Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingeniería de la Computación
Laboratorio de Bio-Robótica

Planeación de acciones utilizando una máquina de inferencias y modelos de redes de Petri

T E S I S

QUE PARA OBTENER EL GRADO DE:

Maestro en Ciencias de la Computación

PRESENTA:

Adrián Revuelta Cuauhtli

DIRECTOR DE TESIS:

Dr. Jesús Savage Carmona

Ciudad de México, D. F. Noviembre, 2014

Planeación de acciones utilizando una máquina de inferencias y modelos de redes de Petri

por

Adrián Revuelta Cuauhtli

Tesis presentada para obtener el grado de

Maestro en Ciencias de la Computación

en el

Posgrado en Ciencia e Ingeniería de la Computación

Universidad Nacional Autónoma de México

Ciudad de México, D. F.. Noviembre, 2014

AGRADECIMIENTOS

Ejemplo:
Gracias a cada una de las personas que me apoyaron e hicieron que este sueño se cristalizara en una hermosa realidad.

Se agradece a la DGAPA-UNAM por el apoyo proporcionado para la realización de esta tesis a través del proyecto PAPIIT IN117612, Robot de Servicio para Asistencia a Adultos Mayores y en Sistemas Hospitalarios"

TABLA DE CONTENIDO

Planeación de acciones utilizando una máquina de inferencias y modelos de redes de Petri

por

Adrián Revuelta Cuauhtli

Resumen

Aquí se redacta el resumen en español.

Planeación de acciones utilizando una máquina de inferencias y modelos de redes de Petri

by

Adrián Revuelta Cuauhtli

Abstract

Here goes the english abstract.

CAPÍTULO

1

INTRODUCCIÓN

Desde sus inicios, los humanos han construido herramientas que les permitan realizar su trabajo más fácilmente o con mayor eficiencia. Esta práctica sigue vigente, e incluso más pronunciada, gracias al rápido avance de la tecnología y a que ésta ha permeado en distintos niveles de la sociedad, resultando en una comunidad generadora de ideas y retos. Esto produjo una vertiente en la cual se persigue la creación de sistemas cada vez más independientes y que realicen tareas de complejidad cada vez mayor, hasta llegar a crear sistemas autónomos que ayuden a llevar a cabo algunas actividades que cualquier persona podría hacer; tal es el caso de los robots de servicio. En este trabajo se referirá en ocasiones a un sistema autónomo como agente (inteligente), para no confundir con el término *sistema* con el que muchas veces se refiere al entorno donde se desarrollan las acciones y a la dinámica del mismo, o bien a un sistema computacional o robótico.

Se espera que un robot de servicio de propósito general sea capaz de realizar un conjunto de actividades de alto nivel con la menor ayuda posible en un ambiente determinado; estas actividades involucran comportamientos básicos como navegar, reconocer y manipular objetos, así como reconocer e interactuar con personas.

El desarrollo de la robótica en el ámbito de la asistencia en la vida cotidiana representa muchos retos aún por superar, inherentes de un ambiente dinámico y poco controlado, y de la complejidad del sensado y de las acciones que debe llevar a cabo, así como del nivel de abstracción que requiere la tarea. Así mismo, el alcance y complejidad de las tareas que realice el robot depende de la arquitectura con que esté construido, que a su vez depende de la aplicación.

Estas dificultades frecuentemente se traducen en que el robot pretenderá actuar de una forma que no corresponde con lo que el usuario espera o le gustaría que hiciera. Por esta razón es importante modelar la estructura de la tarea de una forma que permita manejar de manera eficaz la mayor cantidad de situaciones que se puedan presentar de este tipo.

En este sentido, se han planteado distintos enfoques de control de ejecución, los autores en [1] mencionan cuatro. Uno de estos enfoques es el reactivo: "no pienso, actúo (reacciono)". En este esquema el agente simplemente actúa o reacciona a lo que detectan sus sensores para llegar a, o mantenerse en, un estado adecuado; un ejemplo de sistema autónomo que utiliza este esquema es un robot seguidor de línea. Este enfoque es adecuado para ambientes estáticos y muy controlados; sin embargo, conforme avanza la tecnología, y con ella las expectativas que se tienen de los robots autónomos, un sistema puramente reactivo resulta no ser lo suficientemente poderoso para realizar tareas más complejas.

Otro enfoque es el deliberativo: "pienso, luego actúo". En este esquema, el agente mantiene una representación interna de su entorno y de sí mismo, para primero "decidir"qué debe hacer para llegar a un estado deseado, y después actuar. El esquema deliberativo representa un avance en la complejidad de las tareas que puede realizar un agente; sin embargo, falla cuando las expectativas son distintas a la realidad del entorno, especialmente cuando el ambiente es poco controlado.

Posteriormente se comenzó a utilizar un enfoque híbrido: "pienso y actúo concurrentemente". En este esquema, el agente tiene una parte reactiva que sirve para reaccionar a eventos imprevistos en el entorno, y una parte deliberativa que le permite "decidir"qué debe hacer para lograr los objetivos. La parte difícil es unificar ambas partes para obtener el resultado deseado.

Finalmente, se considera le enfoque basado en comportamientos: "pienso cómo actuar". En este enfoque existen diferentes comportamientos, que corresponden a diferentes módulos del robot, cada uno con funciones específicas. Los comportamientos reciben información de los sensores, mantienen una representación interna del mundo y deciden cómo actuar en consecuencia. Comúnmente existe un planeador para coordinar sus efectos.

Conforme los robots se han hecho capaces de realizar acciones cada vez más complejas, la necesidad de una planeación eficiente se hace más tangible, e incluso más en el contexto de robots móviles, con ambientes dinámicos y poco controlados, donde debe primero identificar la situación en la que se encuentra, y después "decidir"qué hacer a continuación para alcanzar sus objetivos.

Para que estos problemas sean manejables por un robot, se le debe dar de alguna forma una especificación de la tarea para que sepa qué hacer en cada situación que encuentre. A estas especificaciones de tareas usualmente se les refiere como planes y el resultado de la parte deliberativa de un sistema se concreta en un plan más refinado, que incluya las acciones primitivas que el robot debe usar para llegar al estado final.

Un planeador debe, de manera general, tomar en cuenta los recursos que tiene disponibles y formar un plan que de llevarse a cabo correctamente, influiría en el cumplimiento de los objetivos planteados. Existen varios aspectos importantes a considerar en un planeador, como la eficiencia computacional del planeador o el espacio de planes que puede generar, es decir, para qué aplicaciones es capaz de planear.

Con esto en mente, se han utilizado planeadores de dominio específico para resolver los problemas eficientemente, y por lo tanto, se puede hablar de distintos tipos de planeación, como planeación de movimientos, planeación temporal o calendarización de tareas, planeación de asignación de recursos, planeación de acciones, etc.

En este trabajo se desarrolla un esquema de planeación de acciones de un sistema autónomo, particularmente enfocado a un robot de servicio.

1.1. Presentación

Planeación de acciones en general hace referencia a la selección y ordenamiento de acciones dentro de un grupo de posibilidades con la finalidad de llevar a cabo una tarea más compleja o alcanzar un objetivo de manera eficiente. En el campo de robótica, consiste en dirigir las acciones del robot de tal forma que realice lo que se le pide o se espera de él. Para lograr esto debe tomar información del medio a través de sus sensores, compararla con su base de conocimiento y procesarla para después realizar alguna acción utilizando sus actuadores.

La planeación de acciones en cualquier sistema cobra gran importancia cuando se desarrolla en un ambiente dinámico y poco controlado, donde las expectativas pueden ser muchas e incluso todas distintas de las observaciones realizadas. Tal es el caso de los robots de servicio o robots domésticos, que deben interactuar de manera natural las personas en un entorno social común, como es en un hogar, en una escuela, en una oficina, o en un hospital.

El esquema de planeación que se utilice definirá la dificultad que representa definir nuevas tareas para el robot, así como qué tipos de planes se pueden diseñar, y cómo actuará el robot cuando los planes no sean exitosos. El esquema de planeación debe ser adecuado para que la ejecución de las tareas sea exitosa.

En este trabajo se presenta una propuesta que pretende atacar dos retos comunes en la robótica que derivan de los puntos arriba mencionados: la tolerancia a fallas, para encontrar cursos de acción cuando no se encuentran las expectativas esperadas; y la concurrencia de procesos o la ejecución multitarea, por ejemplo: mientras el robot realiza una tarea y es interrumpido por una persona, puede suspender la primera tarea para atender a la persona, y después continuar con la ejecución.

El motor de planeación desarrollado en esta tesis está basado en dos esquemas de planeación existentes: *Planeación por búsqueda en espacio-plan* y *Red de tareas jerárquicas (Hierarchical Task Network, HTN)*. El diseño de los planes hace uso de los modelos de redes de Petri de alto nivel. La implementación está hecha sobre el lenguaje de programación lógica CLIPS, para hacer las inferencias sobre los planes y los hechos del mundo; y sobre python para diseñar los planes de manera gráfica.

A pesar de que el título menciona únicamente la planeación, en realidad se trata tanto de un motor tanto de planeación como de ejecución integrados, de modo que exista una planeación dinámica, donde las observaciones realizadas durante la ejecución permitan la replaneación cuando fuera necesario.

1.2. Motivación

Este trabajo se realizó en el contexto del *Laboratorio de Bio-Robótica* de la Facultad de Ingeniería de la UNAM, dirigido por el Dr. Jesús Savage Carmona, donde se desarrollan robots de servicio de propósito general. Particularmente en 2011 comenzó el proyecto del robot de servicio *Justina*, con el que se realiza investigación en distintas áreas de la robótica.

En este laboratorio participan alumnos de distintos niveles académicos en distintas áreas de especialización, desde nivel licenciatura hasta doctorado y post-doctorado, con un equipo multidisciplinario que ha incluido en distintos períodos de tiempo estudiantes de diferentes ingenierías, e incluso estudiantes de algunas ciencias sociales y biológicas. El trabajo que se realiza aquí, frecuentemente consiste en desarrollos de tesis y servicio social, por lo que es frecuente que se incorporen nuevos estudiantes, a la vez que otros se vayan al concluir sus trabajos.

Esta situación produce algunos inconvenientes con los que hay que lidiar para que el trabajo del laboratorio sea productivo, aunados a las dificultades inherentes a la investigación y la implementación de la misma; entre ellas, los nuevos estudiantes se tienen que incorporar al equipo y deben ser capacitados, para posteriormente analizar de qué manera pueden contribuir al laboratorio. Cuanto mayor sea la rotación de personal, más difícil es la tarea de utilizar, mantener y mejorar los desarrollor previos.

Por esta razón, uno de los objetivos de esta tesis es crear una interfaz que permita diseñar los planes de manera gráfica, de modo que sea más fácil e intuitivo utilizarlo para personas que se incorporan al laboratorio. Con este fin se utilizaron modelos de redes de Petri como formalismo de representación de los planes, particularmente la clase de diagramas conocida como redes de Petri de alto nivel. El motor de planeación que utiliza diseños hechos con modelos de redes de Petri, como se describe en esta tesis, es utilizado por la robot Justina en las tareas que ella ejecuta para ayudar a adultos mayores y para la atención hospitalaria.

1.3. Objetivos

Los objetivos principales del presente trabajo son: (1) el desarrollo de un motor de planeación de acciones para un robot de servicio, y (2) el desarrollo de una interfaz gráfica que facilite el diseño de planes y lo haga una tarea más intuitiva de lo que podría ser codificarlos en algún lenguaje de programación.

Particularmente, las características buscadas en el motor de planeación son las siguientes:

- Especificar planes como objetivos que se tienen que cumplir, de manera concisa y clara.
- Ejecutar planes concurrentes y multitareas.
- Especificar planes alternativos y correctivos como rutinas de recuperación cuando la ejecución no sea exitosa.
- Especificar un esquema de prioridades para definir qué acción o acciones ejecutar a continuación, que contemple la ejecución asíncrona de algunas acciones.

El planeador está implementado utilizando un sistema experto que hace uso de una máquina de inferencias para unificar las reglas de ejecución con hechos sobre el mundo. Los diagramas de redes de Petri diseñados en la herramienta gráfica serían traducidos a reglas útiles para el motor de planeación.

1.4. Organización de la tesis

En este primer capítulo se ha presentado una introducción a los problemas que pretende solucionar este trabajo, las motivaciones, y los objetivos planteados. A continuación, en el capítulo 2, se presentan algunos antecedentes en los que está basado este trabajo, así como otros trabajos relacionados y el estado del arte. El capítulo 3 presenta los detalles del motor de planeación diseñado en esta tesis. El capítulo 4 presenta la herramienta desarrollada para diseñar los planes gráficamente.

Posteriormente, el capítulo 5 presenta algunos resultados en pruebas que se hicieron con le robot de servicio, y argumenta la utilidad y ventajas de este planeador con respecto a otros comúnmente utilizados, así como algunas desventajas y otros contrastes enconrtados. El capítulo 6 presenta las conclusiones y discute algunas opciones de trabajo a futuro o modificaciones al proyecto que podrían ser útiles.

Finalmente se presentan tres apéndices, uno conteniendo una introducción al lenguaje de programación CLIPS, sobre el que está desarrollado el motor de planeación; y otros dos describiendo dos herramientas también creadas como parte del desarrollo de esta tesis, para conectar el motor de planeación en CLIPS con el robot de servicio.

CAPÍTULO

2

ANTECEDENTES

En este capítulo se presentan los conceptos y temas en los cuales está basada esta tesis. Se presenta primero el estado del arte, mencionando algunos planeadores y controladores utilizados actualmente. En seguida se describe un modelo conceptual de un sistema para plantear un problema de planeación. Posteriormente se expone el enfoque de planeación clásica, que hace uso de la lógica proposicional y de primer orden para representar un problema y su solución. Se hace énfasis particular en la planeación por búsqueda en el espacio-plan, que forma parte de la planeación clásica, como método de búsqueda elegido para resolver dependencias de un plan inicial.

Después se presenta el concepto de redes jerárquicas de tareas, que simplifica el diseño de los planes, los hace más manejables y propicia la reutilización de rutinas presentes en distintas tareas. Finalmente, se describen los modelos de redes de Petri, que son un formalismo que permite identificar y analizar distintas propiedades de un sistema, a la vez que permiten diseñar las reglas de manera gráfica.

En el apéndice A se presenta una introducción al lenguaje de programación lógica CLIPS, utilizado para el desarrollo del sistema experto que unifica las reglas con los hechos sobre el mundo y los planes.

2.1. Estado del arte

En 1971, investigadores de Stanford desarrollaron una herramienta para la demostración de teoremas y la solución de problemas llamado STRIPS [2] que ha sido referencia para muchos planeadores desde entonces. Los planeadores de tipo STRIPS utilizan una descripción del mundo y de operadores que son construidas con fórmulas bien formadas de cálculo de predicados de primer orden, e inlcuyen precondiciones, hechos que son verdaderos después de aplicar el operador, y hechos que han dejado de ser verdaderos. Así, al aplicar un operador, se modificarían los hechos del mundo, lo cual habilitaría a nuevos operadores para ser aplicados.

Los planeadores de tipo STRIPS comúnmente realizan búsquedas en el espacio-estado, donde el nodo raíz es el estado inicial del mundo, y el nodo que se busca es el estado final o deseado. La búsqueda se hace a partir de la "diferencia" entre estos estados, y se eligen a los operadores que contribuyan a cambiar el estado hacia uno más parecido al estado final. Estos planes generan una secuencia totalmente ordenada de acciones, sin embargo, varios autores han defendido que un plan podría ser parcialmente ordenado para poder ser ejecutado con mayor naturalidad y eficiencia. Sacerdoti fue uno de los primeros en discutir esto en [3].

Después, Scott Penberthy y Daniel Weld desarrollaron un planeador llamado UCPOP [4], que de manera similar utiliza una descripción de los operadodes posibles con sus precondiciones y efectos; sin embargo, en lugar de utilizar operadores que busquen alcanzar el estado final a partir del estado inicial, UCPOP incluye a los estados inicial y final como si fueran acciones "dummy'. El estado inicial es la instanciación de un operador con precondiciones nulas y cuyos efectos son el estado inicial; y el estado final corresponde con un operador sin efectos, y cuyas precondiciones son el estado final deseado.

El planeador UCPOP realiza una búsqueda en espacio-plan, donde el objetivo no es directamente llegar a un estado particular, sino satisfacer las precondiciones aún no satisfechas del plan actual. La búsqueda en espacio-estado busca una configuración del estado actual del mundo que cumpla con la especificación del estado final. La búsqueda en espacio-plan, en cambio, realiza una búsqueda en la configuración de distintos planes, agregando acciones, reordenando acciones o modificando restricciones en las acciones.

Posteriormente, surgieron los planeadores basados en redes jerárquicas de tareas (HTN, *Hierarchical Task Networks*) [5], donde existen tareas primitivas y no primitivas, y el conocimiento sobre cómo resolver el problema está embebido en la descomposición de las tareas en un plan concreto. Nau *et al.* presentan en [6] un enfoque basado en HTNs, donde utilizan LISP para representar tareas primitivas y tareas no primitivas, a la vez que mantienen una representación del estado del mundo.

De manera conceptual, estos planeadores obtuvieron buenos resultados para encontrar planes aparentemente eficientes y efectivos. Sin embargo, al pasar a la práctica, se hacen notorios los efectos del llamado *reality gap*: lo que ocurre en aplicaciones que se ejecutan en un ambiente real no es igual a lo que ocurre durante la simulación o planeación. Particulamente en el campo de la robótica, los sensores y los actuadores pueden tener errores, tales como que los procesos de visión no hayan reconocido un objeto, o que el actuador no haya sido capaz de tomar un objeto y lo haya tirado al intentarlo.

Por esta razón, se han implementado planeadores que al mismo tiempo son ejecutores. Los planes tienen una estructura jerárquica como la utilizada en el enfoque de HTN, y la planeación/ejecución se ejecuta de manera *greedy*; es decir, ejecuta paso a paso sin haber generado el plan completo. De esta forma, se podrían revisar los efectos de las acciones en el ambiente, y continuar planeando conforme al resultado.

Las máquinas de estados jerárquicas son una técnica muy recurrida para este fin. Estructuran procedimientos en distintas jerarquías, donde cada nivel jerárquico resuelve un problema cada vez más específico. El módulo de ROS llamado SMACH [7] se utiliza para el diseño y ejecución de máquinas de estado jerárquicas. ROS (Robot Operating System, `http://www.ros.org/`), lejos de ser un sistema operativo, funge como *middleware* para manejar fácilmente los recursos del robot como los actuadores y los sensores. Incluye un conjunto de herramientas y librerías que facilitan el desarrollo de programas que se comunican e interactúan entre sí. Actualmente es una de las plataformas de robótica más utilizadas, y las herramientas compatibles con ROS son, entonces, ampliamente utilizadas también.

El grupo GOLEM, otro grupo interdisciplinario de la UNAM que trabaja con robots de servicio, utiliza un enfoque similar a las máquinas de estados jerárquicas, basado en Redes de Transiciones Recursivas (RTN, *Recursive Transition Networks*). SitLog [8] es el nombre de la herramienta que desarrollaron, y en ella modelan situaciones en que se puede encontrar el robot y definen transiciones a otras situaciones a través de las expectativas que puede manejar cada situación.

Por otra parte, ha habido desarrollos para diseñar planes de manera gráfica, de modo que el diseño se simplifique y el tiempo de desarrollo se acorte. Particularmente los modelos de redes de Petri han sido utilizados con este propósito. Vittorio Amos Ziparo y Lucca Iocchi en [9] presentaron un framework para diseñar y ejecutar planes robóticos con redes de Petri, donde definen distintos tipos de acciones y estructuras de control para definir la secuencia de las acciones.

De manera similar, Hugo Costelha y Pedro Lima en [10] presentaron un framework para representar tareas robóticas con redes de Petri, que utilizan distintos tipos de lugares para modelar hechos sobre el mundo, acciones y subtareas (redes de Petri anidadas en una jerarquía menor), así como distintos tipos de diagramas para representar controladores de tareas, efectos de las acciones y modelos sobre el ambiente. Todo esto con el fin de construir un modelo de cadenas de Markov y realizar diversos análisis sobre el producto.

