLab [Pareja, 2004].

A continuación se detalla la estructura hardware utilizada para la realización del proyecto.

6.2. Estructura hardware del proyecto

En esta sección se expondrán los componentes hardware de los que consta el presente proyecto fin de carrera.

6.2.1. Higgs: Robot Pioner 2AT-8

El robot Pioneer 2-AT8 pertenece a la familia de robots móviles de ActivMedia. Fue adquirido por ASLab en febrero del 2003. ActivMedia Robotics diseña y construye robots móviles inteligentes así como sistemas de navegación, control y de soporte a la percepción sensorial para los mismos.

Figura 6.1: Pioneer 2-AT8

Higgs (Figura 6.1) tiene un reducido tamaño si lo comparamos con sus prestaciones. Los dos elementos más significativos a tener en cuenta en este proyecto fin de carrera son: los sónares y el microcontrolador.

El cuerpo (Figura 6.3) de aluminio del robot aloja las baterias, los motores, los circuitos electrónicos y el resto de componentes. Además existe espacio para alojar diversos accesorios, como un PC de a bordo, un radio modem o radio ethernet o para incorporar sensores.

El panel (Figura 6.2) está destinado al montaje de nuevos accesorios o elementos para el robot, como podrían ser cámaras, láser o brazos articulados. El panel está dotado con diversos orificios a través de los cuales podrían ubicarse los cables de los posibles dispositivos a añadir. Además a través del panel se puede acceder al interior del robot mediante una ranura de acceso.

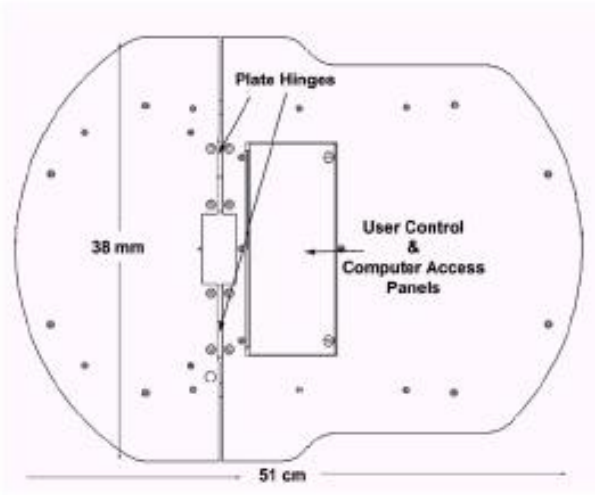

Figura 6.2: Panel superior

Posee dos grupos de 8 transductores (sónares) cada uno. Ver Figura 6.4. Estos dispositivos permiten la detección de objetos y la determinación de la distancia a la que se encuentran. Esta información puede ser utilizada para la elaboración de sistemas de navegación y control. Los sónares están situados en la parte frontal y trasera del robot. Cada grupo de transductores cubre un rango de 180º, de tal modo que el conjunto de sónares cubre los 360º alrededor del robot. Cada grupo de sónares tiene su propia tarjeta electrónica controladora, lo que permite realizar un control independiente.

Cada grupo de sónares está multiplexado. La frecuencia de adquisición de datos es de 25 Hz (40 milisegundos) por sonar. El rango de detección se sitúa entre los 10 cm y los 4 m. Igualmente se puede ajustar la sensibilidad de los sónares y el rango detectado mediante un potenciómetro. Este hecho permite adaptar el robot al ambiente que le rodea. La configuración de los

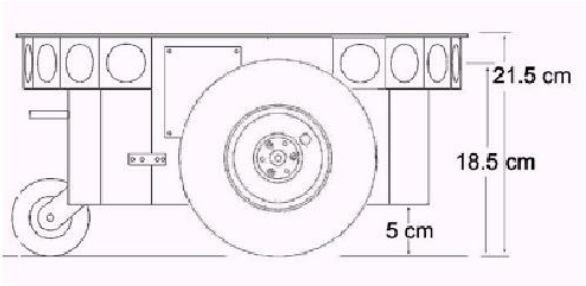

Figura 6.3: cuerpo

sónares con ganancias bajas reduce sus capacidad de detectar pequeños objetos. Este hecho puede ser beneficioso en el caso de que el robot se mueva en ambientes ruidosos o con superficies muy reflectantes. Por el contrario, aumentar la sensibilidad de los sónares estableciendo ganancias altas aumenta las posibilidades de detectar objetos pequeños y objetos que están a gran distancia. Esto es beneficioso si el ambiente en el que opera el robot es abierto y silecioso.

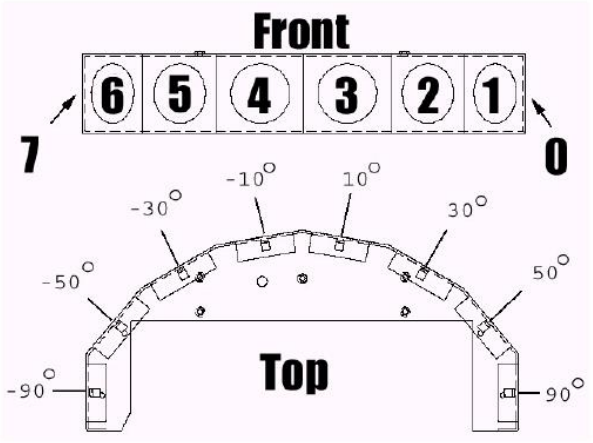

Figura 6.4: Sonars

Tiene un microcontrolador (Hitachi H8S) que gobierna los distintos dispositivos electrónicos conectados, maneja los actuadores, dispara y recoge la señal de los sónares, controla la electrónica del robot y realizar el resto de funciones de bajo nivel. Es capaz de comunicarse con otras máquinas a través de una interfaz serial RS232.