Bruno Lancerda y Pedro Lima en [11] propusieron un controlador que utiliza por una parte una red de Petri para modelar las acciones posibles para el agente, y en paralelo un conjunto de restricciones de lógica temporal lineal (LTL, *Linear Temporal Logic*) para restringir las acciones que debía ejecutar según la actividad que realizara.

Robograph [12] es un IDE para diseñar tareas robóticas con redes de Petri en una interfaz gráfica. Está diseñada particularmente para comunicarse con otros módulos, de manera similar a como lo hace el trabajo desarrollado en esta tesis, a través de mensajes de comunicación entre procesos (IPC, *Inter-Process Comunication*). Así mismo, existen varios diagramadores como pipe2 [13], pneditor [14], viptool [15], CPN (Colored Petri Nets) [16], o TAPAAL [17]; que servirían para diseñar redes de Petri. Algunos incluyen ciertas herramientas para hacer análisis de procesos, diseño de sistemas, etc.

TAPAAL es útil para modelar y analizar tiempos involucrados en la ejecución de las tareas, mientras que CPN es un subconjunto de las redes de Petri de alto nivel, donde los tokens presentan un color, que significaría otro tipo de información asociada con la ejecuión.

No obstante, la mayoría de los trabajos mencionados trabajan con redes de Petri simples o binarias, y se necesitarían ayuda de otras herramientas para pasar información entre las tareas de un plan. El trabajo aquí desarrollado utiliza redes de Petri de alto nivel, concretamente redes de Petri de Predicado/Transición, que agregan expresividad a los diseños y proporcionan una estructura para pasar información entre las tareas, además de que es compatible con el sistema de mensajes que se utiliza en el laboratorio para comunicar distintos módulos del robot.

Actualmente en el laboratorio de Bio-Robótica se utilizan máquinas de estados jerárquicas para definir los planes de las tareas que tiene que ejecutar el robot. El robot utiliza una arquitectura híbrida, tratando de seguir el modelo de ViRBot, propuesto por Savage *et al.* en [18], a través de la herramienta BlackBoard [19], desarrollada en el mismo laboratorio. La figura 2-1 muestra una segunda versión de este modelo que ha sido desarrollada en el laboratorio. El trabajo actual haría el trabajo del planeador, ejecutor y supervisor mostrado en esta figura.

El BlackBoard es una herramienta de paso de mensajes y concentradora de variables compartidas que comunica a distintos módulos del robot utilizando los patrones de llamada a procedimientos remotos (RPC, *Remote Procedure Calls*) a través de comandos, y publicación-subscripción, a través de las variables compartidas. Así, el planeador envía comandos a los otros módulos para realizar las tareas que se le piden, y cualquier módulo publica en las variables compartidas cualquier información que podría ser útil para otro módulo.

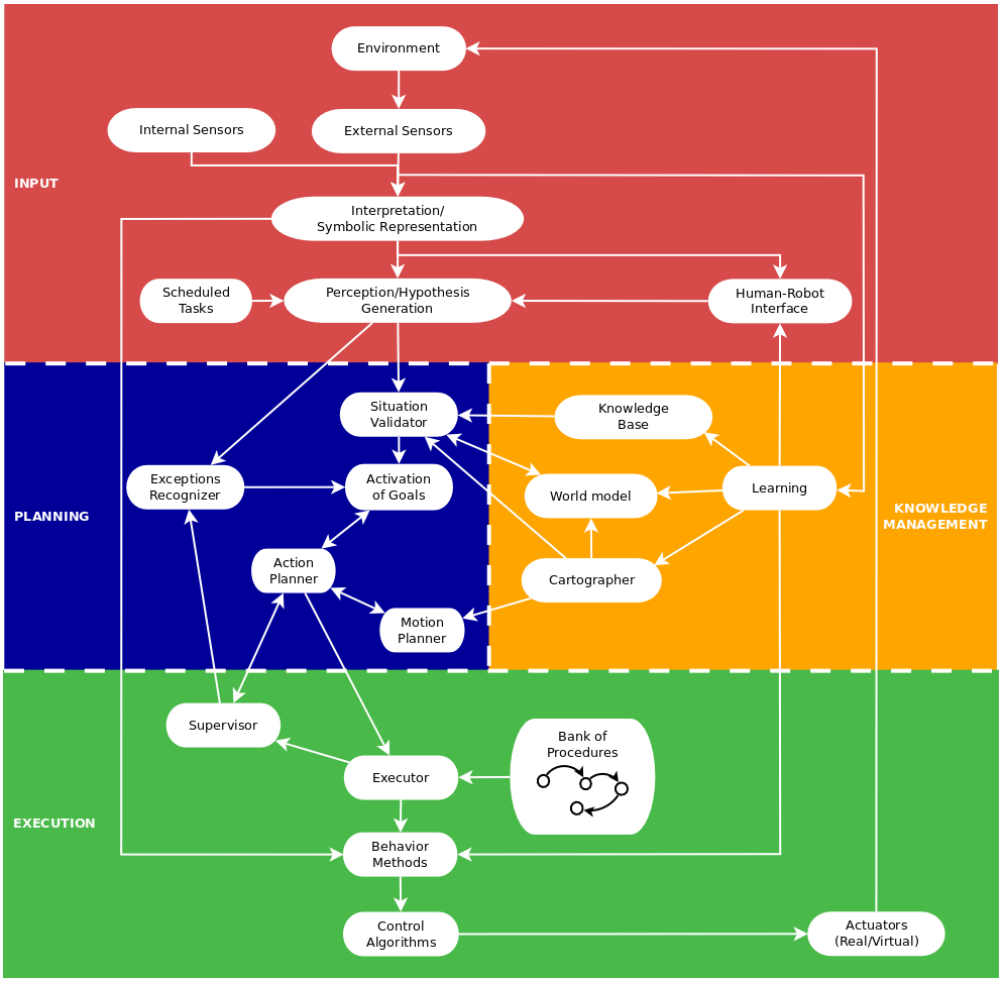


Figura 2-1: Versión 2 del modelo ViRBot.

2.2. Modelo conceptual para planeación

Para realizar la planeación, se necesita un modelo que represente con *cierto grado de precisión* el entorno en el que se desarrollarán las tareas. El modelo debe ser capaz de abstraer el estado del mundo y las posibles acciones o eventos que sucedan en él, así como los cambios que estos produzcan.

Para esto se utiliza un modelo de un sistema dinámico, que puede diferir significantemente de un problema real y es independiente de las implicaciones algorítmicas y de eficiencia de la implementación, pero que es útil para presentar conceptos básicos necesarios para modelar un problema y así diseñar algoritmos de planeación. El modelo que comúnmente se utiliza como referencia para representar un ambiente es el sistema de estados y transiciones (también conocido como sistema de eventos discretos).

Formalmente, un sistema de estados y transiciones es una tupla $\Sigma = (S, A, E, \gamma)$, donde:

- $S = \{s_1, s_2, ...\}$ es un conjunto finito o recursivamente enumerable de estados;
- $A = \{a_1, a_2, ...\}$ es un conjunto finito o recursivamente enumerable de acciones;
- $E = \{e_1, e_2, ...\}$ es un conjunto finito o recursivamente enumerable de eventos; y
- $\gamma : S \times A \times E \rightarrow 2^S$ es una función de estado-transición.

Un sistema de estados y transiciones puede a su vez representarse gráficamente como un grafo dirigido, donde los nodos son los estados del sistema, y los arcos representan transiciones. Los arcos estarían etiquetados por acciones y eventos, que producen las transiciones de un estado a otro. La representación gráfica de un ejemplo sencillo se muestra en la figura 2.2. Este modelo es apropiado para modelar un ambiente dinámico en donde el agente puede realizar varias acciones con resultados distintos, modelados como eventos, o bien eventos inherentes al sistema, al mismo tiempo que el agente realiza una acción. Sin embargo, es también conveniente introducir un evento neutral ϵ para contemplar transiciones que ocurren solamente debido a acciones y, simétricamente, una acción neutral no-op para denotar transiciones causadas únicamente por un evento. Escribimos $\gamma(s, a, \epsilon)$ como $\gamma(s, a)$ y $\gamma(s,\text{no-op}, e)$ como $\gamma(s, e)$.

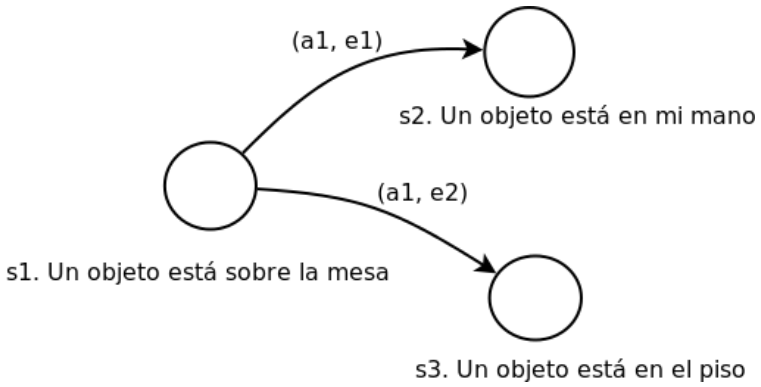


Figura 2-2: Ejemplo de Sistema de Estados y Transiciones.

Este ejemplo involucra a una acción y dos eventos: un robot intenta tomar un objeto (acción $a1$), y lo toma exitosamente (evento $e1$), o se le cae (evento $e2$), por lo que el sistema en general pasará a una configuración distinta.

De este modo, las acciones y eventos contribuyen a la evolución del sistema a través de distintos estados. La diferencia está en que las acciones son provocadas o controladas por el agente que las realiza, mientras que los eventos forman parte de la dinámica interna del sistema, o provienen de las acciones de otros agentes que interactúen en él, y no se pueden controlar ni activar. Un planeador debe tomar en cuenta los eventos que podrían ocurrir en un determinado estado o con una determinada acción para decidir qué acciones incluir en el plan.

Aunado a esto, el sistema podría no ser completamente observable, es decir, que el robot podría no poder sensar todas las características del ambiente en todo momento, lo cual complica la planeación. Dependiendo del ambiente y la aplicación de planeación, en ocasiones habrá que considerar métodos probabilísticos o técnicas de estimación para compensar estas limitantes. Las observaciones se pueden modelar como una función $\eta : s \rightarrow o$, donde $s \in S$ es un estado del sistema y o corresponde a las observaciones realizadas en ese estado. Cuando el sistema es completamente observable, la función η es la función identidad.

2.2.1. Restricciones del modelo

Como se ha mencionado anteriormente, el modelo presentado no necesariamente representa un sistema real, sino que sirve como base para contemplar los elementos presentes en un sistema. Existe una serie de restricciones que definen las características del modelo, y que de eliminarse o relajarse, permitiría modelar un sistema bastante más realista. A continuación se enlistan las catacterísticas de este modelo:

Σ **finito.** Tiene un conjunto finito de estados.

Σ **completamente observable.** Si es completamente observable, el controlador tiene un conocimiento preciso del estado del sistema. En este caso, la función observación η es la función identidad.

Σ **determinista.** Si para cada estado s y para cada evento o acción u, $|\gamma(s,u)| \leq 1$. Si una acción es aplicable a un estado, su aplicación lleva al sistema determinista a un sólo estado distinto; similarmente para las ocurrencias de un posible evento.

Σ **estático.** El sistema Σ es estático si el conjunto de eventos E es vacío. Σ no tiene una dinámica interna; se mantiene en un estado hasta que el controlador aplica alguna acción.

Objetivos restringidos. El planeador maneja sólo *objetivos restringidos* que son especificados como un estado meta s_g o un conjunto de estados meta S_g explícitos; el objetivo es cualquier secuencia de transiciones de estado que termine en uno de los estados meta. Objetivos extendidos, como estados que se deben evitar y restricciones en la trayectoria de los estados o funciones de utilidad no son manejados bajo esta restricción.

Planes secuenciales. La solución a un problema de planeación es una secuencia finita linealmente ordenada de acciones.

Tiempo implícito. Las acciones y los eventos no tienen duración; son transiciones de estado inmediatas. Esta restricción está embebida en sistemas de estados y transiciones, un modelo que no representa el tiempo explícitcamente.

Planeación "offline". El planeador no contempla cualquier cambio que pueda ocurrir en Σ mientras está planeando; planea para el estado inicial dado y lo objetivos meta, independientemente de la dinámica actual del sistema, si hubiere.

Debido a la incertidumbre introducida por los eventos y las (posiblemente) limitadas capacidades de observación del agente, en una aplicación un tanto más realista, comúnmente se necesita un control de lazo cerrado para llevar a cabo las tareas. Esto quiere decir que usualmente el planeador está fuertemente acoplado con un ejecutor o controlador y un supervisor, de modo que el planeador pueda adecuar el plan a las situaciones que se vayan presentando en realidad, en tiempo de ejecución. La figura 2-3 muestra un diagrama que ilustra la interacción existente entre estos componentes.

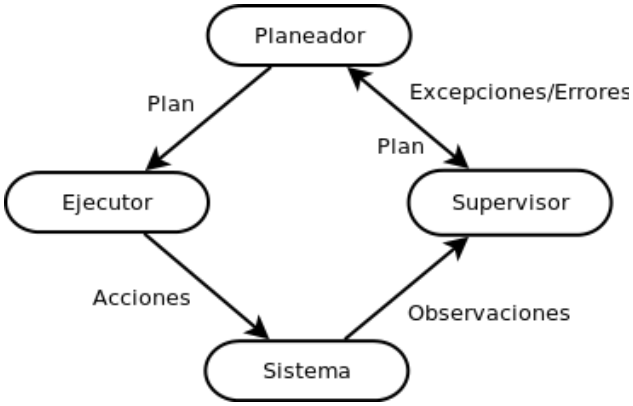


Figura 2-3: Componentes involucrados en la ejecución de un plan y sus interacciones.

2.2.2. Especificación de tareas para la planeación

Los objetivos que se especifican a un planeador, representan implícitamente la realización de una tarea (o varias tareas). Por esto se puede hablar de una especificación de la tarea: se le da al planeador una especificación que tendrá que utilizar como entrada para generar un plan y llegar a un estado final. Algunas maneras comunes de representar los objetivos son:

- La especificación más sencilla consiste en un *estado objetivo* o *estado meta* s_g o un conjunto de *estados meta* S_g. En este caso, el objetivo es cumplido por cualquier secuendia de transiciones de estado que termine en alguno de estos estados meta.

- De manera más general, el objetivo es satisfacer alguna condición en la secuencia de estados seguidos por el sistema. Por ejemplo, se podría querer evitar algún estado, o debería haber estados que el sistema deba alcanzar en algún momento, o estados en los cuales debería quedarse.

- Una especificación alternativa es a través de una función de utilidad en cada estado, con recompensas y castigos. El objetivo es optimizar alguna función compuesta de estas funciones de utilidad (e. r. sumatoria o máximo) a través de la secuencia de estados seguidos por el sistema.

- Otra alternativa es especificar el objetivo como tareas que el sistema debe llevar a cabo. Estas tareas pueden ser definidas recursivamente como conjuntos de acciones u otras tareas.

2.3. Planeación clásica

Como se mencionó anteriormente, la planeación consiste en la selección y ordenamiento de acciones para alcanzar un estado deseado. En aplicaciones reales de planeación, comúnmente el número de sensores que tiene un sistema (o dicho de otro modo, variables sobre el ambiente) es finito y no cambia con regularidad, y aunque el dominio de alguno de estos sensores no sea discreto, interesa interpretar los datos de estos sensores para obtener información de más alto nivel que sí puede ser discretizada, por ejemplo: conociendo las dimensiones de un robot y la posición de un láser montado en él, se pueden procesar sus lecturas para determinar si un obstáculo está lo suficientemente cerca para afectar la trayectoria del robot o no.

Esta interpretación y discretización de los datos permite que se pueda considerar al entorno como un sistema discreto, con variables de dominio finito. De igual forma, los actuadores de un sistema y sus capacidades son también finitos; por lo tanto, el número de acciones que puede generar es finito.

La tarea de planeación consiste entonces en una búsqueda de un camino, desde un estado inicial a un estado final. Y el espacio de búsqueda normalmente puede ser representado como un árbol o un grafo, donde los nodos corresponden *de alguna forma* con una posible solución parcial del problema, y los arcos con las acciones disponibles para modificar la solución actual.

En planeación clásica se utiliza la lógica como herramienta fundamental: el estado del ambiente y las precondiciones y efectos de una acción se pueden representar con enunciados de lógica proposicional o predicados de lógica de primer orden. En general para realizar la búsqueda se define *una forma* de representar las acciones y los estados del sistema, y *una forma* de identificar cuáles son las acciones que más contribuyen a alcanzar el estado objetivo dentro de la lista de acciones aplicables en el estado actual.

En lógica, una expresión se llama aterrizada cuando no contiene símbolos de variables o no contiene variables libres (sus variables han unificado con una constante correspondiente a su dominio, se puede considerar como que ya tienen un valor asignado), mientras que a una expresión que no está aterrizada se dice que tiene variables libres. Para una introducción a lógica proposicional y de primer orden, puede consultar cualquier bibliografía de introducción a la lógica, los libros [20] y [1] contienen un par de capítulos introductorios que podrían ser de utilidad.

Dos de los espacios de búsquedas más comúnmente utilizados en planeación clásica son el espacio-estado y el espacio-plan. En el espacio-estado, cada nodo del árbol corresponde a un estado, y los arcos son las acciones disponibles, de modo que a partir de un estado, dependiendo de la acción que se realice, se llega a un estado distinto. La representación de un sistema de estados y transiciones tiene una equivalencia muy transparente con este espacio de búsqueda; precisamente, una de las ventajas de utilizar este espacio de búqueda, es que mantiene una representación explícita del estado del mundo en cada paso de la búsqueda. Al finalizar la búsqueda, se tendría un plan con acciones totalmente ordenadas.

En el espacio-plan, del cual hablaré más adelante, los nodos del árbol corresponden a refinaciones de un plan inicial que sólo considera al estado inicial y al estado final; este enfoque pretende resolver dependencias entre tareas, pero no impone un orden estricto para cada tarea, sino solamente restricciones de precedencia entre algunas tareas del plan. Es decir, genera un plan parcialmente ordenado.

De estos dos esquemas de planeación, derivan otros nuevos que ayudan a manejar más fácilmente los planes, o considerar planes con restricciones más específicas; entre ellos está la planeación como solución de *Problemas de Satisfacción de Restricciones* (Constraint Satisfaction Problems, CSP), representados por un conjunto de variables, un conjunto de dominios para esas variables, y un conjunto de restricciones sobre las combinaciones de valores que esas variables pueden adoptar. Para un problema que está modelado de esta forma, la búsqueda de la solución consiste en la búsqueda de combinaciones de valores que satisfagan dichas restricciones, lo cual se puede interpretar como el conocido *problema de satisfacibilidad booleana* (SAT), que es NP-completo. Es decir que esta técnica de planeación podría ser poco eficiente para problemas muy grandes o que tienden a crecer fácilmente.

Otro esquema de planeación que extiende a los de planeación lógica es el llamado de *Red de Tareas Jerárquicas (Hierarchical Task Network, HTN)*, del cual también hablaré más adelante, y consiste en que una tarea se descompone en una secuencia de elementos, que pueden ser acciones primitivas u otras tareas de menor jerarquía, que a la vez se descompondrían en más elementos, para así conformar el plan a realizar. Este esquema hace más fácil el diseño de los planes y la administración de las tareas que el robot tiene que realizar.

2.3.1. Representación de un problema de planeación clásica

Un planeador necesita como entrada una descripción del problema a resolver en alguna forma que pueda interpretar. Se necesita entonces, una forma de representar el modelo conceptual presentado en la sección 2.2. La manera más directa de representar los estados de un sistema y las acciones disponibles es a través de la *representación conjuntista* (o *reprsentación por permutaciones*), utilizando proposiciones lógicas, con valores verdadero o falso, para abstraer el conocimiento sobre el sistema.

En esta representación, cada estado del mundo es un conjunto de proposiciones lógicas que son verdaderas, y cada acción contiene dos conjuntos: un conjunto de proposiciones que deben ser verdaderas para que la acción sea aplicable (precondiciones) y un conjunto de proposiciones que se hacen verdaderas (son añadidas al estado actual) o se hacen falsas (son removidas del estado actual, en caso de que existieran en él) al realizar la acción.

Ejemplo 1. Un ejemplo muy simple con esta representación es el sistema de estados y transiciones $\Sigma = (S, A, E, \gamma)$, donde:

- $S = \{\{p, \neg q\}, \{\neg p, q\}\}$, donde p y q se refieren respecticamente a los enunciados "un objeto está en la mesa" y "estoy sosteniendo un objeto".
- $A = \{a_1\}$, donde a_1 se refiere a la acción "tomar objeto", con los conjuntos $precond(a_1) = \{p, \neg q\}$ y $efectos(a_1) = \{q, \neg p\}$.
- $E = \emptyset$
- $\gamma(\{p, \neg q\}, a_1) = \{\neg p, q\}$, γ está indefinida en cualquier otro caso.

Este sistema podría incluir cuatro estados distintos, correspondientes a las permutaciones de los posibles valores de cada enunciado, y contempla sólo una acción y ningún evento, por lo tanto, siendo un sistema determinista, existe sólo una transición: del estado donde hay un objeto en la mesa y el agente no está sosteniendo un objeto, al estado en que no hay un objeto en la mesa y el agente está sosteniendo un objeto. A pesar de lo simple del sistema, ejemplifica claramente cómo se puede representar un sistema de estados y transiciones como un conjunto de proposiciones lógicas para resolver un problema de planeación.

Considere ahora la posibilidad de representar no sólo que exista un objeto en la mesa, sino que un objeto *en particular* está sobre la mesa, por ejemplo, "el libro de matemáticas está sobre la mesa". Es también fácil ver con este ejemplo que los enunciados de lógica proposicional son poco expresivos, y conforme el problema crezca, se necesitarán nuevos enunciados.

La *representación clásica* de un sistema de estados y transiciones es la forma más popular de representar un problema de planeación. En esta representación los estados y acciones son como los descritos para la representación conjuntista, excepto que se utilizan predicados de lógica de primer orden y operadores lógicos, en vez de sólo un conjunto de proposiciones.

Considerando un lenguaje de primer orden $\mathcal{L}$ con un número finito de símbolos de predicado y símbolos de constantes, y ningún símbolo de función, un *estado* es un conjunto de átomos aterrizados de $\mathcal{L}$. Dado que $\mathcal{L}$ no contiene símbolos de función, se garantiza que el conjunto S de todos los estados posibles es finito. Tanto la representación clásica como la conjuntista utilizan el principio llamado *closed world assumption*, que quiere decir que cualquier predicado que no esté explícitamente especificado en un estado, es considerado falso en ese estado; es decir que un átomo p es verdadero en el estado s si y sólo si $p \in s$.

Nótese que aunque $\mathcal{L}$ es un lenguaje de primer orden, un estado no es un conjunto de fórmulas de primer orden, sino solamente de átomos aterrizados. Para evaluar si un estado s satisface un objetivo g ($s \models g$) , donde g es un conjunto de literales (i. e. átomos y átomos negados), habrá que verificar si existe una sustitución σ tal que todo predicado positivo en $\sigma(g)$ está en s, y ningún predicado negado en $\sigma(g)$ está en s.

La función γ se especifica a través de un conjunto de operadores de planeación, que son instanciados en acciones. Las siguientes definiciones fueron tomadas de [21]

Definición 1. Un *operador de planeación* es una tripleta $o = (nombre(o), precond(o), efectos(o))$, donde:

- $nombre(o)$, el nombre del operador, es una expresión sintáctica de la forma $n(x_1, ..., x_k)$, donde n es un símbolo llamado *símbolo de operador*, $x_1, ..., x_k$ son símbolos de variables que aparecen en cualquier lugar de o, y n es único (i. e. no hay dos operadores en $\mathcal{L}$ que tengan el mismo símbolo de operador).

- $precond(o)$ y $efectos(o)$ son conjuntos definidos de manera similar a los definidos para la representación conjuntista, utilizando los predicados de primer orden.

Definición 2. Para cualquier conjunto de literales $\mathcal{L}$, $\mathcal{L}^+$ será el conjunto de todos los átomos (no negados) en $\mathcal{L}$, y $\mathcal{L}^-$ será el conjunto de todos los átomos cuyas negaciones se encuentran en $\mathcal{L}$.

Definición 3. Una *acción* es cualquier instancia aterrizada de un operador de planeación. Si a es una acción y s es un estado, tal que $precond^+(a) \subseteq s$ y $precond^-(a) \cap s = \emptyset$, entonces a es aplicable en s, y el resultado de realizar esta acción es el estado:

$$\gamma(s, a) = (s - efectos^-(a)) \cup efectos^+(a)$$

Definición 4. Sea $\mathcal{L}$ un lenguaje de primer orden con un número finito de símbolos de predicado y de símbolos de constantes. Un *dominio de planeación clásica* en $\mathcal{L}$ es un sistema de estados y transiciones restringido $\Sigma = (S, A, \gamma)$ tal que:

- $S \subseteq 2^{\{\text{todos los átomos de } \mathcal{L} \text{ aterrizados}\}}$;
- $A = \{$todas las instancias aterrizadas de operadores en $O\}$, donde O es un conjunto de operadores como se definieron anteriormente;
- $\gamma(s, a) = (s - efectos^-(a)) \cup efectos^+(a)$ si $a \in A$ es aplicable a $s \in S$, $\gamma(s, a)$ está indefinida en cualquier otro caso; y
- S es cerrado bajo γ, i. e. si $s \in S$, entonces para cada acción a que es aplicable en s, $\gamma(s, a) \in S$.

Definición 5. Un *problema de planeación clásica* es una tripleta $\mathcal{P} = (\Sigma, s_0, g)$, donde:

- s_0, el estado inicial, es cualquier estado en el conjunto S;
- g, el objetivo, es cualquier conjunto de literales aterrizadas, y
- $S_g = \{s \in S |$s satisface g$\}$.

El *planteamiento de un problema de planeación* $\mathcal{P} = (\Sigma, s_0, g)$ es $P = (O, s_0, g)$.

Resulta impráctico enumerar todos los estados posibles para especificar un sistema para un problema de planeación, sobre todo por que, en general, el sistema podría contener un número infinito de estados. Además, los estados a los que se pueda llegar, dependen de las acciones y eventos existentes en el sistema, y del estado inicial, por lo que con sólo especificar un estado inicial y los operadores del sistema, es posible ir construyendo los estados siguientes.

Existen diversas extensiones que se pueden utilizar con la representación clásica de un problema de planeación. Una de estas extensiones es la de precondiciones disyuntivas, donde se contempla la posibilidad de que una acción sea aplicable en más de una situación, i. e. existen conjuntos distintos que podrían servir como precondiciones para un operador. Note que un conjunto de precondiciones de un operador, es equivalente a una fórmula con conjunciones de esas precondiciones.

Para no tener que crear nuevos operadores para la misma acción que contemplen cada conjunto de precondiciones, esta extensión admite que el conjunto de precondiciones se sustituya por cualquier fórmula lógica compuesta de conjunciones y disyunciones de literales. Para profundizar en los principales conceptos, algoritmos y extensiones de planeación clásica, se refiere al lector a [21].

2.4. Búsqueda en el espacio-plan

Como ya he mencionado, la planeación consiste básicamente en una búsqueda en grafo que de alguna manera contiene información sobre el estado del mundo, así como las operaciones disponibles y cómo lo afectan. En la planeación por búsqueda en el espacio-plan, el grafo tiene estructura de árbol, cada nodo del árbol corresponde a una *especificación parcial de un plan*, y cada arco corresponde a una refinación de un plan. Este tipo de búsqueda sigue el *principio del menor compromiso (least commintment principle)*, esto significa que procurará no hacer refinaciones al plan (exploraciones en el árbol) que no contribuyan a la solución del problema.

Las definiciones presentadas en esta sección también fueron tomadas de [21].

Un plan parcial es un plan cuya ejecución llevaría al agente a un estado intermedio para cumplir el objetivo. A continuación se muestra la definición formal de un plan parcial para planeación por búsquedas en espacio-estado:

Definición 6. Un plan parcial es una tupla $\pi = (A, \prec, B, L)$, donde:

- $A = a_1, ..., a_k$ es un conjunto de operadores de planeación parcialmente instanciados (contienen variables libres).
- $\prec$ es un conjunto de restricciones de ordenamiento en A de la forma $(a_i \prec a_j)$, que indica que a_i debe ejecutarse antes que a_k.
- B es un conjunto de restricciones de unificación para las variables de las acciones en A de la forma $x = y$, $x \neq y$, $x \in D_x$, donde D_x es un subconjunto del dominio de x, etc.
- L es un conjunto de relaciones causales de la forma $(a_i \xrightarrow{P} a_j)$, tal que a_i y a_j son acciones en A, la restricción $(a_i \prec a_j)$ está en $\prec$, el predicado p es un efecto de a_i y una precondición de a_j, y las restricciones de unificación para las variables que aparecen en p están en B.

En pocas palabras, un plan parcial definido de esta forma contiene las acciones que involucra, el orden parcial de dichas acciones, las relaciones causales entre ellas, y las relaciones de unificación que permitirían aterrizar los predicados involucrados en ellas.

La refinación de un plan consiste en una o más de las siguientes operaciones:

- Agregar una acción a A.
- Agregar una restricción de ordenamiento a $\prec$.
- Agregar una restricción de unificación a B.
- Agregar una relación causal a L.

La eliminación de un elemento de algún conjunto de un plan parcial equivale a regresar en la búsqueda a un plan que no lo incluya, que ya habría sido explorado, por lo que eso no genera arcos nuevos (no son operaciones de refinación). La búsqueda de la solución consiste entonces en una búsqueda del camino desde un nodo con un plan inicial π_0 a un nodo que corresponde con un plan reconocido como solución.

Puesto que el plan contiene un conjunto de acciones a realizar, y cada acción contiene precondiciones para ser aplicables, se puede considerar a las precondiciones de una acción, como un subobjetivo que hay que cumplir; así, se puede incluir el estado objetivo como una acción falsa, llamada a_∞, que tiene al estado meta s_g como precondiciones, y al conjunto vacío como efectos. De la misma manera, el estado inicial se representa por un acción falsa a_0 que no tiene precondiciones, y que tiene como efectos, los predicados del estado inicial. El plan inicial π_0 se define entonces con $A_0 = \{a_0, a_\infty\}$, $\prec_0 = \{(a_0 \prec a_\infty)\}$ y $B = L = \emptyset$.

Un conjunto de restricciones de ordenamiento parcial $\prec$ es consistente si no existen ciclos en sus restricciones. Denotaré como c-l a la parte izquierda de una restricción c, y c-r a la parte derecha. Para que no existan ciclos, se tiene que cumplir que por cada restricción de ordenamiento $c_1 = (a_i \prec a_j)$ no exista una restricción $c_2 = (a_j \prec a_i)$ ni una restricción $c_3 = (a_j \prec a_k)$ tal que exista un conjunto de restricciones $c_1, ..., c_n$ donde c_1-$l = a_k$, c_n-$r = a_i$ y por cada c_p donde $1 < p \leq n$, c_p-$l = c_{p-1}$-r. Es decir, tomando en cuenta que la relación de orden parcial es transitiva (si $a_i \prec a_j$ y $a_j \prec a_k \Rightarrow a_i \prec a_k$), por cada restricción de ordenamiento explícita o implícita por transitividad $(a_i \prec a_j)$, no puede existir una restricción de ordenamiento $(a_j \prec a_i)$.

Un conjunto de restricciones de unificación B es consistente si cada asociación de una variable x con un valor permitido del dominio D_x es consistente con las restricciones restantes. Dos restricciones son consistentes si no asignan valores distintos del dominio D_x a una misma variable x, y ninguna impide que la variable se asocie con el valor con el que la otra restricción la asocia.

Definición 7. Un plan parcial $\pi = (A, \prec, B, L)$ es una solución al problema de planeación $P = (\Sigma, s_0, g)$ si:

- Sus restricciones de ordenamiento y unificación son consistentes; y
- cada secuencia de acciones totalmente ordenadas y totalmente instanciadas que satisfaga a $\prec$ y B es una secuencia que define un camino en un sistema de estados y transiciones Σ desde el estado inicial s_0 a un estado que contenga todos los predicados en g.

Sin embargo, no es eficiente verificar todas las secuencias de acciones totalmente instanciadas y ordenadas de A. Se necesita otra forma de verificar que un plan sea una solución a un problema de planeación; particularmente, se necesitan cumplir dos cosas: las precondiciones de cada acción en el plan sean producidas por alguna acción anterior, y que no existan acciones que creen conflictos en las relaciones causales, i. e. que no existan acciones intermedias que eliminen precondiciones de las acciones posteriores.

Definición 8. Una acción a_k en un plan π es una *amenaza* en una relación causal ($a_i \xrightarrow{P} a_j$) si y sólo si:

- a_k tiene un efecto $\neg q$ que sea posiblemente inconsistente con p, i. e. q y p son unificables;
- las restricciones de ordenamiento $(a_i \prec a_k)$ y $(a_k \prec a_j)$ son consistentes con $\prec$; y
- las restricciones de unificación que involucren a variables de q y p son consistentes con B.

Definición 9. Una *falla* en un plan $\pi = (A, \prec, B, L)$ es alguno de estos casos:

- existe una precondición de una acción en A (un subobjetivo) sin una relación causal que la produzca; o
- existe una amenaza, i. e., una acción que podría interferir con una relación causal.

Podemos entonces redefinir la solución a un plan de la siguiente manera:

Definición 10. Un plan parcial $\pi = (A, \prec, B, L)$ es una solución a un problema de planeación $\mathcal{P} = (\Sigma, s_0, g)$ si π no tiene *fallas* y los conjuntos de restricciones de ordenamiento $\prec$ y de restricciones de unificación B son consistentes.

2.5. Red de Tareas Jerárquicas

La planeación por búsqueda en espacio-plan es un formalismo muy útil para representar distintos tipos de problemas, sin embargo, para que el plan resultante sea efectivo, es necesario proporcionar una especificación precisa y adecuada de los operadores (acciones) involucrados en el sistema, lo cual puede ser complicado para tareas muy complejas, e incluso existen objetivos para los cuales la planeación clásica no puede generar un plan adecuado, por ejemplo, piense en un objetivo en que un agente deba realizar exactamente dos viajes redondos, o que realice una actividad tres veces más que otra. En general, el estado final es el mismo, por lo que no podría generar un plan.

Dicho de otra forma, puede haber tareas en que se requiera una secuencia de pasos que no tienen una relación explícita, de modo que una acción no produce las precondiciones de otra, por ejemplo el plan: primero contesta una pregunta, después ve a la cocina. En este ejemplo una tarea no representa ningún tipo de dependencia con respecto a la otra, sin embargo se pretende que un agente realice esas acciones, y que las realice en ese orden en particular. Dada una especificación *adecuada* de las acciones y condiciones del sistema, con planeación por búsqueda en espacio-plan se puede encontrar un plan que resuelva dependencias para lograr un objetivo, pero será necesario incluir explícitamente algún otro tipo de abstracción para generar planes más complejos.

Si se extiende el lenguaje con el que se construyen los operadores de la planeación clásica, para incluir predicados y símbolos de función que realicen operaciones de aritmética de enteros, se podrían considerar algunos planes más complejos, pero en general el conjunto de planes que puede generar un planeador por búsqueda en espacio-plan es más reducido del que puede generar un planeador de Redes Jerárquicas de Tareas.

La planeación con **Redes Jerárquicas de Tareas** (*Hierarchical Task Networks*, **HTN**) utiliza *tareas primitivas* y *tareas no-primitivas* de distintas jeraquías, en lugar de utilizar directamente acciones, y consiste en una técnica donde cada tarea de un plan inicial, de tamaño relativamente pequeño, puede sustituirse recursivamente por una red de tareas cada vez de menor jerarquía hasta llegar a una red de acciones primitivas.

Las acciones primitivas aterrizadas y no aterrizadas equivalen a las acciones y operadores de la planeación clásica, respectivamente. Para las tareas no primitivas existen *métodos*, que son los que describen cómo se llevará a cabo la descomposición de tareas, y pueden existir disitintos métodos para descomponer de maneras distintas una misma tarea, dependiendo del estado actual del mundo. Este esquema de planeación permite elevar el nivel de abstracción y embeber conocimiento sobre el problema en los métodos de decomposición de tareas, de manera que es más intuitivo y más fácil diseñar un plan.

A continuación se presentan varias definiciones de conceptos útiles para la construcción de un plan con HTN. Las definiciones presentadas en esta sección también fueron tomadas de [21].

Definición 11. Una *red de tareas* es un par $w = (U, C)$, donde U es un conjunto de tareas y C es un conjunto de restricciones.

La planeación con *HTN* generaliza los conceptos de precondiciones y efectos de las acciones como restricciones en los métodos. Cada restricción especifica un requerimiento que debe satisfacerse en cada plan que sea una solución al problema de planeación. Las restricciones que se pueden utilizar son:

Restricción precedencia. Se denota por la expresión $u \prec v$, donde u y v son tareas primitivas o no primitivas, y quiere decir que la última acción de la tarea u debe ejecutarse antes de la primera acción de la tarea v.

Restricción antes. Se denota con la expresión $before(U', l)$, donde $U' \subseteq U$ es un conjunto de tareas y l es una literal. Especifica que la literal l debe ser cierta en el estado inmediato anterior a la primer tarea en U'.

Restricción después. Se denota por la expresión $after(U', l)$. Análoga a la *restricción antes*, la literal l debe ser cierta en el estado inmediato posterior a la ejecución de la última tarea en U'.

Restricción entre. Tiene la forma $between(U', U'', l)$. Indica que la literal l debe ser verdadera en el estado después de la última tarea en U', en el estado anterior a la primera tarea en U'', y en todos los estados intermedios.

Definición 12. Un *método* de HTN es una tupla $m = (nombre(m), tarea(m), subtareas(m), restricciones(m))$, donde:

- $nombre(m)$ es una expresión de la forma $n(x_1, ..., x_k)$, donde n es un símbolo de método único, y $x_1, ..., x_k$ son todos los símbolos de variables que ocurren en m.
- $tarea(m)$ es una tarea no primitiva.
- (subtareas(m), restricciones(m)) es una red de tareas.

Definición 13. La *descomposición* de una red de tareas se realiza de la siguiente forma: Sea $w = (U, C)$ una red de tareas, $u \in U$ una tarea, m una instancia de un método y $tarea(m) = u$. Entonces *m descompone a u* produciendo la red de tareas w' de la siguiente manera:

$$w' = \delta(w, u, m) = ((U - \{u\}) \cup subtareas(m), C' \cup restricciones(m))$$

donde C' es una versión modificada de C de la siguiente manera:

- Cada restricción precedencia que contenga a u, se reemplaza con restricciones precedencia que contengan a las tareas en $subtareas(m)$ en su lugar. Por ejemplo, si $subtareas(m) = \{u_1, u_2\}$, se reemplazaría la tarea $u \prec v$ con las restricciones $u_1 \prec v$ y $u_2 \prec v$.
- En cada restricción "antes", "después" o "entre" en que exista un conjunto de tareas U' que contenga a u, se reemplaza U' por $(U' - \{u\}) \cup subtareas(m)$. Por ejemplo, si $subtareas(m) = \{u_1, u_2\}$, entonces se reemplazaría la restricción $before(\{u, v\}, l)$ por la restricción $before(\{u_1, u_2, v\}, l)$.

Definición 14. Un *dominio de planeación HTN* es un par:

$$\mathcal{D} = (O, M),$$

y un *problema de planeación HTN* es una tupla:

$$\mathcal{P} = (s_0, w, O, M),$$

donde s_0 es el estado inicial, w es la red de tareas inicial, O es un conjunto de operadores, y M es un conjunto de métodos de HTN.