Sistema Operativo AROS: Para el desarrollo de la comunicación de **Higgs** con cualquier otra máquina es preciso utilizar una arquitectura cliente - servidor. Según este modelo, sobre el microcontrolador del robot corren los procesos servidores que se encargan del manejo y control de las tarjetas controladoras de los dispositivos electrónicos y de realizar las funciones de bajo nivel del sistema. AROS (*ActivMedia Robotics Operating System*) es el conjunto de estos procesos servidores y constituye un sistema operativo que corre en el microprocesador.

Es un software de bajo nivel cuyo cometido es manejar la regulación de velocidad de los motores, disparar y recoger la señal de los sonars, recoger las señales de los encoders y en general llevar a cabo todas las funciones de bajo nivel. Además es el responsable de transmitir por medio de comandos esta información a otra aplicación, la cuál será cliente de AROS. El cliente de AROS es un programa que corre sobre la CPU de Higgs y que envia comandos y recibe información del microprocesador Hitachi. Dicho proceso cliente de AROS es a su vez servidor de eventuales programas clientes de la red local. Una de las principales características de AROS.

Dichos servidores se limitan a la gestión de las tarjetas electrónicas y de las funciones de bajo nivel de la plataforma móvil. Sin embargo en el lado del servidor AROS no se pueden llevar a cabo tareas inteligentes o de control. ARIA es el software del lado del cliente.

ARIA Con este paquete de clases se puede comunicar y controlar el robot desde aplicaciones cliente. ARIA está registrado bajo la GNU Public Licence, lo que significa que es un software de código abierto.

6.2.2. Computadoras

En el proyecto entran en juego dos computadoras.

En una de ellas estará instalado el sistema operativo GNU/Linux (Mandrake 9.2) en la cual estará corriendo el servidor que se encarga de leer y modificar los parámetros que lee de los sensores del robot [Pareja, 2004], así como el Servidor de Nombres, necesario para la comunicación entre objetos CORBA.

En la otra máquina, estará instalado el sistema operativo Windows XP en la cual estará corriendo **SOAR**, el servidor **SOAR** y el cliente CORBA/SOAR, objeto principal del proyecto final de carrera.

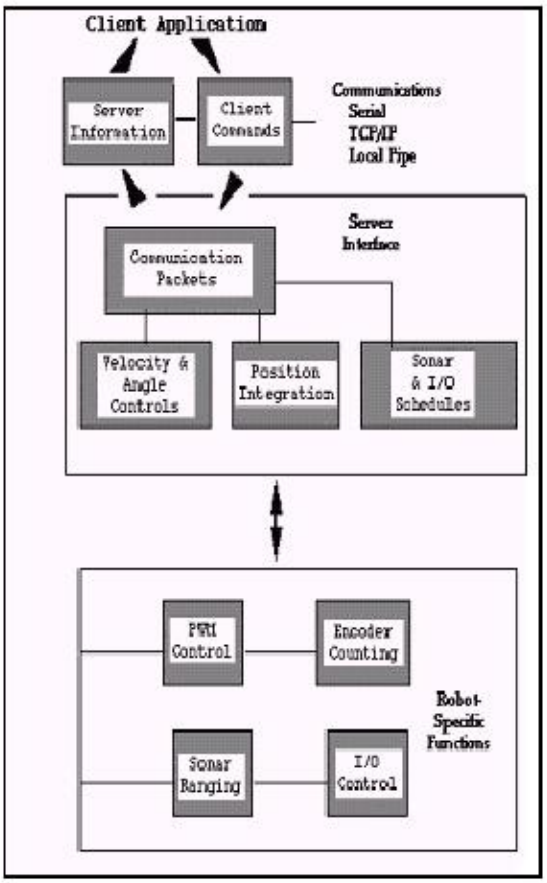

Figura 6.5: Arquitectura cliente-servidor

6.3. Desarrollo de la aplicación

Una vez descrita la estructura hardware del sistema de estudio, las características de la norma CORBA, su especificación omniORB y los componenetes principales de **SOAR**, en esta sección se describen las aplicaciones software desarrolladas para el presente proyecto fin de carrera.

La aplicación se ha desarrollado en cuatro fases principales:

1. Establecimiento de la conexión con Higgs utilizando la tecnología CORBA.
2. Establecimiento de la conexión con **SOAR** utilizando las librerías ofrecidas por SGIO.

3. Ensamblaje de ambas conexiones en una sóla.

4. Verificación del correcto funcionamiento de la solución ensamblada.

6.3.1. Cliente Higgs/CORBA

En esta primera fase se realizó un programa cliente que se comunicaba, gracias a la tecnología CORBA, con el servidor que gobierna los estados de Higgs [Pareja, 2004]. Dicho cliente fue escrito en C++, utilizando el ORB de omniORB bajo Windows. Ver Figura 6.6.

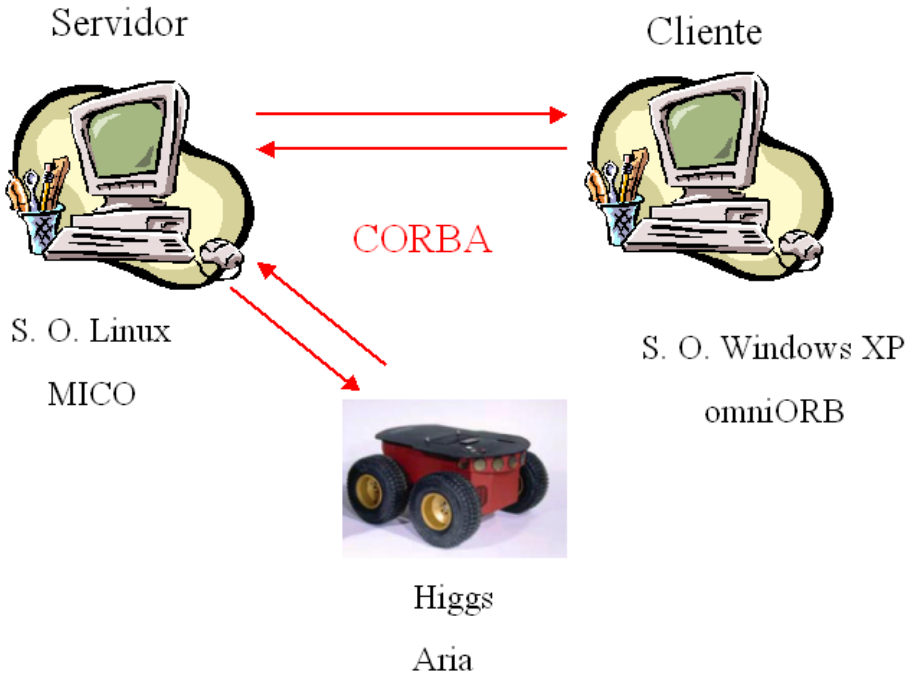

Figura 6.6: Cliente CORBA

Como se mencionó en el capítulo 3, la norma CORBA define un lenguaje de descripción de interfaces, llamado Lenguaje de Definición de Interfaces **IDL** (*Interface Definition Language*) y traducciones de este lenguaje de especificación IDL a lenguajes de implementación (como pueden ser C++, Java, ADA, etc.). Mediante este lenguaje se podrán definir interfases de clases, métodos y tipos de datos que después serán mapeados a un lenguaje concreto.

Por tanto, como la invocación que se utiliza es estática (ver Sección 5.4.2), para la realización del cliente se utilizó la misma interfaz .idl que disponía Higgs (`interfazcorba.idl`). En este fichero se describen los interfaces y métodos de las clases que son utilizados en las aplicaciones cliente y servidor.

Una vez definido el archivo .idl, se compila con el compilador específico de idl (`omniidl`) para obtener los dos archivos necesarios para la generación del código del cliente: `interfazcorba.h` e `interfazcorbaSK.cpp`.

Compilación idl: Para llevar a cabo la compilación idl se abre una consola y se teclea: `omniidl -bcxx interfazcorba.idl` en el directorio en el que se encuentre el archivo .idl.

Como bien se ha dicho, en este fichero .idl sólo se describen los métodos, no su implementación. El cuerpo de los métodos descritos en el fichero idl está desarrollado en el lado del servidor.

De este modo, los métodos pueden ser invocados remotamente por cualquier objeto cliente sin que éste conozca la implementación de los mismos. Así mismo, se puede modificar su implementación de forma transparente al cliente.

En el fichero `interfazcorba.idl` se declara la clase Pioneer2AT. Los métodos de esta clase, en líneas generales, posibilitan la conexión y desconexión con el robot, leen los diversos datos sensoriales del robot (velocidades, rango de los sonar, etc) y establecen en el robot las velocidades lineal y de rotación que le sean pasadas como parámetros.

Una vez obtenidos los archivos generados por el compilador idl, se generó un proyecto Visual C++ 6.0 en el que se escribió el programa cliente que hablaba con Higgs. Como se vio en el capítulo de CORBA, en la Sección 5.6.3, para poder compilar el cliente es necesaria la configuración del Visual C++ 6.0.

6.3.2. Cliente SOAR

En esta segunda fase, se desarrolló un cliente que establecía una comunicación con **SOAR**. Figura 6.7.

Durante su realización hubo que familiarizarse con el entorno de programación de **SOAR**: VisualSoar y con el método de programación específico de **SOAR**.

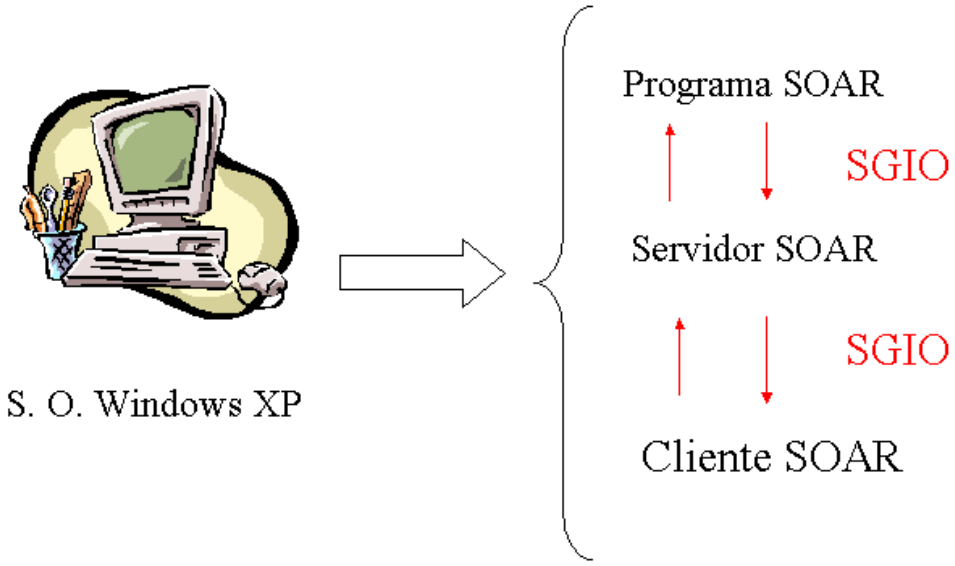

Figura 6.7: Cliente SOAR

Así mismo, se desarrolló una clase `CSgioSys`, basándose en las librerías que proporciona SGIO para establecer la comunicación con **SOAR**.