Definición 15. Si $w = (U, C)$ es primitivo (contiene solamente tareas primitivas), entonces un plan $\pi =< a_1, ..., a_k >$ es una *solución* para un problema $\mathcal{P}$ si existe una instancia aterrizada (U', C') de (U, C) y un ordenamiento de las tareas en U' tal que todas las siguientes condiciones se cumplen:

- Las acciones en π son las tareas $u_1, u_2, ..., u_k$, i. e. $nombre(a_i) = u_i$ para $i = 1, ..., k$.
- El plan π es ejecutable en el estado s_0.
- El ordenamiento satisface las restricciones precedencia en C'. Es decir que no hay ciclos como se definió en la planeación espacio-plan.
- Por cada restricción $before(U', l)$ en C', l es cierta en el estado s_{i-1} que precede inmediatamente a la acción a_i, donde a_i es la primera acción en las tareas de U'.
- Por cada restricción $after(U', l)$ en C', l es cierta en el estado s_j producido por la acción a_j, donde a_j es la última acción de las tareas en U'.
- Por cada restricción $between(U', U'', l)$ en C', l es cierta en cada estado entre las acciones a_i y a_j, donde a_i es la última acción de las tareas en U', y a_j es la primera acción de las tareas en U''.

Si $w = (U, C)$ no es primitiva (contiene al menos una tarea no primitiva), entonces π es una *solución* para $\mathcal{P}$ si existe una secuencia de descomposición de tareas que pueda ser aplicada a w para producir una red de tareas primitiva w' tal que π es una solución para w'.

Para profundizar en los principales conceptos y algoritmos de HTN, refiero al lector a [5] y [21].

2.6. Redes de Petri

Las *redes de Petri* son un formalismo que consiste en un lenguaje gráfico para modelar sistemas discretos, así como procesos concurrentes, paralelos y no deterministas. Se pueden realizar distintos tipos de análisis en los modelos de redes de Petri, que proveen información útil acerca de la ejecución y la relación entre los nodos.

Una red de Petri es un grafo dirigido bipartito con dos tipos de nodos: *lugares* (places) y *transiciones* (transitions). Los lugares se conectan a las transiciones y vice versa. Cada lugar tiene cero o más *tokens* (o marcas, aunque prefiero utilizar el término en inglés), y la configuración de tokens en una red de Petri, llamado *marcado* (marking), define un estado del sistema. En general, los lugares representan diferentes componentes del sistema modelado, como procesos, hechos o variables de algún tipo. La presencia de tokens en ellos se interpreta también de acuerdo al sistema que se esté modelando, pudiendo significar que un proceso marcasestá activo, que un hecho es verdadero, o que una condición o recurso particular está presente. Las transiciones sirven para modelar los cambios que suceden en el sistema, a partir de las relaciones entre sus elementos.

Una transición está habilitada cuando todos sus lugares de entrada (conectados a ella) tienen la cantidad necesaria de tokens. La ejecución de una transición consume tokens de sus lugares de entrada y produce tokens en sus lugares de salida (a los que la transición está conectada). Los tokens que son consumidos/producidos corresponden con el número especificado por los pesos de cada arco del grafo. En algunos casos se puede considerar un arco especial para los lugares de entrada, llamado *inhibidor*, que indica que la transición está habilitada solamente cuando el lugar que apunta a ella no tiene ningún token (y los demás lugares de entrada también permiten que esté habilitada). En algunas fuentes, a las redes de Petri que incluyen este tipo de arcos se les llama Redes de Petri Extendidas.

La ejecución de una red de Petri consiste en la ejecución no determinsta de las transiciones que vayan siendo activadas hasta que no se pueda ejecutar ninguna otra transición (si acaso se da el caso). En la representación gráfica del grafo de una red de Petri, los lugares se representan con círculos sin rellenar, las transiciones con rectángulos, los arcos con flechas y los tokens con círculos negros pequeños, dentro de los lugares. La figura 2.6 muestra una red de Petri sencilla con sus elementos.

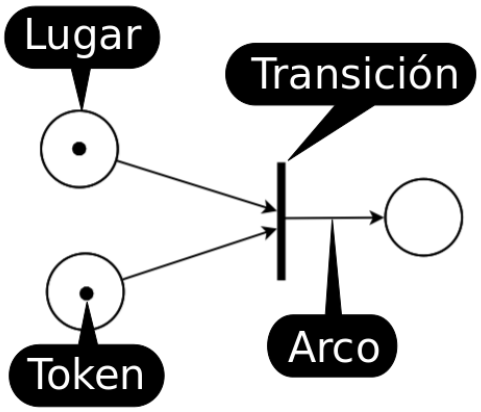


Figura 2-4: Ejemplo de una red de Petri sencilla.

Note que el lugar de salida de la transición mostrada podría conectarse a otras transiciones, produciendo una ejecución en cadena.

Definición 16. La definición formal de una red de Petri es una tupla $PN = (P, T, F, W, M_0)$ donde:

- $P = \{p_1, p_2, p_3, ..., p_m\}$ es un conjunto finito de lugares,
- $T = \{t_1, t_2, t_3, ..., t_n\}$ es un conjunto finito de transiciones,
- $F \subseteq (P \times T) \cup (T \times P)$ es un conjunto de arcos (relación de flujo),
- $W : F \rightarrow \{1, 2, 3, ...\}$ es una función de pesos,
- $M_0 : P \rightarrow \{0, 1, 2, 3, ...\}$ es el marcado inicial,
- $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$

En algunas referencias en la literatura, el conjunto de arcos se divide en dos conjuntos: arcos de entrada (I) y arcos de salida (O). Muchas veces los nodos del grafo (lugares y transiciones) pueden estar etiquetados para hacer referencia a algún proceso o recurso de alguna naturaleza, y los arcos pueden estar etiquetados con pesos para indicar que más de un token es necesario para activar la transición.

El ejemplo mostrado es de la clase de redes de Petri de lugares y transiciones (*Place/Transition Petri Nets*, o *P/T PN*), y es el modelo más básico que existe. Las redes de Petri son una herramienta de modelado poderosa, sin embargo existen muchos casos donde se necesitan algunas modificaciones o restricciones. Se han propuesto distintas clases y extensiones a los modelos de redes de Petri base para modelar sistemas con diferentes propiedades, uno de ellos es el de las redes de Petri es el de las redes de Petri de alto nivel.

2.6.1. Redes de Petri de alto nivel

Las redes de Petri de alto nivel (*High Level Petri Nets*, HLPN) incluyen a varias clases, como las redes de Petri de Predicado/Transición (*Predicate/Transition Petri Nets* o *Pr/T PN*, diferente de *P/T PN*) y las redes de Petri con colores (*Colored Petri Nets*). Agregan expresividad a las redes de Petri al considerar distintos "tipos" de tokens para asociar información adicional a los lugares. El poder de modelado es el mismo que el de los modelos de P/T PN (i. e. un modelo de HLPN puede traducirse en un modelo de P/T PN), pero los diagramas equivalentes suelen ser mucho menores.

Las Pr/T PN representan un salto con respecto a las P/T PN, análogo a lo que la lógica de primer orden es para la lógica proposicional. En este tipo de redes de Petri, los lugares reciben extensiones de tokens que representan una tupla de valores particulares; es decir, en lugar de que un lugar represente sólo un elemento y la presencia de un token indique si está presente, los tokens de las redes de Petri de alto nivel incluyen información adicional, particular para cada uno de ellos.

A este tipo de tokens o tuplas se les llama elementos (*items*, o también individuos), de modo que los lugares pueden contener un multiconjunto de tuplas (i. e. puede haber más de un elemento idéntico en el mismo lugar), y las etiquetas de los arcos indican una suma formal de elementos que deberán existir en los lugares de entrada para que las transiciones sean activadas, o bien, que serán producidas en los lugares de salida. Las transiciones podrán además contener restricciones expresadas con operaciones y relaciones de los elementos involucrados en esa transición, por ejemplo, comparaciones de igualdad entre los campos de los elementos.

En esta clase de redes de Petri, los lugares se llaman predicados, porque los elementos que fluyen en ellos son como predicados de lógica de primer orden. Cada valor de la tupla corresponde con un campo de una relación de primer orden, y las restricciones especificadas en las transiciones, que restringirían su activación, corresponde con restricciones de unificación de los predicados.

Definición 17. Una *Pr/T PN* consiste en las siguientes partes:

(1) Una red dirigida (S, T, F) donde:

- S es el conjunto de predicados (lugares);
- T es el conjunto de transiciones; y
- F es la relación de flujo del grafo (arcos y pesos).

(2) Una estructura Σ que consiste en algunos tipos de elementos con sus operaciones y relaciones.

(3) Un etiquetado de los arcos con (una expresión denotando) una suma formal de tuplas de variables. El "elemento zero", indicando predicados sin argumentos (lugares normales de redes de Petri) se denota por un símbolo especial ϵ.

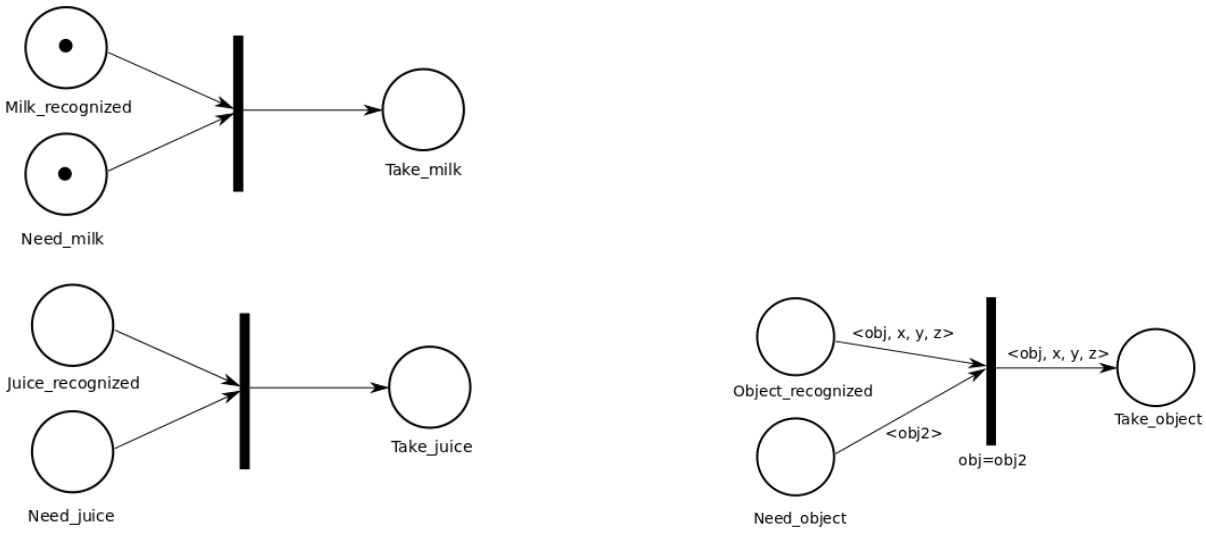
(4) Inscripciones en algunas transiciones que sea una fórmula lógica construida con las operaciones y relaciones de la estructura Σ. Las variables libres que aparezcan en una fórmula deben aparecer en alguno de los arcos adyacentes.

(5) Un marcado M_0 de predicados en S con sumas formales de tuplas, llamadas también elementos.

(6) Una función K que asigna a los predicados un límite superior para el número de copias del mismo elemento que pueden tener. Se puede llamar a $K(s)$ la capacidad de s.

La figura 2-5 muestra el ejemplo de una red de Petri de lugares y transiciones que realiza la acción de tomar un objeto, y su contraparte en una red de Petri de predicados y transiciones. Considere que la primera red crecería demasiado conforme se agreguen objetos en el sistema, además de que el procedimiento de tomar el objeto debe de alguna manera obtener información adicional que necesita.

Para más información sobre redes de Petri, el artículo [22] incluye una muy completa introducción, además de muchas referencias para leer más sobre el tema. Para más información sobre redes de Petri de Predicado/Transición, consulte [23] y [24].



(a) P/T PN de toma de objetos.

(b) Pr/T PN de toma de objetos.

Figura 2-5: Comparativa de redes de Petri.

CAPÍTULO

3

EL PLANEADOR DE ACCIONES

El motor de planeación desarrollado para esta tesis está construido sobre el lenguaje de programación CLIPS, utilizado para diseñar sistemas expertos basados en reglas, y se comunica con la herramienta de paso de mensajes y concentradora de variables compartidas `BlackBoard`, desarrollada por Mauricio Matamoros en este mismo laboratorio [19]. Para la implementación del sistema fue necesaria la librería pyclips, desarrollada por un tercero, que permite embeber un intérprete de CLIPS en un programa de python; y el desarrollo de las herramientas presentadas en los apéndices B y C de esta tesis.

PyRobotics (apéndice B) es una API para el lenguaje python que sirve para construir módulos que se conecten al BlackBoard, de modo que pueda enviar y recibir comandos, y publicar y suscribirse a variables compartidas. Esta herramienta no sólo se utilizó para el desarrollo de esta tesis, sino que ya está siendo utilizada por otros miembros del laboratorio, así como estudiantes que asisten a las clases impartidas por los miembros del mismo. En la competencia RoCKIn@HOME se utilizó para conectar módulos diseñados para funcionar con BlackBoard con módulos de ROS.

BBCLIPS (apéndice C) es un módulo de BlackBoard escrito en python que utiliza la pyRobotics para conectarse a él, y la librería pyclips para utilizar un intérprete de CLIPS. Incluye reglas y funciones de CLIPS para manejar los comandos y variables compartidas del BlackBoard.

Ambas herramientas fueron desarrolladas como parte de la realización de esta tesis. Para más información, consulte los apéndices mencionados, donde podrá encontrar una descripción más detallada, así como ligas a una documentación más extensa y al código fuente. La figura 3-1 muestra un diagrama del *stack* de herramientas utilizadas para la implementación de este sistema.

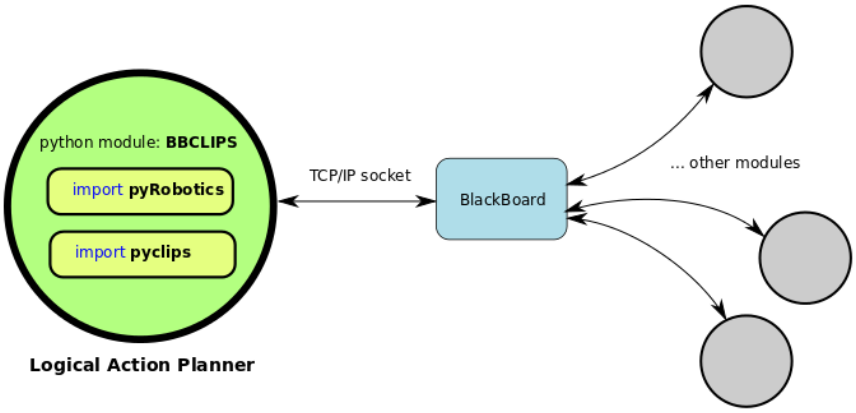


Figura 3-1: *Stack* de herramientas utilizadas en la implementación de este planeador.

Uno de los propósitos de este trabajo es que los planes se especifiquen de una manera menos procedural que en otros esquemas comúnmente utilizados, como las máquinas de estado, sino que sólo representen objetivos de distintos niveles jerárquicos que habrá que cumplir, independientemente del procedimiento utilizado para lograrlos. Al final, el plan completo consistiría de manera similar en una secuencia (sólo que parcialmente ordenada) de acciones, pero que no está embebida en la especificación del plan, sino que se genera a través de la descomposición del mismo.

Así, una tarea de alto nivel (objetivo) debería descomponerse recursivamente en tareas de menor jerarquía, hasta llegar a tareas primitivas (acciones), dependiendo del contexto de ejecución. Y la descomposición o ejecución de las tareas estaría desacoplada o sería independiente de las demás tareas incluidas en el plan, y dependerían de la situación actual que perciba el robot, representada por los hechos en las precondiciones de las reglas.

Para ilustrar la diferencia, considere el ejemplo donde el objetivo de un robot de servicio es traer una bebida a una persona que se la pidió. Una máquina de estados jérarquica o alguna especificación equivalente puede existir donde el robot primero pregunta dónde está la bebida, después va a ese lugar, la busca, la toma y regresa a donde está la persona. Sin embargo, este ejemplo asume que la tarea inicia con unas condiciones particulares, que son: que no conoce la ubicación de la bebida, y que no se encuentra en su posesión (en alguno de sus manipuladores).

Si estas condiciones fueran diferentes, habría que utilizar una máquina de estados distinta que las considere, o de alguna manera hacer que las contemple en un estado "cero" (anterior al estado inicial de esa tarea), para decidir a dónde hacer la transición; por ejemplo, el robot podría conocer previamente la ubicación habitual de las bebidas, o podría tener la bebida en su manipulador, como resultado de una tarea anterior que se le haya ordenado.

Especificando las tareas como objetivos, en donde la descomposición depende de las condiciones actuales del sistema, se puede especificar un objetivo de alto nivel que sea "traer una bebida". La descomposición podría tomar en cuenta distintas circunstancias, como por ejemplo: si se conoce la ubicación de la misma, para descomponerlo en un plan o en otro, que contemple o no la adquisición de esa información. El robot podría incluso preguntar dónde está la bebida y al no obtener una respuesta satisfactoria, generar un plan para buscarla en distintas ubicaciones; siempre dependiendo del contexto de ejecución actual, y no de la especificación inicial que simplemente hacía referencia a un objetivo.

Un sistema experto basado en reglas propicia o facilita el diseño de planes que dependa de la situación actual, y no del contexto de ejecución de la tarea dentro de un plan; es decir, independientemente de qué tareas la precedieron, o qué tareas están planeadas para ejecutarse después.

El trabajo aquí desarrollado consiste tanto en un planeador como en un ejecutor, de modo que tomando en cuenta las situaciones que se presenten durante la ejecución se logre una planeación dinámica que sea más efectiva que un plan prediseñado. En este trabajo la eficiencia computacional no es una de las principales preocupaciones, sino la expresividad de los planes que ejecuta. La velocidad de las computadoras comunes actuales es suficiente para manejar los planes que realizamos en el laboratorio.

Este capítulo se divide en tres secciones: la primera describe el modelo conceptual de planeación, la segunda describe la sintaxis y la semántica de la especificación de los planes; finalmente, la tercera presenta el algoritmo de planeación y ejecución.

3.1. Modelo de planeación

El modelo de planeación utilizado mezcla conceptos presentes en la planeación por búsqueda espacio-plan y HTN. El resultado es un modelo similar a HTN y máquinas de estado jerárquicas, pero que favorece a una filosofía de diseño de los planes un tanto distinta a otras aproximaciones utilizadas.

Por una parte, el enfoque de solución de dependencias de tareas propuesto en la planeación por búqueda en espacio-plan propicia una planeación orientada a objetivos, donde lo que es importante es el *qué se quiere lograr*, mientras que el *qué se necesita para lograrlo* se determina a través de la planeación (búsqueda). Es decir, se utilizaría la planeación para ubicar al robot en una situación en que es posible y tiene sentido realizar dicha tarea.

Por otra parte, en ocasiones no existe una relación implícita de dependencia o precedencia que relacionen dos tareas, sino que su relación se hace explícita a través de un plan de alto nivel, diseñado por una persona; o bien, aunque en realidad sí exista una realción de este tipo, resulta difícil construir un modelo del mundo tan completo que el sistema pueda inferir todas las condiciones que deban existir para la realización de una tarea. Para esto es útil la manera en que el enfoque de HTN permite embeber conocimiento sobre la solución de una tarea en la descomposición de la misma en otras tareas de menor jerarquía.

El modelo resultante se podría definir como una **Búsqueda Jerárquica en el Espacio-Plan**, donde puede existir una red de tareas inicial o producto de una descomposición anterior, y se realiza una búsqueda local por cada tarea para la solución de dependencias o precondiciones necesarias antes de su ejecución, o bien, para descomponerla recursivamente. La figura 3-2 muestra un diagrama que describe este modelo de planeación donde cada nodo contiene un árbol de búsqueda, representando la búsqueda local para la solución de dependencias.

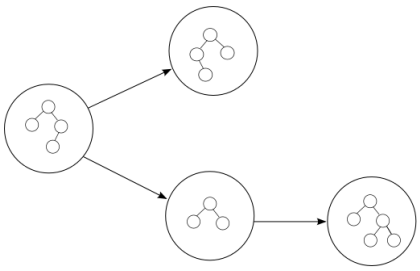


Figura 3-2: Esquema del modelo de Búsqueda Jerárquica en Espacio-Plan.

La filosofía de diseño de los planes pretendida es que en lugar de proponer inicialmente un procedimiento que se descompone en tareas de distinto nivel jerárquico, se proponga un objetivo de bajo nivel que recursivamente resuelva dependencias necesarias para realizar esa acción. La figura 3-3 muestra una comparación de dos planes equivalentes (realiza las mismas acciones para lograr el objetivo), donde el de la izquierda es un procedimiento que se descompone en otras tareas, más parecido a la propuesta original de HTN donde la tarea original se sustituye por las tareas hijas; mientras que el de la derecha, más apegado a la propuesta en esta tesis, lleva a cabo la planeación para resolver dependencias antes de realizar cada tarea, después de lo cual el control de ejecución regresaría a la tarea padre para que sea ejecutada.

```
-> Traer objeto a persona.
    '-> Obtener objeto.
        '-> Preguntar dónde está el objeto.
        '-> Ir a donde está el objeto.
        '-> Buscar objeto con visión.
        '-> Tomar el objeto.
    '-> Regresar a la ubicación de la persona.
    '-> Entregar objeto a persona.
```

```
-> Entregar objeto a persona.
    '-> Obtener objeto.
        '-> Tomar objeto.
            '-> Estar frente a objeto.
                '-> Moverse a ubicación de objeto.
                    '-> Averiguar dónde está el objeto.
                        '-> Preguntar dónde está el objeto.
                    '-> Ir a donde está el objeto.
                '-> Buscar objeto con visión.
    '-> Estar frente a la persona.
        '-> Ir a donde está la persona.
```

Figura 3-3: Comparación de dos planes equivalentes con una filosofía de planeación distinta.

Podrá notar que la descomposición de la alternativa de la derecha parece un tanto más compleja y un poco más larga. Sin embargo, la practicidad de este enfoque radica en el poder considerar escenarios diferentes en que sea necesario cumplir esos objetivos, tales como que ya se conoce la ubicación del objeto, o ya se tiene el objeto en uno de los manipuladores. La descomposición mostrada correspondería con el resultado de la búsqueda en el espacio-plan para la realización de esa tarea, según un escenario determinado.

Para lograr esto con el primer enfoque, tendrían que existir diferentes maneras de descomponer el plan, considerando cada situación, lo cual correspondería con un número de planes exponencialmente creciente conforme se incrementen las condiciones necesarias para seleccionar un plan u otro. Más aún, al no comprometerse con una descomposición particular del plan, sería más sencillo (re)planear cuando se presenten situaciones inesperadas.

3.2. Especificación de los planes

En la sección 2.2.1 se presentaron algunos criterios para clasificar el dominio de los problemas de planeación. En este sentido, se pueden aumentar algunos criterios que determinan el espacio de planes que puede generar un planeador:

Ejecución de las acciones Una solución a un problema de planeación puede ser una secuencia lineal de acciones síncronas, es decir que cada acción se ejecuta hasta que la anterior haya terminado; o bien, una secuencia de conjuntos de acciones, donde cada conjunto de acciones se ejecuta después del conjunto anterior, pero las acciones de cada conjunto pueden ejecutarse de manera asíncrona.

Número de agentes El plan generado puede coordinar las acciones de un sólo agente, o de múltiples agentes.

Número de tareas Cuando se quiere que el agente realice más de una actividad, un planeador podría contemplar planes para actividades concurrentes, donde dos o más tareas se ejecutan concurrentemente; o planes multi-tarea, en que una tarea se suspende mientras otra se ejecuta. Esta característica está relacionada con las capacidades de replaneación de un planeador para incorporar la ejecución de nuevas tareas una vez que hay un plan formado o en ejecución.

Utilizando los criterios presentados aquí y en la sección 2.2.1, el motor de planeación aquí desarrollado contempla las siguientes propiedades (el símbolo Σ representa el modelo discreto del sistema):

- Σ finito.
- Σ parcialmente observable.
- Σ no determinista.
- Σ dinámico.
- Objetivos extendidos.
- Planes parcialmente ordenados.
- Tiempo implícito.
- Planeación “online”.
- Ejecución asíncrona de acciones.
- Un sólo agente.
- Multitarea.

La implementación del motor de planeación propiamente, consiste en una serie de reglas de CLIPS que sirven para controlar la ejecución de un plan estructurado a partir de hechos. El control del flujo de ejecución se realiza con hechos "banderas", así como con las prioridades (*salience*) de las reglas. Algunas reglas del motor de planeación utilizan salience entre -9501 y -10000, o entre 9501 y 10000. Esto significa que el usuario (diseñador de los planes) tiene disponible el rango de -9500 a 9500 para el diseño de los planes.

La especificación de los planes para este motor de planeación se logra utilizando hechos estructurados de CLIPS, con campos que contienen tanto la información del objetivo al que hacen referencia, como su ubicación en el contexto de ejecución de los planes. Las precondiciones de las reglas de CLIPS usualmente son hechos que modelan el estado actual del entorno, sin embargo en este trabajo no se hace ninguna suposición sobre la estructura ni el contenido de la base de conocimiento (hechos de CLIPS) que se utilice para el diseño de los planes, aparte de los hechos aquí presentados que sirven para detectar el estado de ejecución de un plan.

La figura 3-4 muestra el `deftemplate` *task* utilizado para la representación de las tareas. El ordenamiento y vinculación de estas tareas generan un plan. Las tareas, que representan objetivos de distinto nivel jerárquico, pueden pertenecer a diferentes planes, especificados por el campo `plan`.

```
(deftemplate task
        (slot id
                (default-dynamic (gensym*))
        )
    (slot plan
        (type LEXEME)        ; STRING or SYMBOL
        (default ?NONE)
    )
    (slot action_type
        (type SYMBOL)
        (default ?NONE)
    )
    (multislot params
        (default "")
    )
    (multislot step
        (type INTEGER)
        (default 0)
    )
    (slot parent  (default nil)  )
)
```

Figura 3-4: `deftemplate` de una tarea.

El campo `id` sirve para identificar ese hecho, pues más adelante se expondrá que existen hechos relacionados a esa tarea, como los que indican si una tarea está activa, o cuál es el estado final de su ejecución.

El campo `step` contiene el orden parcial de las tareas con respecto al plan al que pertenecen, y es un campo multi-lugar (*multislot*) que contiene números enteros. El nivel jerárquico de una tarea corresponde con la longitud de este campo, mientras que el orden relativo entre tareas de la misma jerarquía corresponde con el orden natural de los números que se encuentren en la posición jerárquica más baja (hasta la izquierda, vea la figura 3-5). Así, las tareas de más alto nivel inician con sólo un número entero, indicando el orden en que se deben ejecutar (o descomponer).

El campo `parent` almacena el `id` correspondiente con la tarea que produjo a esta tarea en el proceso de descomposición de los planes (los planes de más alto nivel tienen el símbolo `nil` como valor en este campo). El campo `action_type` hace referencia a una tarea particular, se puede considerar el "nombre" de la tarea. El campo `params` es un campo multi-lugar que incluye información necesaria para esa tarea; puede utilizarse con dos propósitos: especificar parámetros que necesitará para la ejecución, o diferenciar dos formas de resolución de una tarea con un mismo *action_ type*, en cuyo caso, la información necesaria se obtendría de la base de conocimiento representada en otros hechos.

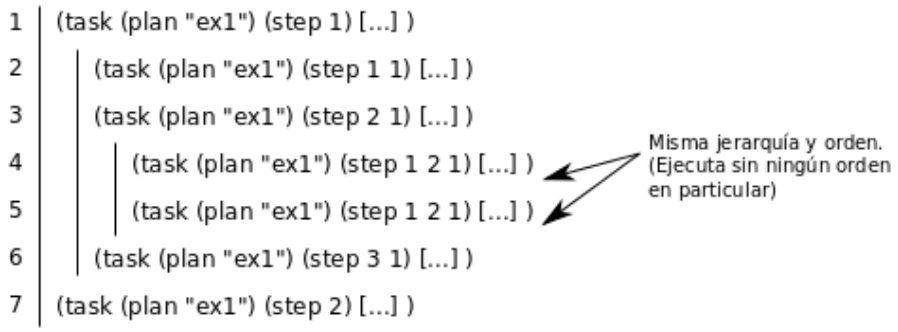


Figura 3-5: Estructura de un plan al descomponerse en tareas de menor jerarquía.

Para establecer una similitud con el paradigma de planeación por búsqueda en espacio-plan, el plan mostrado en la figura 3-5 se puede representar como un plan de este tipo interpretando las tareas de menor jerarquía como acciones necesarias para cumplir los prerrequisitos de las tareas de mayor jerarquía, y el orden parcial estaría determinado en primer lugar por la jerarquía, y en segundo lugar por el orden natural del número de menor jerarquía (hasta la izquierda).

Las restricciones de unificación, así como las relaciones causales estarían embebidas en las reglas de descomposición y ejecución, de modo que una tarea que se descompone en tareas de menor jerarquía tendría relaciones causales por cada una de las tareas en las que se descompone.

Tomando el ejemplo mostrado en la figura 3-5, se podría considerar un plan más o menos equivalente, generado con planeación por búsqueda en espacio plan de acuerdo a la definición presentada en la sección 2.4, similar al mostrado a continuación, donde las tareas se representan por el número de línea mostrado en la figura anterior.

$\pi = (\{1, 2, 3, 4, 5, 6, 7\},$

$\{(1 \prec 7), (2 \prec 3), (3 \prec 6), (6 \prec 1), (4 \prec 3), (5 \prec 3)\},$

$< B >,$; por simplicidad se omite este conjunto, consulte la sección de planeación espacio-plan

$\{(2 \rightarrow 1), (3 \rightarrow 1), (6 \rightarrow 1), (4 \rightarrow 3), (5 \rightarrow 3)\})$

Así mismo, es posible notar algunas similitudes con la planeación por redes jerárquicas de tareas (HTN): las tareas de alto nivel se descomponen en tareas de menor jerarquía, de modo que un plan de alto nivel se expande a un plan con estructura de árbol que podría incluir varios niveles de profundidad. La diferencia, sin embargo, es que en vez de que una tarea sea sustituida por una red de tareas de menor jerarquía, las tareas hijas son agregadas al plan y serían ejecutadas en un contexto embebido en la ejecución de la tarea de alto nivel, de modo que pueda regresar el control de ejecución a la tarea padre, junto con el resultado de la ejecución de las tareas hijas.

En este sentido, se podría comparar este trabajo con las redes de transiciones recursivas (RTN) [25], aunque se hace referencia a las HTN por la similitud que tienen las reglas de descomposición de tareas con los métodos de HTN. La implementación de esta forma permite considerar procesos de finalización y replanear cuando el resultado no sea el esperado (i. e. cuando las tareas hijas reportan un estado de ejecución fallido a las tareas padre).

Idealmente, una vez diseñados los planes de descomposición de tareas genéricas, y las reglas de acciones primitivas, un plan inicial para realizar una tarea podría especificarse simplemente creando como hechos iniciales, los hechos de tareas de mayor jerarquía, resultando en planes muy concisos y rápidos de diseñar. En el ejemplo mostrado en la figura 3-5, sería necesario solamente crear los hechos correspondiente a las dos tareas que están alineadas hasta la izquierda (nivel más alto de la jerarquía).

3.2.1. Esquema de prioridades de tareas

Como se mencionó anteriormente, este planeador es multitarea. La manera en que se pueden generar los planes de alto nivel puede ser por otras reglas, independientes de las reglas de planeación; hechos iniciales que se indiquen al programa; o bien, por comandos de otro módulo, por ejemplo: el procesador de lenguaje natural al identificar una orden de alguna persona.

El hecho de que la representación de los objetivos (tareas) se haga a través de hechos estructurados permite que se especifique más de un plan distinto en diferentes hechos de tareas, a través del campo `plan`. Sin embargo, esto conlleva más conflictos que resolver para el planeador, pues cuando exista más de un plan para ser ejecutado, tendrá que *decidir* cuál es el que más le conviene realizar.

Cada tipo de tarea (el `action_type` de una tarea) puede tener asociado un número, que representa la *prioridad asociada* de dicha tarea. Una tarea tiene prioridad sobre otra si su prioridad asociada es mayor que la de la otra. Esta prioridad se define con el hecho:

```
(task_priority <action_type> <priority>)
```

Estas prioridades se utilizan en el proceso de selección de tareas para activar las tareas más prioritarias. Se comparan las tareas por pares y se van descartando las menos prioritarias. Es posible que una tarea no tenga información de su prioridad, de hecho es probable que esto ocurra, pues la prioridad de las acciones primitivas o de bajo nivel muchas veces tendrá sentido no por sí mismas, sino por el contexto de ejecución del plan actual. Cuando una decisión no puede ser tomada al comparar dos tareas, ya sea por que alguna de las tareas no tiene una prioridad asociada, o por que sus prioridades son iguales, se utiliza la prioridad asociada de las tareas padres de manera recursiva hasta que se pueda tomar una decisión.

Es importante notar que las prioridades asociadas de las tareas son indepentientes del contexto de ejecución dentro del plan, i. e. si existe una tarea de alto nivel que tenga prioridad sobre una que pertenezca a otro plan, puede existir una tarea de menor jerarquía en el segundo plan, que tenga prioridad sobre la primera.

Esto es así por que a veces podría no ser conveniente interrumpir un proceso en ejecución para atender a otra tarea. Por ejemplo: si una persona llama al robot cuando está a punto de dejar un objeto en su lugar, probablemente sería útil ejecutar el nuevo plan de atender a la persona cuando tenga los manipuladores desocupados; en cuyo caso, con las prioridades asociadas adecuadas, el robot podría primero dejar el objeto en su lugar, y después atender a la persona, pues es una tarea más prioritaria que el plan que estaba ejecutando anteriormente.

Adicionalmente, es posible que más de una tarea se pueda ejecutar en paralelo. Para esto debe existir una declaración explícita de que dos tipos de tareas se pueden ejecutar en paralelo. Esto se logra con hechos de la forma:

```
(can_run_in_parallel <action_type1> <action_type2>)
```

El algoritmo de selección de tareas habilita las tareas de menor jerarquía de cada rama en cada plan como candidatas a ser activadas. La estructura de un plan tiene forma de árbol, entonces sólo los nodos hoja en el plan actual son candidatos a ser activados para su ejecución. El algoritmo de selección se describe con más a detalle en la siguiente sección.

3.2.2. Reglas de planeación

La descomposición y ejecución de los planes se especifica a través de reglas que contienen precondiciones y efectos particulares. Sintácticamente, se trata de reglas de CLIPS, pero semánticamente se pueden considerar tres tipos de reglas: descomposición/ejecución, procesos de finalización de tareas y procesos de cancelación de tareas. La diferenciación se hace a través de distintos hechos (en las precondiciones) que determinan el estado de la ejecución del plan.

El hecho de que la diferenciación sea semántica, además de lo que propone la filosofía de diseño pretendida con este planeador, permite que no haya una diferenciación estricta entre tareas primitivas y no primitivas, como se ha considerado en otros esquemas de planeación. Por el contrario, cuando se ejecuta una tarea de menor jerarquía, el control de ejecución regresa a la tarea padre, que podría generar una nueva tarea hija o ejecutar alguna "acción primitiva" a través de sus actuadores o sensores, o bien, simplemente marcarse como exitosa o fallida. Por esta razón se considera un sólo tipo de reglas para la descomposición y ejecución de las tareas.

También es importante mencionar que, dado que el control de ejecución regresa a la tarea padre, sería necesario que cada tarea incluya reglas que determinen explícitamente que la ejecución de la tarea ha sido exitosa o fallida, pues para el motor de planeación no hay forma de diferenciar cuando un plan ha sido mal diseñado y ya no existen reglas de ejecución habilitadas, de cuando de verdad ha fallado un plan y por lo tanto tampoco están habilitadas sus reglas (no sabe cómo manejar la situación actual en el entono), o bien, cuando la tarea ha hecho todo lo que debía y debería marcarse como exitosa.

Una **regla de descomposición/ejecución de tarea** (a las que me referiré como **dexec**) es una regla que maneja la descomposición de tareas y/o ejecución de acciones primitivas del robot. Debe incluir precondiciones que restrinjan su activación, dependiendo de la situación presente en el entorno; y efectos que creen nuevos hechos de tareas, especificando el orden relativo entre ellas y la jerarquía con respecto a su padre, o bien, hechos que reflejen la situación en el entorno después de la ejecución de alguna acción primitiva o la detección de algún evento. Las reglas de dexec deben además incluir como precondiciones:

(1) El hecho estructurado *task* que corresponda con esa tarea.

(2) El hecho `(active_task ?t)` , donde $?t$ es el id del hecho que representa esa tarea (vea la figura 3-4).

(3) La restricción de que no exista un hecho de `task_status` para esa tarea (i. e. no es una regla de finalización).

(4) La restricción de que no exista el hecho `(cancel_active_tasks)` (i. e. no es una regla de cancelación).

En otras palabras, es una regla que corresponde a una tarea que está activa y que no es de finalización ni de cancelación. Distintas reglas pueden existir para generar planes alternativos, y su activación dependería de la situación presente en el ambiente. La figura 3-6 muestra una plantilla para este tipo de reglas donde se generan tareas hijas, aunque por supuesto, no sería necesario en todas las reglas. Los campos entre llaves angulares ('<'y '>') deben ser sustituidos por elementos en concreto por el diseñador del plan. La figura 3-7 muestra un ejemplo de regla dexec donde se descompone una tarea en otras de menor jerarquía.

```
(defrule <rule_name>
    (task (id ?t) (plan ?planName) (action_type <action_type>)
            (params <params variables>) (step $?steps) )
    (active_task ?t)
    (not (task_status ?t ?) )
    (not (cancel_active_tasks) )
    <other_preconditions>
    =>
    (assert
        (task (plan ?planName) (action_type <action_type2>)
            (step 1 $?steps) (parent ?t) )
        (task (plan ?planName) (action_type <action_type3>)
            (step 2 $?steps) (parent ?t) )
        ...
        (task (plan ?planName) (action_type <action_typek>)
            (step <n> $?steps) (parent ?t) )
    )
    <other_effects>
)
```

Figura 3-6: Plantilla de regla de descomposición de tarea no primitiva.

Usualmente las tareas que ejecutan acciones primitivas consisten en conjuntos de reglas que realizan uno o más llamados a procedimientos remotos (*Remote Procedure Call*, RPC), aunque podrían hacer cualquier cosa; la única restricción en este sentido es que alguna produzca en algún momento un hecho que indique si la tarea fue exitosa o no.

En la implementación desarrollada para esta tesis, los RPC corresponden con comandos que se envían a los módulos del robot a través del BlackBoard, utilizando las herramientas descritas en los apéndices B y C, particularmente con la función `send-command` de la herramienta BBCLIPS. Las acciones primitivas más sencillas están compuestas por tres reglas: una regla inicial que envía el comando, una regla que captura una respuesta fallida, y una regla que captura una respuesta exitosa; aunque puede haber muchas más reglas, con un diseño complejo.

Las reglas dexec pueden hacer uso de otros hechos que dan información sobre el estado de ejecución de la tarea. Particularmente, existen cuatro hechos que dan información de este tipo: los hechos `waiting`, que se crean cuando se envió un comando y se está esperando una respuesta; los hechos `BB_timer`, que se crean después de un tiempo determinado, especificado en alguna regla con la función `setTimer`; los hechos `BB_answer`, que contienen la respuesta a un comando; y los hechos `children_status`, que contienen el estado final de la última tarea hija ejecutada (útil cuando el control de ejecución regresa a la tarea padre).

```
(defrule enter_arena-task
        (task (id ?t) (plan ?planName) (action_type enter_arena)
                    (params ?entrance_location) (step $?steps) )
        (active_task ?t)
        (not (task_status ?t ?)        )
        (not (cancel_active_tasks) )
        =>
        (assert
                (task (plan ?planName) (action_type wait_obstacle) (params "door")
                    (step 1 $?steps) (parent ?t) )
                (task (plan ?planName) (action_type arms_goto) (params "navigation")
                    (step 2 $?steps) (parent ?t) )
                (task (plan ?planName) (action_type getclose_location)
                    (params ?entrance_location) (step 3 $?steps) (parent ?t) )
        )
)
```

Figura 3-7: Ejemplo de regla de dexec.

El hecho estructurado `waiting` se crea cada vez que se llama a un procedimiento remoto con la función `send-command`, y se destruye cuando una respuesta es recibida; en la figura 3-8 se muestra su definición. La función `setTimer` sirve para iniciar un temporizador, para que después de transcurrido un determinado tiempo, se cree un hecho con la forma `(BB_timer <symbol>)`, donde $< symbol >$ es un símbolo para identificar el timer del que se trate.