Gracias a esta clase (`CSgioSys`) se puede iniciar la comunicación con **SOAR**, generar cualquier tipo de estructura interna en **SOAR**, diferentes WME´s (elementos de la memoria de trabajo), correr los programas **SOAR** y obtener los resultados de salida, eliminar agentes creados, terminar la conexión...

Para el desarrollo de este cliente, es necesario dividir el trabajo en dos partes:

1. Desarrollo de un programa **SOAR** que tenga como entradas todos los estados posibles que **Higgs** le pueda dar, y como salidas, aquellos valores que se quieran modificar en **Higgs**. Su estructura interna se puede ver en la Figura 6.8

2. Desarrollo de un programa en Visual C++, utilizando la clase `CSgioSys`, el cual, mapea todos los estados posibles de **Higgs**, tanto parámetros de entrada como los salida objeto del proyecto, para que sean entendidos por **SOAR**.

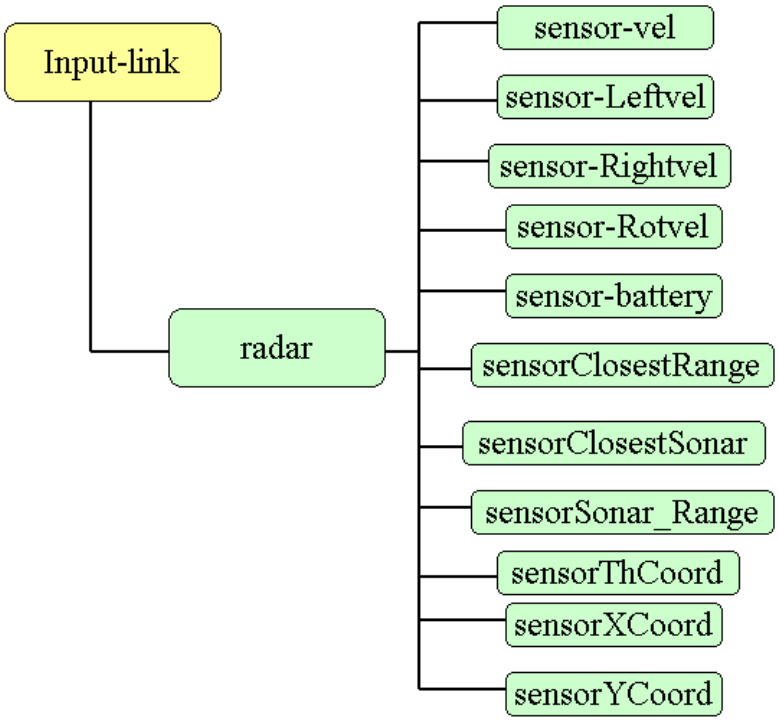

Figura 6.8: Estructura interna de todas las entradas en SOAR

Para el correcto funcionamiento, es necesario editar un archivo:

`soarside-init.tcl`

indicándole la ruta en la que se encuentra el archivo .soar a cargar en el mencionado programa de Visual C++ 6.0.

6.3.3. Ensamblado del cliente CORBA y el cliente SOAR

Posiblemente esta etapa haya sido una de las más complicadas en el desarrollo de todo el proyecto. Se parte de dos programas que funcionan correctamente cuando lo hacen de forma independiente y hay que conseguir un único programa que realice las mismas comunicaciones anteriormente explicadas. Figura 6.9.

El unión fue difícil porque el trabajo no era tan sencillo como copiar el código del cliente **SOAR** y añadirlo al código del cliente CORBA. Fue nece-

sario ir haciendo las cosas paso a paso; hubo que configurar correctamente el entorno de desarrollo Visual C++ 6.0 para que no hubiese ningún tipo de problema de linkado y de conflicto entre todas las librerías que entraban en juego.