La respuesta a un comando se registra a través de un hecho que tiene la siguiente forma:

```
(BB_answer <command> <symbol> <answer> <params>)
```

donde <`command`> es un string con el nombre del comando respondido, <`symbol`> es un símbolo para identificar la respuesta a una instancia particular del comando, en caso de que más de una tarea envíe el mismo comando; <`answer`> es un 1 ó un 0, indicando si la ejecución fue exitosa o no, respectivamente; y <`params`> es un string conteniendo cualquier parámetro de salida o respuesta del comando. Para más información sobre este sistema de mensajes, consulte el apéndice B. La figura 3-9 muestra un ejemplo sencillo de una tarea primitiva.

El hecho `chlidren_status` es un hecho ordenado que tiene la forma:

```
(chlidren_status <id> <status>)
```

donde <`id`> se refiere al identificador del hecho de tarea que creó las tareas hijas, y <`status`> corresponde con alguno de los símbolos `successful` o `failed`, según haya concluido la ejecución de la última tarea hija ejecutada.

```
(deftemplate waiting
    (slot cmd (type STRING))
    (slot id (type INTEGER))
    (slot args (type STRING))
    (slot timeout
        (type INTEGER)
        (range 0 ?VARIABLE)
    )
    (slot attempts
        (type INTEGER)
        (range 1 ?VARIABLE)
    )
    (slot symbol
        (type SYMBOL)
    )
)
```

Figura 3-8: `deftemplate` de un hecho `waiting`.

Para todo tipo de hechos de este tipo, existen reglas con *salience* menor (menor prioridad para ejecutarse que otras reglas habilitadas) que se encargan de eliminarlos cuando ya no son necesarios (i. e. ya no hay otra regla habilitada que los utilice). De modo que no es estrictamente necesario que el usuario se encargue de destruirlos, sobre todo por que podrían ser utilizados en distintas reglas.

```
(defrule arms_goto-send_command
    (task (id ?t) (action_type arms_goto) (params ?position))
    (active_task ?t)
    (not (task_status ?t ?) )
    (not (cancel_active_tasks))

    (not (waiting (symbol arms_goto)))
    (not (BB_answer "arms_goto" arms_goto ? ?))
    =>
    (send-command "arms_goto" arms_goto ?position 10000)
)

(defrule arms_goto-failed_or_timedout
    (task (id ?t) (action_type arms_goto) (params ?position))
    (active_task ?t)
    (not (task_status ?t ?) )
    (not (cancel_active_tasks))

    (BB_answer "arms_goto" arms_goto 0 ?)
    =>
    (send-command "arms_goto" arms_goto ?position 10000)
)
```

```
(defrule arms_goto-succeeded
    (task (id ?t) (action_type arms_goto))
    (active_task ?t)
    (not (task_status ?t ?) )
    (not (cancel_active_tasks))

    (BB_answer "arms_goto" arms_goto 1 ?)
    =>
    (assert
        (task_status ?t successful)
    )
)
```

Figura 3-9: Ejemplo de reglas de una acción primitiva.

Como se mencionó anteriormente, una tarea puede hacer más de un llamado a algún procedimiento remoto. La recepción de una respuesta fallida no necesariamente significa el fallo de la tarea. Podría hacer falta recibir otras respuestas, o realizar otras acciones, incluso tal vez se podría reintentar enviar un comando fallido un número de veces antes de considerarlo como fallido. Es deber del usuario diseñar los planes de forma tal que se asegure que la ejecución de la acción termina y se crea el hecho que indica el estado final de la ejecución.

Para prevenir el caso en que la ejecución se detenga sin que esté esperando por alguna respuesta o un timer, y sin haber establecido el estado final de la tarea. Existen unas reglas dentro del motor de planeación con un *salience* negativo, que indicarían que la tarea ha fallado, aún sin que las reglas de la tarea lo indiquen.

Estas reglas se muestran en la figura 3-10, y pertenecen a la parte del motor de planeación que corresponde con manejar el resultado de las tareas, que se describirá con un poco más de detalle en la siguiente sección. En este caso, antes de marcar el plan como fallido, se genera un plan para notificar por medio del sintetizador de voz que el robot no sabe cómo realizar esa tarea. La tarea `PE-fail` no debería tener reglas de ejecución diseñadas por el usuario, para forzar a que falle y lo reporte al plan padre (el que falló originalmente).

La tarea `spg_say` se refiere a decir algo con el sintetizador de voz, pues se trata de un planeador pensado para un robot de servicio, y debe anunciar cuando no puede realizar una tarea. Los hechos mostrados en las figuras en este capítulo y el siguiente que comienzan con "PE-" se refieren a "Planning Engine", y son hechos internos que utilizan las reglas del motor de planeación.

De la misma manera, si las tareas hijas de otra tarea han fallado y no existe un plan alternativo (y no se puede volver a activar el mismo plan), estas reglas se encargarían de marcar a la tarea de alto nivel como fallida también.

```
(defrule failed_task-mark_task_without_rules_as_failed-set_failure_task-children_status ; When a task has no enabled rules (either to
assert new sub-tasks or perform an action) it is either an incomplete design or (most likely) a task set to re-plan but with no more
alternatives, which should be marked as failed.
    (declare (salience -9500))
    (task (id ?t) (plan ?planName) (step $?steps) (action_type ?action_type&~PE-fail) (params $?params)) ; see next rule
    (active_task ?t)
    (not (task_status ?t ?))
    (not (waiting))
    (not (timer_sent $?))
    (not (PE-failed ?))
    ?cs <-(children_status ?t failed)
    =>
    (retract ?cs)
    (assert
        (task (plan ?planName) (action_type spg_say) (params "I don't know how to perform action:" ?action_type ".") (step 1 $?steps) (
            parent ?t))
        (task (plan ?planName) (action_type PE-fail) (params "") (step 2 $?steps) (parent ?t))
        (PE-failed ?t)
    )
)

(defrule failed_task-mark_task_without_rules_as_failed-after_failure_task
    (declare (salience -9500))
    (task (id ?t) (plan ?planName) (step $?steps) (params $?params) (action_type ?action_type&~PE-fail)) ; see next rule
    (active_task ?t)
    (not (waiting))
    (not (timer_sent $?))
    (not (task_status ?t ?))
    ?failed <-(PE-failed ?t)
    =>
    (retract ?failed)
    (assert
        (task_status ?t failed)
    )
    (log-message INFO "Task " ?t " failed. (No executable rules).")
    (log-message INFO "Task " ?t ": " ?planName " - " ?action_type " - " (implode$ $?params))
    (stop)
)

(defrule failed_task-mark_task_without_rules_as_failed-delete_orphan_failed_facts
    ?failed <-(PE-failed ?t)
    (not (task (id ?t)))
    =>
    (retract ?failed)
    (log-message WARNING "Orphan PE-failed fact was found.")
)

(defrule failed_task-catch_failed_task ; When a task fails, a task to say that it failed and to later fail is asserted, this rule is
to catch the failure so it won't create a loop and error.
    (task (id ?t) (action_type PE-fail))
    (active_task ?t)
    (not (task_status ?t ?))
    =>
    (assert
        (task_status ?t failed)
    )
)
```

Figura 3-10: Reglas para marcar una tarea inactiva como fallida. Cuando no exista un hecho `children_status` también se marca como fallida.

Tanto para la ejecución de las tareas, como para la selección de planes alternativos, muchas veces es necesario utilizar hechos como banderas que habiliten una u otra regla. Sin embargo, tras la ejecución de la tarea estas banderas se deberían borrar. Más aún, es probable que se quieran crear hechos para actualizar la base de conocimiento sobre el mundo, dependiendo del estado final de la tarea (exitoso o fallido).

Las **reglas de finalización** son reglas para realizar acciones una vez que la tarea haya finalizado, ya sea con estado final exitoso o fallido. Estas reglas no pueden generar tareas de menor jerarquía, sólo son para procesos de actualización de la base de conocimiento. Las reglas de finalización necesitan precondiciones similares a las de las otras reglas, con la diferencia de que en este caso, no importa si existe el hecho que indique que las tareas activas se deban cancelar, pues ya se terminó de ejecutar la tarea; y sí debe existir un hecho que indique que la tarea ha concluido y cuál es su estado final. Los hechos `task_status` (similares a los hechos `children_status`) sirven para representar esta información, y el formato que utilizan es el siguiente:

```
(task_status ?t <status>)
```

donde ?t contiene el id de la tarea cuyo resultado se está reportando (vea la figura 3-4), y <`status`> es uno de los símbolos "`successful`" o "`failed`" (sin comillas), o bien, un comodin sencillo (?), indicando que no importa el estado en que haya terminado, sino que sólo interesa saber que la tarea terminó.

Utilizando una notación similar a la utilizada para describir la planeación clásica, donde una acción corresponde con una tupla que contiene un conjunto de precondiciones y un conjunto de efectos, en este caso se puede modelar una tarea primitiva como una tupla con los mismos elementos, pero que además de los efectos inherentes a la tarea, contiene un conjunto de efectos que se producen si la tarea es exitosa, y un conjunto de efectos que corresponden a que la tarea haya fallado.

Una regla de finalización se muestra en la figura 3-11, donde se actualiza la base de conocimiento después de revisar que una ubicación existe en el módulo que tiene el mapa del ambiente.

```
(defrule check_location_exists-finished
    (task (id ?t) (action_type check_location_exists))
    (active_task ?t)
    (task_status ?t ?)
    =>
    (assert
        (checked_location_exists)
    )
)
```

Figura 3-11: Ejemplo de regla de finalización.

Las **reglas de cancelación** son útiles cuando por alguna razón se crea un nuevo plan mientras se está llevando a cabo una tarea. A veces es necesario cancelar un proceso que está ejecutándose, o modificar la representación del mundo para reflejar el estado en que fue interrumpido. Un ejemplo de esta situación podría ser en que un robot esté navegando hacia alguna ubicación determinada, y una persona le habla para llamar su atención y pedirle algo. Si el comportamiento deseado es que el robot se detenga y atienda a la persona, debería enviar un comando "`stop`" al planeador de movimientos antes de ejecutar el otro plan.

En general, el planeador de acciones detectaría que hay un plan nuevo que podría ser activado antes que el plan que está actualmente en ejecución y ejecutaría las acciones de cancelación de las tareas activas para re-planear. Las reglas de cancelación deben tener las mismas precondiciones que las de las reglas dexec, con la diferencia de que el hecho `(cancel_active_tasks)` debe existir. La figura 3-12 muestra unas reglas de cancelación como las descritas en el ejemplo mencionado.

```
(defrule getclose_location-cancel-start_cancel
    (task (id ?t) (action_type getclose_location))
    (active_task ?t)
    (not
        (task_status ?t ?)
    )
    (cancel_active_tasks)
    (not (BB_answer "mp_stop" cancel_getclose_location ? ?))
    (not (waiting (symbol cancel_getclose_location)))
    =>
    (send-command "mp_stop" cancel_getclose_location "" 1000)
)

(defrule getclose_location-cancel-successful_response
    (task (id ?t) (action_type getclose_location))
    ?at <-(active_task ?t)
    (not
        (task_status ?t ?)
    )
    (cancel_active_tasks)
    (BB_answer "mp_stop" cancel_getclose_location 1 ?)
    =>
    (retract ?at)
)
```

Figura 3-12: Reglas de cancelación de una tarea.

3.3. Algoritmo del planeador

El algoritmo de planeación y ejecución consiste en un ciclo iterativo que revisa en cada paso cuál será la siguiente tarea a realizar, y cada iteración consta de cuatro partes: selección de tareas, descomposición/ejecución, finalización de tareas, y actualización del plan. Frecuentemente existe un ciclo entre las primeras dos partes, para descomponer una parte del plan en acciones primitivas, antes de hacer la actualización del plan.

La selección de tareas consiste en habilitar las tareas de menor jerarquía en cada rama de cada plan y activar las tareas de manera que sean consistentes con el esquema de prioridades. En este contexto, habilitar una tarea solamente significa que son candidatas a ser activadas para su ejecución. Cabe mencionar que las tareas de alto nivel (que han generado tareas de menor jerarquía) serían activadas al menos dos veces: una vez durante la selección de tareas (parte 1) para realizar la descomposición, y otra vez como parte de la actualización del plan (parte 3), cuando la ejecución de las tareas hijas finalice. La figura 3-13 muestra el diagrama de flujo correspondiente al algoritmo básico de planeación/ejecución.

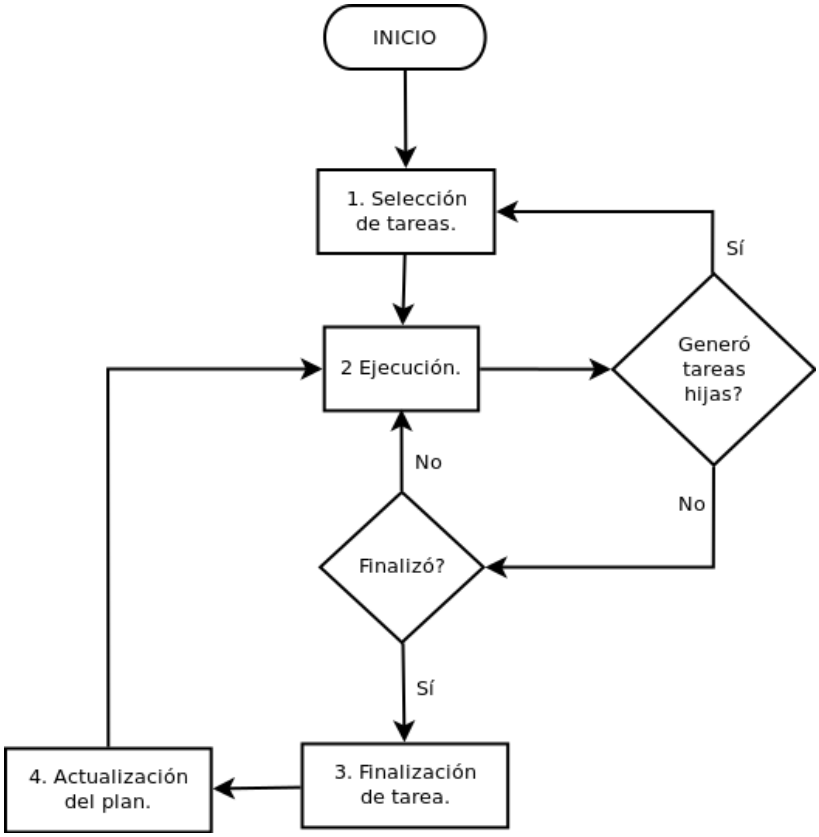


Figura 3-13: Algoritmo de planeación y ejecución.

Por simplicidad, se presentan los algoritmos en diagramas de flujo, aunque su implementación está hecha en código de CLIPS. La secuencia de ejecución de las reglas de CLIPS depende de las precondiciones en cada regla, por lo que es necesario implementar algún mecanismo de control que permita ejecutar algoritmos a través de la manipulación de hechos, de tal forma que se habiliten las siguientes reglas que se deban ejecutar, y deshabiliten las que no deberían ejecutarse.

Para este fin, se utilizaron dos mecanismos: 1) banderas como hechos contenidos en las precondiciones de las reglas; y 2) el *salience* de las reglas, que es una propiedad de prioridad que provee el intérprete de CLIPS para indicar la preferencia por una u otra regla cuando más de una esté habilitada.

3.3.1. Selección de las tareas

La selección de las tareas es el primer proceso en el ciclo del planeador, y consiste a su vez en varios pasos. El primero es detectar si hay una tarea que no esté activa y no existe una tarea con igual jerarquía y orden anterior, o una de otro plan con mayor prioridad que deba estar activa, es decir, identificar si hay una tarea que debería activarse. Tratándose de un proceso iterativo, es posible que existan otras tareas activas, ya sean tareas de mayor jerarquía en el mismo plan, o bien, tareas pertenecientes a otro plan.

La estructura de los planes y la relación entre las tareas que pertenecen a él, está codificada en los datos en los hechos estructurados `task`, pero cada tarea es en realidad un hecho independiente, que podría crearse, borrarse o de otra forma modificarse por acción del usuario, independientemente de su contexto en el plan al que pertenece. Sin embargo, sería imprudente que un usuario manipule estos hechos arbitrariamente, pues sería dificil obtener los resultados esperados y probablemente corrompería los planes. Por eso, las tareas (a excepción de las de más alto nivel) son creadas solamente por las reglas del mismo plan, donde se tiene una idea del contexto de ejecución actual; dicho de otra forma, una regla sólo puede crear tareas hijas de la tarea a la que pertenece dicha regla.

Por lo tanto, las tareas nuevas (que no son de alto nivel) son producto precisamente de las tareas activas en el mismo plan, y no tendría sentido cancelarlas. Entonces cuando se detecte que hay una tarea nueva que deba activarse, habrá que primero desactivar las tareas pertenecientes al mismo plan (que son de una jerarquía mayor), y después cancelar las tareas activas de otros planes que no puedan ejecutarse en paralelo con esta tarea.

Una vez que se desactivaron o cancelaron las tareas activas, comienza la planeación, para lo cual se marcarán como *habilitadas* las tareas que tengan la menor jerarquía en cada rama de cada plan distinto, es decir, las tareas que no tengan tareas hijas y que no exista una tarea de la misma jerarquía con un orden anterior que el suyo. Las tareas habilitadas son posibles candidatas para ser activadas, sin embargo, es posible que no todas puedan ejecutarse en paralelo y habrá que escoger una para activar primero, y después activar cualesquiera que puedan ejecutarse en paralelo con ella.

Para definir la tarea con mayor prioridad se comparan las tareas por pares y se va descartando una a una, hasta tener sólo una tarea, la más prioritaria, que sería activada. Esta tarea se marca entonces como *activable*, e inicia un proceso que iterativamente busca y marca como *activable* a una tarea que pueda ejecutarse en paralelo con las tareas *activables* hasta el momento. Cuando no haya más tareas potencialmente activables, se activan las tareas seleccionadas y comienza su ejecución.

Al ser las prioridades asociadas, independientes del contexto de ejecución dentro del plan, es probable que las tareas no presenten la propiedad de transitividad, i. e. podrían existir ciclos en las prioridades asociadas de las tareas pertenecientes a distintos planes. En ese caso no es posible definir una tarea *más* prioritaria, pues al comparar dos tareas pertenecientes al ciclo se descartaría necesariamente una que es prioritaria sobre otra que no está siendo comparada. Cuando esto ocurre, se activará un plan arbitrario en el ciclo, dependiendo del orden en que el intérprete de CLIPS compare las tareas.

El algoritmo de comparación de dos tareas que define cuál es prioritaria, utiliza las prioridades asociadas de las tareas en cuestión. Cuando no hay información suficiente para determinar cuál es prioritaria, se utilizaría la prioridad asociada de los nodos ancestros en el árbol de tareas. Es decir que se necesita considerar todo el camino de tareas desde el nodo raíz hasta el nodo hoja que represente cada tarea habilitada, i. e. una lista de nodos por cada tarea.

La figura 3-14 muestra un ejemplo de dos listas correspondientes con dos tareas y sus antecesores. Los nodos $r1$ y $r2$ corresponden con los nodos raíz de los planes 1 y 2, respectivamente, los nodos $p1$ y $p2$ son tareas intermedias o nodos padres en los caminos, y los nodos $t1$ y $t2$ son los nodos hoja correspondientes con las tareas que se están comparando en esos planes, las de menor jerarquía. Como se puede observar, la prioridad del nodo $p1$ es mayor que la de $p2$, por lo que esa tarea es prioritaria, sin embargo, existe una tarea de menor jerarquía ($t2$) que tiene una mayor prioridad.