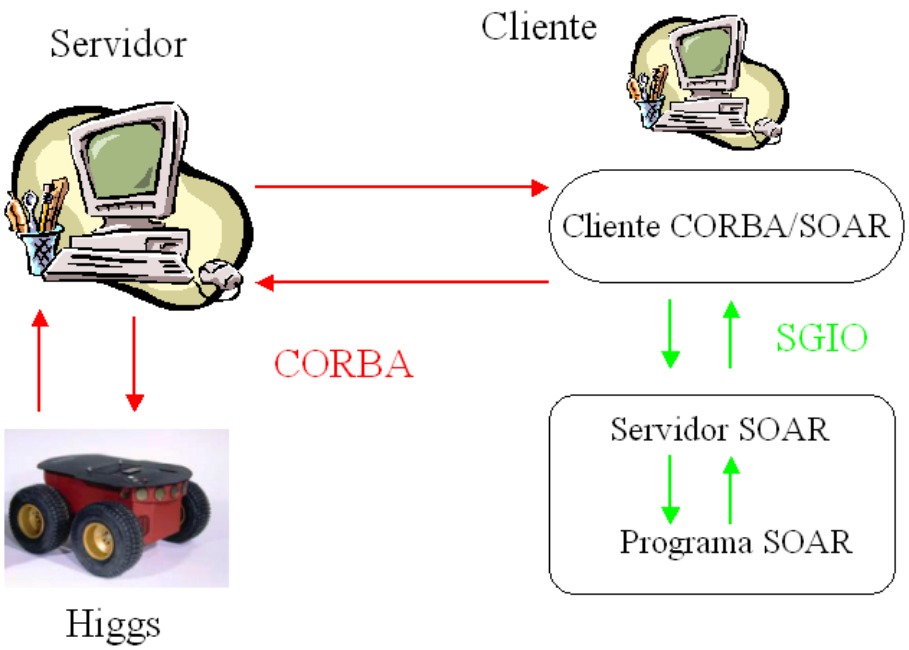

Figura 6.9: Estructura de la solución

6.3.4. Verificación de la solución

Una vez montada toda la solución, se procedió a verificar la validez del programa.

Para ello, con ayuda de un simulador (SRIsim) del robot, ver Figura 6.10, se hicieron las pruebas pertinentes. En dicho simulador se tiene la imagen de un robot al cual se le puede manipular obteniendo los mismos resultados que si se tratase del robot real.

Utilizando dicho simulador, se conocen los valores de todas las variables del robot en cada momento, por tanto, se podía verificar si los datos que le

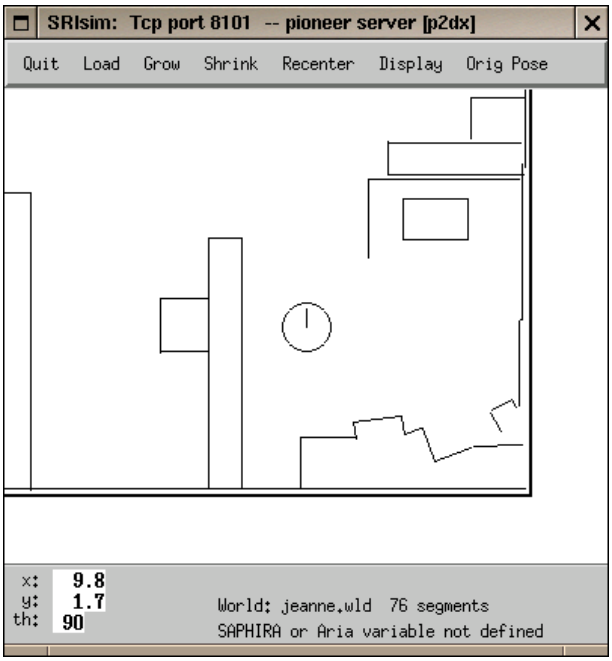

Figura 6.10: Simulador SRIsim

eran pasados al programa **SOAR** eran correctos o no, además, se podían ver los efectos que tenían sobre el robot los cambios que se sugerían a través de **SOAR**.

6.3.5. Mejoras

Una vez conseguida la comunicación y el flujo de datos entre **Higgs** y un programa **SOAR** se pensó en ampliar la funcionalidad del sistema.

La aplicación desarrollada está basada en una estructura cliente/servidor, ésto significa que las invocaciones de operaciones entre uno y otro se realizan de forma síncrona. Con peticiones síncronas, el cliente (activo) invoca operaciones al servidor (pasivo); después de enviar una petición, el cliente se bloquea esperando una respuestas. El cliente es consciente del destino de la petición, porque mantiene la referencia al objeto destino y por tanto, cada petición tiene un único destino denotado por el objeto usado para llamar. Si por cualquier circunstancia el objeto destino, el servidor **Higgs** no existiera o por alguna razón no se pudiera alcanzar, el cliente recibiría una excepción. Por esta razón, se pensó en desacoplar cliente y servidor para atender situa-

ciones de emergencia. El esquema de la aplicación con esta mejora se puede ver en la Figura 6.11

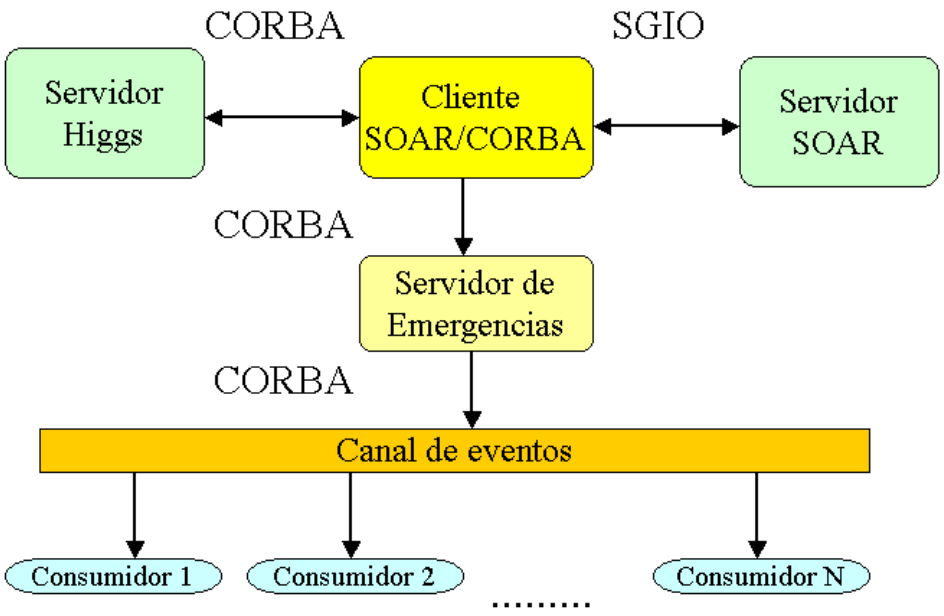

Figura 6.11: Estructura genérica de la aplicación

Observando la Figura 6.11 se observa la aparición de un nuevo elemento: el canal de eventos. Como se vio en el Capítulo 5, Sección 5.4.2 es necesario hacer uso del Servicio de Eventos del OMG, que permite a las aplicaciones usar un modelo de comunciaciones desacopladas en lugar de las invocaciones estrictas de tipo síncrono cliente servidor.