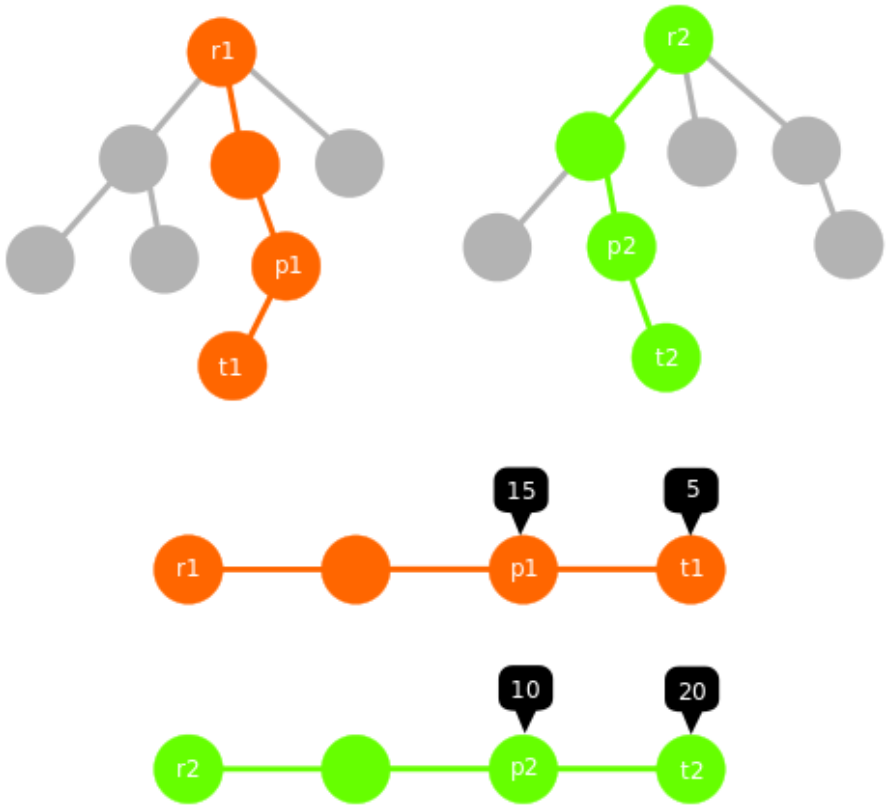


Figura 3-14: Ejemplo de caminos de tareas con prioridades asociadas.

Dado que las tareas pueden tener prioridades asociadas independientemente del contexto de ejecución (i. e. aunque sean tareas padres de otras), surge la cuestión de si utilizar las prioridades asociadas de una tarea de alto nivel o de las de menor jerarquía para definir si una tarea tiene prioridad sobre otra. Cada nivel de profundidad en el árbol del plan corresponde con una tarea más granular y específica que la de su padre, y de igual forma su contexto es más específico y concreto que el de una de más alto nivel. Interesa entonces, tomar en cuenta las prioridades asociadas de las tareas de menor jerarquía.

No obstante, los niveles de profundidad de un plan no necesariamente coincide con el mismo nivel de especialización que las tareas del mismo nivel en otro plan. Además habría también que tomar en cuenta que las tareas podrían no tener una priodidad asociada. Entonces se necesitaría buscar hacia atrás, por cada lista, el nodo más cercano al nodo hoja (a la tarea que se está comparando) que tenga una prioridad asociada. Este nodo sería la *referencia* inicial de la tarea.

El algoritmo de comparación de tareas mantiene una referencia a un nodo en cada lista, que podría servir para determinar si una tarea tiene prioridad sobre otra. Inicialmente, esta referencia sería el primer nodo que tenga un valor de prioridad asociada (recorriendo la lista en orden inverso).

Idealmente, una tarea tendría entonces prioridad sobre otra si la prioridad asociada de su referencia es mayor que la de la referencia de la otra tarea. En caso de que las prioridades asociadas de las referencias sean iguales, habría que encontrar la siguiente referencia útil por cada lista, buscando la lista en orden inverso a partir de ese nodo, de modo que siempre se encuentre el nodo de menor jerarquía que podría ser útil.

Cuando no se ha podido determinar la prioridad de una tarea sobre otra, y alguna de las listas ha alcanzado el inicio de la lista sin encontrar un nodo con prioridad asociada (ya no se encontró una nueva referencia), le selección de la tarea sería arbitraria. Sin embargo, dependiendo del orden en que las tareas sean comparadas por el intérprete, podría ocurrir alternancia de planes, lo cual es indeseable, pues la ejecución de los planes podría ser ineficiente y poco natural.

Para evitar esto, en cada iteración del planeador se actualiza un hecho que guarda una referencia al plan de la tarea que fue seleccionada para activarse primero (la *más prioritaria*), de modo que cuando no se pueda determinar la prioridad de una tarea sobre otra, si una de ellas pertenece al mismo plan, se descartaría la otra.

La figura 3-15 muestra los diagramas de flujo de los algoritmos de selección y comparación de tareas.

3.3.2. Ejecución de tareas

Una vez que se han activado las tareas para ser ejecutadas, se habilitarán las reglas correspondientes a ellas. El poder de un planeador implementado como un sistema experto, es que el conocimiento sobre la solución del problema está embebido en el diseño de las reglas. El diseño de las reglas es un proceso manual y el usuario creador de los planes debe diseñarlos de forma tal que realicen lo que él desee.

En esta sección, aunque denominada de manera genérica como ejecución de tareas, se contemplan la descomposición, ejecución, cancelación y finalización de tareas. Dado que los planes son diseñados por los usuarios, no hay mucho qué decir en concreto sobre el propósito de la ejecución de las reglas, más allá de que deberían producir una secuencia de acciones cuyos efectos logren los objetivos esperados.

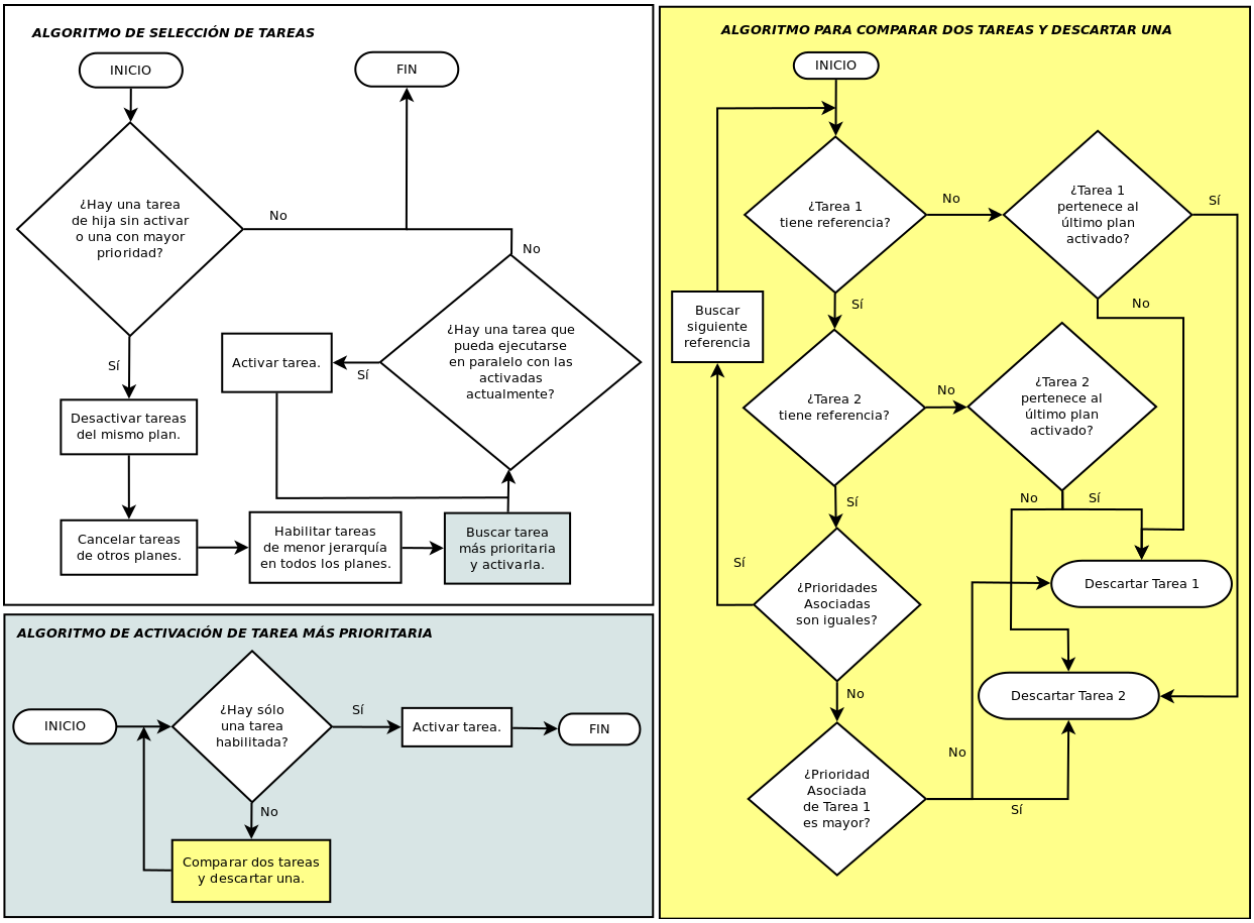


Figura 3-15: Diagramas de flujo de los algoritmos de selección y comparación de tareas.

No obstante, es importante contemplar esta parte del algoritmo de planeación/ejecución para entender la relación que tiene con los otros procedimientos, así como para adquirir una visión más aproximada del enfoque pretendido en el diseño de los planes.

Para cada uno de los casos (dexec, cancelación y finalización), distintas reglas pueden existir. Idealmente, cada una debería manejar un escenario que las otras no contemplan, aunque no necesariamente serían mutuamente excluyentes, i. e. más de una regla puede estar habilitada. La activación de una regla dependería del cumplimiento de sus precondiciones; es decir, de la situación presente durante la ejecución, representada a través de hechos.

Las tareas dexec son los principales elementos de los planes, su función es resolver las dependencias necesarias para la ejecución de esta tarea (o ese objetivo). Note que una tarea no primitiva en el sentido tradicional corresponde con una tarea dexex con ciertas dependencias o precondiciones que tiene que cumplir, y sin ninguna acción primitiva que ejecutar; es decir, la ejecución de las tareas hijas determinaría el éxito de esta tarea.

La descomposición de las tareas en tareas de menor jerarquía es la parte donde se hace la planeación propiamente. De acuerdo a la sitaución actual del ambiente (que cumple con las precondiciones de las reglas), el plan podría descomponerse de una manera u otra. Será deber del diseñador de los planes contemplar las precondiciones pertenecientes a cada situación que se quiera manejar, para que las tareas generadas efectivamente logren los objetivos esperados.

Una tarea dexec de alto nivel podría activarse varias veces, ya sea debido a que después de la ejecución exitosa de tareas hijas aún necesite resolver alguna dependencia, o bien, que las tareas hijas hayan fallado y necesite generar un plan alternativo para resolver esa dependencia.

La función de las tareas que realizan acciones primitivas es afectar directamente el ambiente real, a través de sensonres y actuadores del robot, o más comúnmente, a través de interfaces hacia ellos con ayuda de otros programas. Para esto utilizan llamadas a procedimientos remotos (RPC), que en esta implementación corresponden con `comandos` de BlackBoard. El usuario tendría que diseñar las reglas de tal forma que asegure que esto suceda en algún momento.

Otra forma en que puede terminar la ejecución de una tarea es si las tareas activas son canceladas por el motor de planeación. Las reglas de cancelación tienen la finalidad de realizar operaciones que dejen al robot en un estado disponible para que se lleve a cabo otro plan (cualquier otro plan), y que permitan continuar o reiniciar la ejecución de la tarea actual en un momento posterior. Esto es especialmente útil cuando existen procedimientos remotos que necesiten detenerse o notificarles de alguna forma, por ejemplo: si el robot está navegando, otro módulo (como el planeador de movimientos) sería quien tendría el control del movimiento del robot, y habría que decirle que pare para que pueda atender al otro plan.

Las reglas de cancelación pueden manejar llamadas a procedimientos remotos, pero no deben generar nuevas tareas. Opcionalmente podrían generar un hecho `task_status` para la tarea que están cancelando, si el diseñador del plan lo considera adecuado, aunque probablemente en la mayoría de los casos no sería el caso.

Las reglas de cancelación se activarían solamente cuando un nuevo plan, que tendría prioridad sobre el que se está ejecutando actualmente, se creara en tiempo de ejecución, de modo que habría que interrumpir las acciones que se estén llevando a cabo. Las reglas de finalización, en cambio, pueden tener una función más inherentemente relacionada con el diseño y la ejecución de los planes.

El propósito de las reglas de cancelación es actualizar la representación interna del mundo después de finalizar la ejecución de una tarea, lo cual usualmente puede significar borrar banderas que se hayan utilizado para la ejecución de esta tarea o sus descendientes, aunque puede ser cualquier

creación o borrado de hechos de un modelo de representación interno.

Las reglas de finalización de una tarea pueden ser varias, no necesariamente excluyentes, y pueden tomar en cuenta el estado final de la tarea (exitoso o fallido) para realizar distintas acciones, o ignorarlo y realizar acciones independientemente del resultado de la tarea.

3.3.3. Actualización del plan

Si una tarea está activada, pero no hay timers ni llamadas a procedimientos remotos en ejecución, es decir, que no está haciendo nada, se marcará como fallida o exitosa automáticamente, aunque idealmente el usuario debería diseñar los planes para que las tareas siempre generen un estado final. Se marcaría exitosa cuando exista un hecho `children_status` con un estado exitoso. Cuando exista un hecho `task_status` para una tarea, su ejecución terminará, y el control pasará al motor de planeación, que actualizará el plan.

Como se ha mencionado, cada tarea primitiva debe al final producir un hecho que indique si la tarea fue exitosa o fallida. Este hecho es el que el motor de planeación utiliza para actualizar el plan. Con la estructura jerárquica que se genera al descomponer los planes, cada tarea reporta el estado final de su ejecución a la tarea padre para que ésta pueda continuar con la ejecución de la siguiente tarea hija, ejecutar un plan alternativo, en caso de que haya fallado, o bien, reportar su estado final a la tarea del siguiente nivel jerárquico hacia arriba.

En el algoritmo de actualización de los planes, lo primero que se hace con una tarea que tiene un hecho de `task_status` es eliminar recursivamente a sus hijos. Esto permite que al detectar que un objetivo de alto nivel se ha cumplido o haya fallado, se cancele y borre cualquier tarea que haya derivado de él. Después, dependiendo del estado final de la tarea, se manejará de distinta manera.

Si la tarea fue exitosa, primero hay que eliminar la tarea que se está reportando como exitosa (junto con los hechos de `active_task`, `task_status`, etc.), después habrá dos opciones: ejecutar la siguiente tarea de la misma jerarquía, o bien, si no hay otras tareas de la misma jerarquía, reportar a la tarea padre que todas las tareas hijas han sido exitosas. En el segundo caso, lo que ocurrirá después de borrar la tarea es que se creará un hecho `children_status` con un estado final exitoso.

Si la tarea fue fallida, habría que primero eliminar las demás tareas de la misma jerarquía, pues la ejecución no podría continuar. Después habría que eliminar la tarea cuyo estado final fue fallido y generar un hecho `children_status` con un estado final fallido. Esto permitiría que el planeador, en la siguiente iteración, vuelva a seleccionar a la tarea padre como la de menor jerarquía para ser ejecutada, y la tarea podría reintentar realizar el mismo plan, o descomponerse en un plan

alternativo. Si no hubiera un plan alternativo ni es posible reintentar realizar el mismo plan, se marcaría automáticamente la tarea como fallida, para reportarlo a la tarea padre.

En cada caso, habría que considerar el caso especial en que se trate de una tarea raíz, que ya no tiene tarea padre a quién propagar el resultado. En este caso, la implementación actual además emite un mensaje indicando que el plan ha fallado o se ha ejecutado exitosamente, según sea el caso. La figura 3-16 muestra un diagrama de flujo con el algoritmo de actualización del plan.

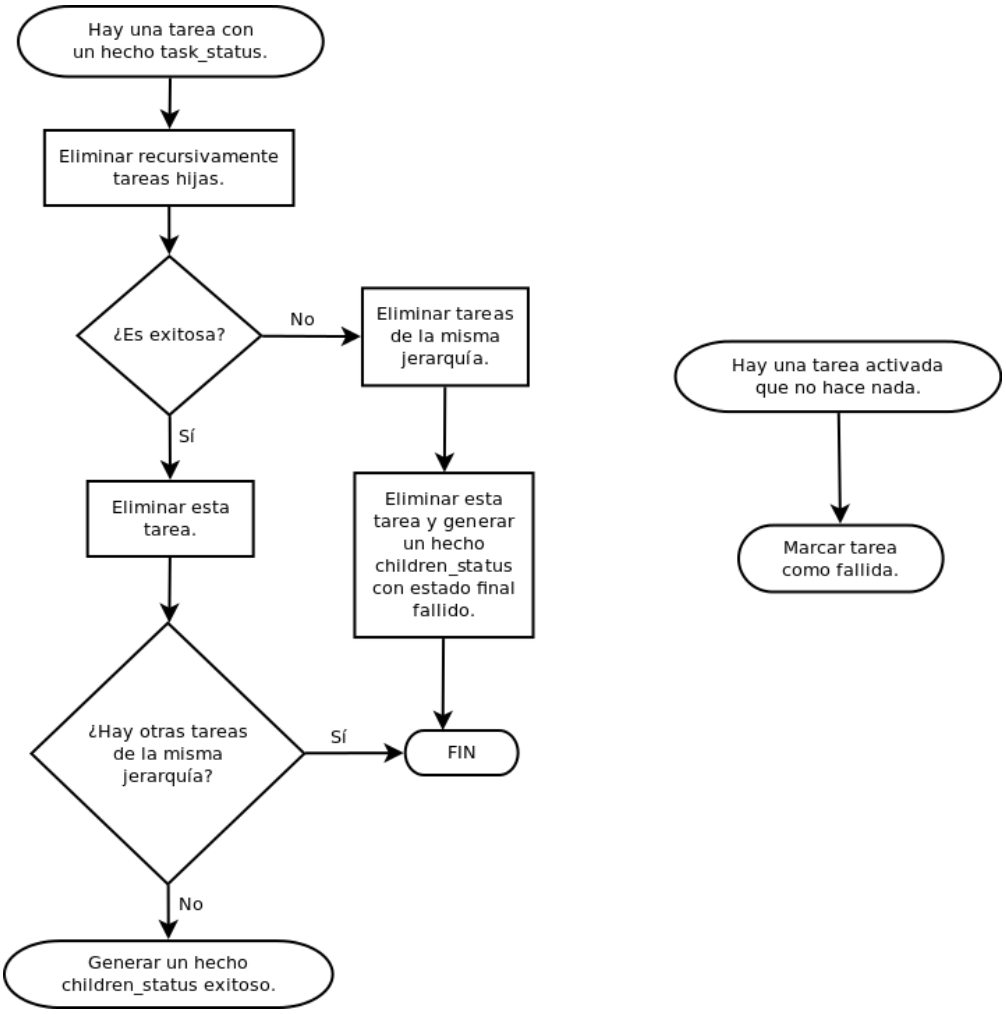


Figura 3-16: Diagrama de flujo del algoritmo de actualización del plan.

Note que las tareas van eliminándose conforme se ejecuta el plan, de modo que el algoritmo de selección pueda encontrar la siguiente tarea de menor jerarquía en cada iteración. También observe que el árbol correspondiente al plan no se expande completo, sino que se expande una rama a la vez, la que está actualmente en descomposición/ejecución (a menos que se puedan ejecutar tareas en paralelo).

Esto quiere decir también que las únicas tareas que pueden estar activadas son las de menor jerarquía, los nodos hoja del árbol, y serán más de una siempre que puedan activarse en paralelo con la tarea principal seleccionada por el planeador.

Finalmente, es importante notar que de acuerdo al algoritmo de actualización del plan, si se quiere que una tarea propague a la tarea padre que ha fallado, todas sus reglas de descomposición tendrían que estar deshabilitadas. Al no tener ninguna regla habilitada que pueda descomponer el plan, se marcaría la tarea como fallida, o mejor aún, incluir explícitamente una regla que marque que el estado final de ejecución de esa tarea es fallido.

Esto implica que en todos los planes donde se quiera que la tarea reporte su estado fallido (lo cual es el comportamiento deseado, salvo casos específicos que estén contemplados para reintentar el mismo plan en caso de falla) deberían generar o eliminar hechos, de tal forma que se deshabiliten las reglas de descomposición que ya hayan sido ejecutadas, y así, en dado caso, sólo estaría habilitada la regla que marca como fallida la ejecución.