Con esta idea se desarrolló un nuevo servidor: servidor de emergencias, el cual se encarga de recibir las posibles señales de alarma generadas por el cliente SOAR/CORBAy volcarlas en el canal de eventos para que cualquier otro objeto CORBA que esté conectado al canal sea consciente de la emergencia y pueda ejecutar acciones pertinentes.

El modelo elegido para la entrega de eventos ha sido el modelo de inyección (*push*), ver Figura 6.12

Se eligió este modelo de entrega porque se pretende que en cuanto el

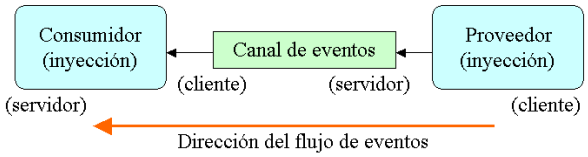

Figura 6.12: Modelo de entrega de eventos por inyección

sistema detecte cualquier tipo de emergencia la comunique lo antes posible al canal de eventos para que los objetos interesados actúen lo más rápido posible. Si se hubiera utilizado el otro modelo, el modelo de extracción (*pull*), ver Figura 6.13,

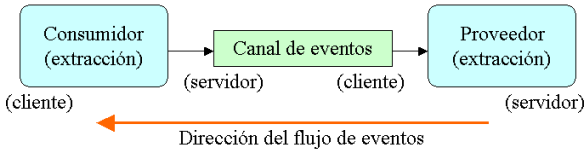

Figura 6.13: Modelo de entrega de eventos por extracción

los objetos interesados sólo se enterarían de la emergencia cuando se lo solicitasen al canal de eventos, pudiendo haber pasado un tiempo razonablemente largo desde el momento en que se produjo la emergencia hasta la solicitud del estado.

6.3.6. Ejecución de la aplicación

El lanzamiento de la aplicación no se limita simplemente a lanzar una archivo ejecutable obtenido de la compilación de un código fuente, es decir, la aplicación en sí, no consiste en un archivo fuente de un cliente y su archivo ejecutable obtenido de la compilación del fuente, si no que entran en juego más elementos como servidores, clientes y servicios; por tanto habrá que realizar una serie de pasos para lanzar toda la aplicación. Ver Figura 6.14

1. Lanzar el Servicio de Nombres.

   Como se ha visto, se utiliza una distribución de CORBA diferente para el servidor y el cliente, el servidor utiliza MICO y el cliente omniORB,

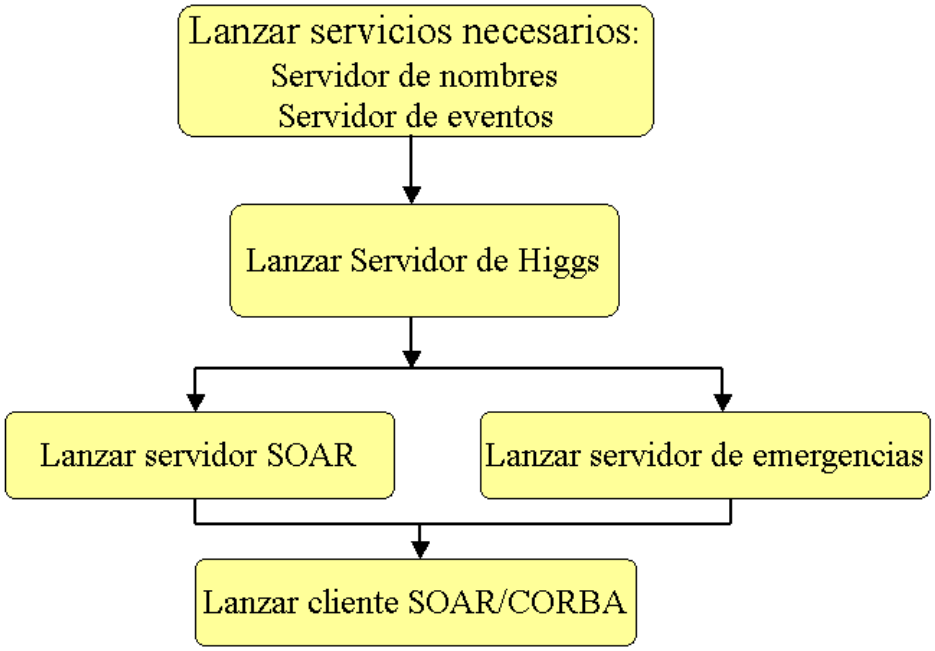

Figura 6.14: Secuencia de operaciones para el lanzamiento de la aplicación

por tanto, se tienen dos posibilidades, ambas perfectamente válidas: lanzar el servicio de nombre de MICO o el servicio de nombres de omniORB. En el proyecto se lanza el servidor de nombres de MICO, pero a continuación se explica cómo lanzar ambos servidores de nombres:

a) Lanzar servidor MICO: en una consola de GNU/Linux hay que situarse en el directorio donde se encuentre MICO y habilitar las variables del sistema haciendo la llamada: `. lib/micosetup.sh` y a continuación lanzar el servidor de nombres: `./mico.sh`.

b) Lanzar servidor de omniORB: lo primero de todo es crear una carpeta con el nombre OMNINAMES que omniORB utilizará internamente. La primera vez que se lanza el servicio de nombres es necesario habilitar dicha carpeta utilizando el siguiente comando:

```
set OMNINAMES_LOGDIR= [Directorio donde se haya
   creado la carpeta OMNINAMES]
```

a continuación, situándose en el directorio:

```
bin\x86_win32
```

se tecleará el siguiente comando: `omniNames -start`.

Para lanzar el servidor de nombres en ocasiones posteriores, simplemente habrá que situarse en el directorio `bin/x86_win32` y teclear el comando: `omniNames -logdir` [Directorio donde se encuentre la carpeta OMNINAMES].

2. Lanzar el servidor que proporciona los estados de Higgs.

   En una consola de Linux, siendo root, habrá que situarse en la carpeta en la que se encuentre el lanzador del servidor y lanzarlo, tecleando su nombre precedido por ./

3. Lanzar el Servidor de Eventos: se ejecutará

   ```
   omniEvents -l C:/omniEvents
   ```

4. Crear un canal de eventos: se lanzará el archivo `eventc`.

5. Lanzar el servidor del lado **SOAR**.

   Para lanzar el servidor **SOAR**, habrá que ejecutar el archivo:

   ```
   start-soarside
   ```

6. Lanzar el cliente.

   Una vez lanzados todos los servicios necesarios y el servidor, para lanzar el cliente, sólo habrá que ejecutar el archivo ejecutable generado por el proyecto de Visual C++ al compilar.

Capítulo 7

Conclusiones y líneas futuras

7.1. Conclusiones

En este proyecto fin de carrera se ha desarrollado una aplicación cliente **SOAR-CORBA** que es capaz de comunicar un programa **SOAR** con **Higgs**, un robot autónomo Pioneer 2AT-8.

Gracias a esta comunicación, el comportamiento de **Higgs** puede ser gobernado a través de un software inteligente: **SOAR**. Con la integración de **SOAR** en la arquitectura será posible dotar a **Higgs** de mayor autonomía, de conocimiento: tanto de sí mismo, como del mundo exterior, de medios adicionales de razonamiento, de capacidad para aprender: bien por su propia experiencia, bien a través de entrenamientos, en definitiva, **Higgs** dispondrá un de cerebro más completo.

Adicionalmente se ha desarrollado un servidor de emergencias. Gracias a este servidor, los posibles estados de alerta detectados a través de **SOAR**, serán conocidos por cualquier otro objeto del sistema que esté involucrado en el correcto funcionamiento de **Higgs**.

7.2. Líneas futuras

El presente proyecto ha basado su esfuerzo en integrar **SOAR** en **ICa**, estableciendo una comunicación entre un robot Pionner 2AT-8 autónomo, **Higgs**, y un programa **SOAR**.

A la vista de los resultados obtenidos, se plantean las siguientes líneas futuras de desarrollo:

1. Desarrollo de un programa complejo **SOAR** que controle de forma inteligente el comportamiento de **Higgs**

   El control desarrollado por **SOAR** sobre **Higgs** es muy simple, por lo que una línea futura sería el desarrollo de una aplicación **SOAR** más compleja, que controlara a **Higgs** en todos sus aspectos dotándole de un comportamiento lo más sofisticado.

2. Desarrollo de una aplicación de comunicación con otros módulos de la arquitectura **ICa**.

   Este proyecto se desarrolla en el grupo ASLab, donde a la par se desarrollan otros proyectos con el mismo enfoque modular, ésto es, se desarrollan aplicaciones específicas que pueden ejecutarse simultáneamente. Por ejemplo, dos de estas aplicaciones son: el módulo encargado del mapeo emocional de los estados sensoriales de **Higgs** utilizando una cara antropomórfica y el módulo de generación de voz.

   Debido a ello, una línea futura consistiría en desarrollar una aplicación que permitise a dichos módulos disponer de la información proporcionada por la aplicación **SOAR**, es decir, una aplicación que abriera la comunicación entre el cliente **SOAR-CORBA** y cualquier módulo de la arquitectura que necesitase los datos proporcionados por éste.

3. Desarrollo de un nuevo servidor para **Higgs**.

   El presente proyecto se basa en un servidor que comunica de forma inalámbrica a **Higgs** con la red de ordenadores de ASLab, como se vio en el Capítulo 3, Sección 3.1.1.

   Dicho servidor satisfacía las necesidades de otro proyecto [Pareja, 2004] y no necesariamente de éste. Por lo que otra línea futura para mejorar esta aplicación sería el diseño de un nuevo servidor; ya que Aria (API para el control del robot) proporciona más funciones de las que se implementaron en aquel momento en dicho proyecto. Algunas de las funciones que proporciona Aria que no son utilizadas son: funciones que dicen si el robot se está moviendo o no, si está atascado o si tiene los motores encendidos. Por tanto, con el desarrollo de un nuevo servidor que implemente dichas funciones, se podrían mejorar las prestaciones que se obtendrían de un programa **SOAR**, ya que se dispondría de más información.

4. Desarrollo de una aplicación de comunicación entre **SOAR** y **Higgs** utilizando GNU/Linux como sistema operativo.

   Debido a las limitaciones que supuso tener unas librerías para la comunicación con **SOAR** en Windows, la aplicación desarrollada utiliza Windows XP como sistema operativo de trabajo.

   Por tanto, otra línea futura de investigación puede ser el desarrollo de una aplicación que gestione la comunicación con **SOAR** utilizando GNU/Linux como sistema operativo.

Apéndice A

SOAR Software License

SOAR uses the BSD license. This license is explained in easy-to-understand terms here:

http://explanation-guide.info/meaning/BSD-license.html

A.1. License Text

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE **SOAR** CONSORTIUM "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE **SOAR** CONSORTIUM OR

CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of Carnegie Mellon University, the University of Michigan, the University of Southern California/Information Sciences Institute, **SOAR** Technology, or the **SOAR** consortium.

Apéndice B

Herramientas software utilizadas

En esta sección se quiere mencionar las herramientas software utilizadas para el desarrollo completo del proyecto fin de carrera:

LaTeX: LaTeXes un procesador de texto, de gran potencialidad en el manejo de fórmulas matemáticas, cuadros y tablas. La idea principal de LaTeXes ayudar a quien escribe un documento, a centrarse en el contenido más que en la forma. Es muy utilizado para la composición de tesis y libros técnicos dado que la calidad tipográfica de los documentos realizados con LaTeX es comparable a la de una editorial científica de primera línea. LaTeXes Software Libre bajo licencia LPPL. Gracias a él se ha escrito el presente documento.

PowerPoint: es un popular programa de presentación desarrollado para sistemas operativos Microsoft Windows y Mac OS. Ampliamente usado en distintos campos como en la enseñanza, negocios, etc. Según las cifras de Microsoft Corporation, cerca de 30 millones de presentaciones son realizadas con PowerPoint cada día. Forma parte de la suite Microsoft Office. Gracias a esta herramienta se han hecho las imágenes del documento y la presentación.

Microsoft Visual C++ Es una suite de programación para el sistema operativo Microsoft Windows. Conformada por varios lenguajes de programación, entre ellos Visual C++ y Visual Basic. Gracias a esta herramienta se han desarrollado los programas necearios para la realización del proyecto fin de carrera.

Bibliografía

[Clavijo et al., 2000] Clavijo, J. A., Segarra, M. J., Sanz, R., Jiménez, A., Baeza, C., Moreno, C., Vázquez, R., Díaz, F. J., and Díez, A. (2000). Real-time video for distributed control systems. In *Proceedings of IFAC Workshop on Algorithms and Architectures for Real-time Control, AARTC'2000*, Palma de Mallorca, Spain.

[Grisby, 2004] Grisby, D. (2004). *The omniORB version 4.0.* AT&T Laboratories Cambridge.

[Laird, 2004] Laird, J. E. (2004). *The Soar 8 Tutorial.* University of Michigan.

[Laird et al., 1999] Laird, J. E., Congdon, C. B., and Coulter, K. J. (1999). *The Soar User´s Manual.* Electrical Engineering and Computer Science Department, University of Michigan.

[Laird et al., 1987] Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: an architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64.

[Marinier, 2004] Marinier, B. (2004). *SGIO User´s Guide.*

[Pareja, 2004] Pareja, I. (2004). *Sistema de comunicaciones de la plataforma móvil Pioneer 2-AT8.* PhD thesis, ETSII.

[Sanz, 2002] Sanz, R. (2002). Embedding interoperable objects in automation systems. In *Proceedings of 28th IECON, Annual Conference of the IEEE Industrial Electronics Society*, pages 2261–2265, Sevilla, Spain. IEEE Catalog number: 02CH37363.

[Sanz, 2003] Sanz, R. (2003). The IST HRTC project. In *OMG Real-Time and Embedded Distributed Object Computing Workshop*, Washington, USA. OMG.

[Sanz et al., 1999a] Sanz, R., Alarcón, I., Segarra, M. J., de Antonio, A., and Clavijo, J. A. (1999a). Progressive domain focalization in intelligent control systems. *Control Engineering Practice*, 7(5):665–671.

[Sanz et al., 1999b] Sanz, R., Matía, F., and Puente, E. A. (1999b). The ICa approach to intelligent autonomous systems. In Tzafestas, S., editor, *Advances in Autonomous Intelligent Systems*, Microprocessor-Based and Intelligent Systems Engineering, chapter 4, pages 71–92. Kluwer Academic Publishers, Dordretch, NL.

[Sanz et al., 2001] Sanz, R., Segarra, M., de Antonio, A., and Alarcón, I. (2001). A CORBA-based architecture for strategic process control. In *Proceedings of IFAC Conference on New Technologies for Computer Control*, Hong Kong, P.R. of China.

[Sanz et al., 2000] Sanz, R., Segarra, M., de Antonio, A., Alarcón, I., Matía, F., and Jiménez, A. (2000). Plant-wide risk management using distributed objects. In *IFAC SAFEPROCESS'2000*, Budapest, Hungary.

[Sanz et al., 1999c] Sanz, R., Segarra, M. J., de Antonio, A., and Clavijo, J. A. (1999c). ICa: Middleware for intelligent process control. In *IEEE International Symposium on Intelligent Control, ISIC'1999*, Cambridge, USA.

[Sanz and Zalewski, 2003] Sanz, R. and Zalewski, J. (2003). Pattern-based control systems engineering. *IEEE Control Systems Magazine*, 23(3):43–60.

[Schildt, 1996] Schildt, H. (1996). *C++ PARA PROGRAMADORES*. McGRAW-HILL.

[Vinoski, 1999] Vinoski, M. H. . S. (1999). *Advanced CORBA Programming with C++*. Addison Wesley.

[Welch, 1997] Welch, B. B. (1997). *Practical Programming in Tcl and Tk*. Prentice Hall, second edition.