3.4. Tolerancia a fallas

Entre los aspectos más complicados de la planeación y ejecución autónoma por parte de un robot, es mantener una representación correcta del entorno donde se realizan. Muchas veces el robot debe estar *situado* en circunstancias particulares para ejecutar una acción, y si no lo está, el resultado no sería el esperado. Para agregar robustez, y evitar que esto suceda, se implementan mecanismos de manejo de errores o tolerancia a fallas en los planeadores y ejecutores. El trabajo aquí presentado cuenta con tres mecanismos que ayudan a que la ejecución de los planes sea adecuada, embebidos en el mecanismo de planeación descrito: planes alternativos, planes correctivos y planes de verificación.

Los **planes alternativos** consisten en que cuando una tarea falla, se eliminan las tareas de la misma jerarquía para que la tarea padre recupere el control de ejecución y pueda descomponerse en un nuevo plan para ejecutar.

Idealmente, la elección de un nuevo plan estaría determinada por una observación diferente del ambiente, que habilite las precondiciones de una regla de descomposición distinta. Sin embargo, es posible que en ocasiones no se cuente con esa información (o no se haya detectado), el plan tendría que estar diseñado para realizar alguna de las siguientes: reintentar el mismo plan (la misma regla de descomposición que se usó anteriormente estaría habilitada), o administrar con banderas la activación de una u otra regla de descomposición.

Puesto que depende de la representación interna del entorno, si ésta no es adecuada, este mecanismo podría resultar poco efectivo por sí mismo, e incluso producir comportamientos no deseados, como reintentar indefinidamente un plan que continúa fallando. Sin embargo, los otros dos mecanismos pueden ser útiles para detectar justamente los motivos de las fallas, y al combinarlo con alguno de ellos, se podría obtener una importante mejoría en el manejo de situaciones inesperadas.

Los **planes correctivos** son planes que deben ejecutarse *antes* de la tarea actual. Este manejo de errores se puede considerar como una especialización o una interpretación particular de la filosofía de resolución de dependencias de tareas que se había presentado. Una tarea podría generar tareas hijas, cuyo fin sería corregir o verificar alguna circunstancia en el ambiente, o dicho de otra forma, *situar* al robot para que pueda ejecutar la acción correspondiente. Dependiendo del estado final de las tareas hijas, al recuperar el control de ejecución, esta tarea podría ejecutar alguna acción primitiva, descomponerse en otra tarea hija para resolver otra dependencia, o bien, descomponerse en un plan alternativo (si hubiera).

Evidentemente, el plan resultante sería equivalente a un plan distinto que desde un inicio hubiera sido descompuesto por una tarea de jerarquía mayor, sin embargo, en este enfoque las tareas son objetivos que deben cumplirse, y los planes correctivos permiten resolver esos objetivos según las condiciones actuales del entorno. Más aún, los planes correctivos frecuentemente son generados en respuesta a algún evento que ocurra durante la ejecución de la tarea que lo crea. Por ejemplo, si el comando de reconocer objetos no está reconociendo nada, podría ser que el sistema de visión necesite reiniciarse, o bien, que tenga que buscar el objeto en una ubicación distinta, para lo cual tendría que navegar hacia la nueva ubicación.

Los **planes de verificación** representan el último mecanismo de manejo de errores y tolerancia a fallas implementado. Estos planes se ejecutarían al final de la ejecución de una tarea para verificar que efectivamente la tarea se ha llevado a cabo exitosamente, o bien, para tratar de identificar por qué una tarea ha fallado, de modo que pueda actualizar su representación interna del mundo y en consecuencia habilitar la descomposición de un plan distinto.

Nuevamente, sintácticamente corresponden a reglas de planeación dexec, pero semánticamente identificar planes de esta forma ayuda a prevenir errores producidos por un proceso imperfecto en una acción primitiva.

La idea es que después de la ejecución de una acción primitiva por una tarea, antes de reportar como exitosa la ejecución de dicha tarea, generaría una tarea hija (o más) para verificar que las acciones en verdad tuvieron el efecto deseado. Para esto, por supuesto, es necesario que se cuenten con mecanismos que permitan verificar esto. Normalmente corresponden con acciones primitivas de sensado, que no pretenden modificar el entorno, sino revisar la situación actual.

Si el resultado original es exitoso, podría llamar a un procedimiento de verificación a través de la generación de tareas hijas. Si este procedimiento fuera exitoso, el resultado sería reportado a la tarea padre, de modo que el estado final de la tarea sería efectivamente, el resultado de la verificación.

Si en cambio el resultado original fuera negativo, podría levantarse un plan de verificación que intentara identificar la causa, verificando una expectativa que se piensa falsa. Si la verificación confirmara que la hipótesis era falsa, tendría que actualizar la representación del mundo y marcar la tarea como fallida, de modo que (después de la actualización) la tarea padre pueda generar un plan alternativo, si hubiera. Si la verificación comprobara que la hipótesis es verdadera (i. e. ésta no es la causa de la falla), la tarea padre tendría que continuar haciendo el diagnóstico hasta encontrar la causa, o finalmente reportarse como fallida.

Note que el diseñador de los planes debería contemplar un mecanismo para determinar cuando ya se ha ejecutado el plan de verificación y la tarea ha fallado nuevamente. Lo más sencillo y directo es crear hechos bandera para representar eso y agregar reglas de finalización que limpien esas banderas, lo cual permitiría al algoritmo de ejecución continuar con la actualización del plan.

Como ejemplo, considere el caso en que se desea tomar un objeto, que corresponde con una acción en que se le indica a un brazo robótico que se mueva a una cierta ubicación y accione su manipulador para cerrar y tomar el objeto, entre las reglas de verificación podría haber una que cree una tarea de menor jerarquía para corroborar por algún método (visión, revisar el torque, confirmación por voz, etc.) que en efecto el objeto ha sido tomado. El éxito o fracaso de la tarea estaría verificado en dos pasos: cuando se realiza la acción propiamente, y cuando se realiza la comprobación; aumentando así la tolerancia a fallas.

3.5. Sugerencias de diseño

La filosofía de diseño de los planes sugerida para usar con este planeador es: plantear objetivos principales y dejar que las reglas generen las tareas que cubran los prerrequisitos para completarlos. La idea es que el diseñador de planes no genere un procedimiento para lograr un objetivo, sino que se genere a partir de las condiciones que el robot detecte, usando los planes alternativos y planes correctivos para generar el plan completo. Cuando una tarea falla, muchas veces (aunque no necesariamente) puede significar que el plan no está correctamente diseñado para considerar todos los casos que lo hacen fallar, y por lo tanto, no ha encontrado una manera adecuada de solucionarlo.

El nombre "correctivo" hace alusión a que debe encontrar una solución a un *problema* existente. El término puede llevar a pensar que sólo deben crearse cuando algún actuador o sensor ha fallado, o bien, que la representación del mundo es diferente del ambiente real. Sin embargo, en realidad su función es situar al robot para que pueda ejecutar los planes correctamente, ya sea que haya ocurrido un error como estos, o que aún haga falta realizar acciones que preparen al robot para ejecutar la acción pretendida. En general los planes correctivos son preferidos sobre la descomposición de planes, pues son los que permitirían la planeación dinámica, ajustada a la situación del entorno.

Justo esta capacidad de generar tareas *anteriores* durante la ejecución es lo que permite que el diseño de los planes esté más orientado a objetivos. Esto a su vez es posible gracias al motor de inferencias y al paradigma de un sistema basado en reglas, en vez de en una ejecución predeterminada, que hace más sencillo considerar diferentes escenarios que se puedan presentar.

La especificación de los planes permite obtener parámetros a través de la descomposición de planes, en el campo "params" de cada tarea, o bien, de hechos independientes que representen el conocimiento sobre el mundo. Sería deber del usuario ponderar cuál es más fácil y/o conveniente usar en cada caso. En general la representación del mundo debería ayudar a discernir si descomponer un plan de una u otra forma, mientras que los parámetros son utilizados por una tarea en concreto, aterrizada al ambiente en el que se está desempeñando la tarea. Por ejemplo, puede ser el nombre de un objeto en específico que el robot tendría que tomar, o una ubicación en el mapa, pero no debería ser un parámetro que indique cómo debe tomar el objeto o qué técnicas debe utilizar para evadir obstáculos.

Cada tarea debería modificar la representación interna del entorno cuando sea conveniente, pero cualquier otro hecho que haya generado durante su ejecución, como banderas para controlar el flujo de ejecución o para deshabilitar algún plan de descomposición, debería limpiarse cuando la ejecución termine, las reglas de finalización son ideales para este fin.

En general, las tareas deberían considerarse como objetivos independientes, no como partes de un procedimiento. Esto implica que en la medida de lo posible se mantenga el diseño de las reglas independiente del contexto de ejecución, i. e. sin suponer nada sobre qué tareas ha tenido como ancestros o qué tareas la sucederán. Incluso se podrían considerar precondiciones que se hayan considerado en una tarea de nivel jerárquico superior. Esto permitiría que si otro plan o algún evento en el ambiente modifica la situación del entorno, las tareas de menor nivel jerárquico puedan utilizar esa información y planear dinámicamente cómo debería actuar en consecuencia.

Esto está relacionado con el concepto de "amenaza" presentado en el capítulo de planeación espacio-plan. Una amenza ocurre cuando una tarea podría eliminar las precondiciones de otra que debería llevarse a cabo en el futuro. Los planes para este motor de planeación necesitan especial atención en este punto debido a que esta implementación permite la ejecución multi-tarea, y una tarea podría haber modificado la situación que otra tarea suspendida consideraba cierta.

Por esta razón es importante que cada tarea modifique los hechos de la representación del mundo correspondientes con sus efectos. Asumiendo que los planes consideren las situaciones que podrían presentarse, mientras más precisa sea la representación del mundo en todo momento, más robusto será el planeador para decidir qué hacer cuando la situación haya cambiado.

Finalmente, puesto que la utilidad de estos planes se verá afectada enormemente por la representación del mundo, o mejor dicho, por la capacidad del robot de mantenerla lo más precisa posible. Sería deseable que los distintos módulos del robot cuenten con pruebas unitarias de distinta granularidad, de modo que permitan determinar el motivo de alguna falla. Aunque por supuesto esto queda fuera del alcance del planeador.

CAPÍTULO

4

DISEÑO GRÁFICO DE PLANES

Entre los desafíos que frecuentemente enfrentan varios grupos interdisciplinarios donde colaboran estudiantes, tales como el laboratorio de Bio-robótica de la UNAM, se encuentra la constante rotación de personal. Continuamente los estudiantes egresan de sus programas de educación superior y posgrado, dejando al laboratorio para continuar su carrera profesional, a la vez que nuevos estudiantes se incorporan. Cuando nuevos estudiantes llegan, comúnmente con distintos conocimientos y perfiles, es necesario que pasen por un proceso de capacitación para que puedan realmente ser productivos para el laboratorio.

Uno de los objetivos de este trabajo es que el diseño de los planes sea un proceso muy intuitivo y fácil de realizar. Esto implica que los usuarios no deberían tener que aprender a programar en un lenguaje y un paradigma distinto a los que conocen. Para esto, se ha desarrollado una herramienta que permite definir las reglas de planeación de manera gráfica, cuya representación en código de CLIPS podría ser generada.

El software presentado en este capítulo está basado en el trabajo desarrollado con el Dr. Pedro Lima en el Instituto Superior Técnico, en Lisboa; en una estancia académica realizada como parte del programa de movilidad internacional de estudiantes de la UNAM, realizada de marzo a junio del año 2014.

La herramienta que facilita el diseño de los planes se llama **Petri Net Plan Design Tool (PNPDT)**, y utiliza el formalismo de redes de Petri de alto nivel para representar los hechos y las reglas necesarias para diseñar los planes. Con ella se obtiene un buen compromiso entre facilidad de uso y expresividad de los planes que pueden ser generados.

La figura 4-1 muestra la interfaz gráfica del programa. Se puede observar a la izquierda el área del *explorador del proyecto* y a la derecha el *área de trabajo*. La interfaz en ambas secciones está basada en eventos del mouse, donde el click derecho muestra menús contextuales, y el izquierdo se utiliza para concluir la operación seleccionada. El teclado serviría para nombrar los lugares y transiciones, y en ocasiones para utilizar algunos atajos como las funciones deshacer (Ctrl+z) y rehacer (Ctrl+y).

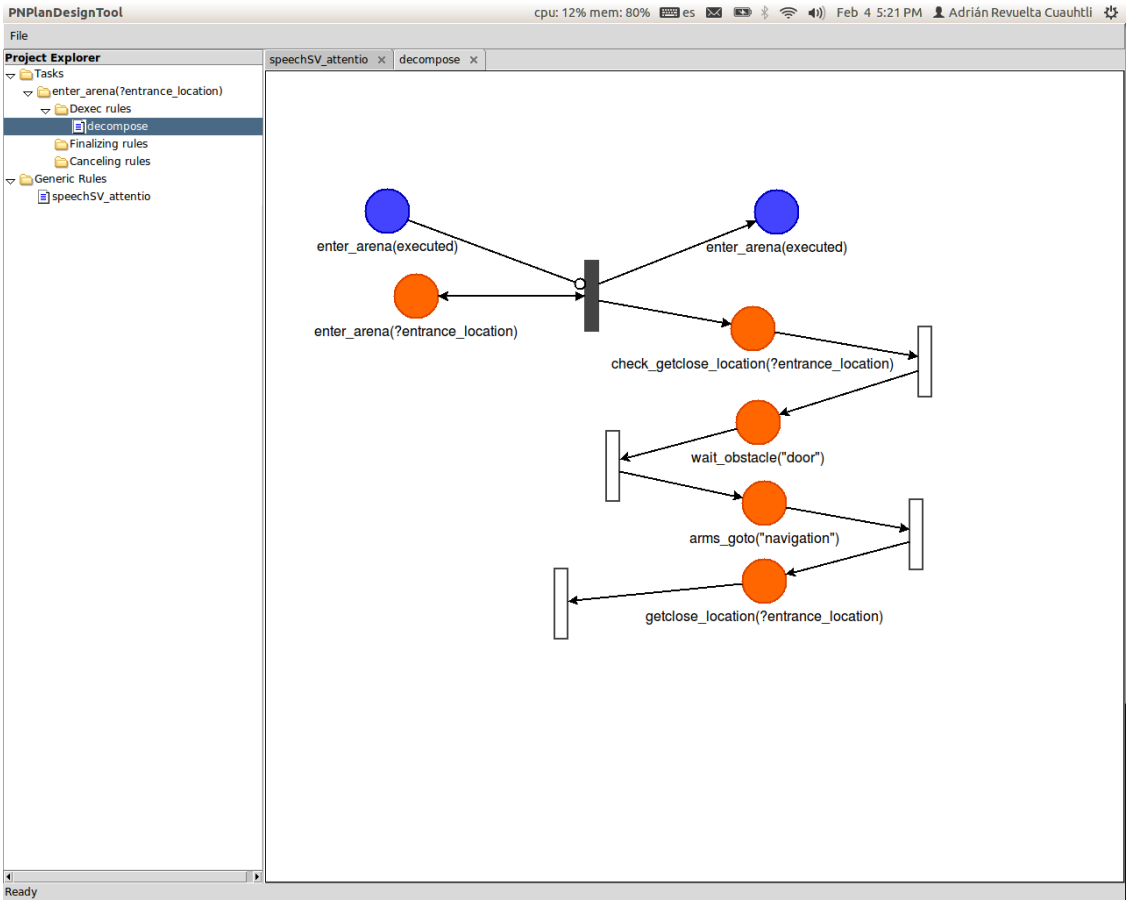


Figura 4-1: Interfaz gráfica de la herramienta PNPDT.

Para consultar el código fuente y descargar la aplicación puede acceder a la página de GitHub del proyecto: `https://github.com/arcra/PNPlanDesignTool`.

4.1. Planes con Redes de Petri

En el explorador del proyecto existen dos carpetas, correspondientes con tareas y reglas genéricas de CLIPS. Al crear una tarea, se creará una carpeta hija conteniendo a su vez tres carpetas: reglas de dexec, reglas de finalización y reglas de cancelación. Cada diagrama de redes de Petri que se agregue a ellas corresponderá con una regla de planeación, según se describieron en el capítulo anterior.

Los hechos en las reglas estarían representados por los lugares, donde cada lugar tiene una etiqueta (o nombre) que sigue una sintaxis similar a un enunciado de lógica de primer orden (FOL, por sus siglas en inglés). Los símbolos constantes comienzan con una letra minúscula, a la que pueden seguir letras minúsculas o mayúsculas, guiones altos o bajos, o números. Los símbolos variables siguen la misma sintaxis, con la diferencia de que antes llevan un signo de interrogación (?). Cuando un lugar tuviera al menos un token en él, indicaría que el hecho existe o es verdadero, de otra forma, se interpretaría como que el hecho no existe, o lo que es lo mismo, según el principio del *Closed World Assumption*, que el hecho es falso.

Aunque estas etiquetas siguen una sintaxis similar a los enunciados de FOL, no es exactamente igual. Las diferencias principales son que se puede utilizar un comodín cuando no interese obtener un argumento del enunciado, y se pueden utilizar comodines múltiples y variables múltiples para capturar varios argumentos del enunciado en una sola variable. Esto está relacionado con la manera en que CLIPS puede utilizar estos símbolos para encontrar correspondencias entre patrones en las precondiciones y los hechos existentes, pero no debería ser un concepto muy complicado de asimilar; en el siguiente ejemplo se ilustrará un poco má este punto.

La figura 4-2 muestra una plantilla de una regla básica: una regla de CLIPS corresponde con una transición de un diagrama de Petri. Los lugares de entrada de la transición son las precondiciones de la regla, y los lugares de salida son las acciones o efectos de la regla. Las precondiciones dos y tres equivalen a enunciados sin argumentos, por simplicidad se pueden omitir los paréntesis.

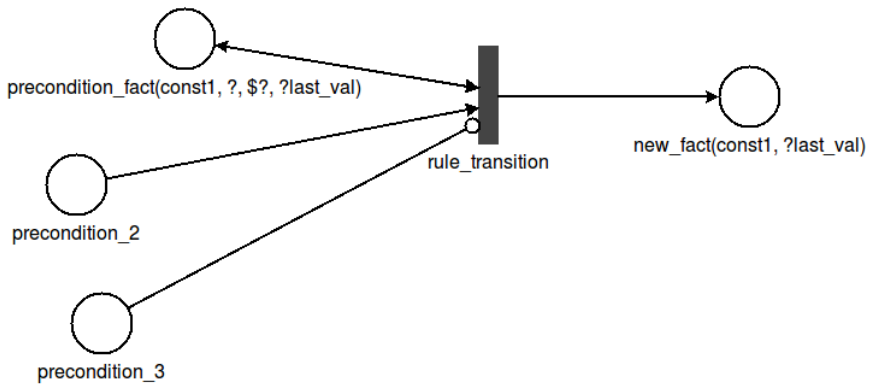


Figura 4-2: Regla genérica de CLIPS en un diagrama de redes de Petri.

Note que en la figura, el arco de cada precondición es distinto: la primera tiene un arco bidireccional, indicando que ese hecho es una precondición y que *no* sería eliminado; la segunda tiene un arco simple, indicando que además de funcionar como precondición, ese hecho sería eliminado por los efectos de la regla; y finalmente la tercera precondición tiene un arco inhibidor, que indica que la regla estaría habilitada cuando ese hecho *no* exista (y se cumplan las demás precondiciones).

En general, las transiciones funcionan como conjunciones de precondiciones (conectivos lógicos AND), es decir, todas las precondiciones que estén conectadas a ellas deben cumplirse para que la regla esté habilitada. Esta característica resulta convenientemente muy parecida al mecanismo por el que se habilita una transición en una red de Petri: cuando todos los lugares de entrada tienen un token (o no lo tienen, cuando están conectados con un arco inhibidor), la transición estaría habilitada.

Por otro lado, el hecho de la derecha de la transición, corresponde con los efectos de la regla, es decir, sería un hecho nuevo que la regla crearía. Los hechos creados por las reglas pueden contener como argumentos solamente constantes o variables aterrizadas (asignadas) en las precondiciones, a diferencia de las precondiciones, donde además se pueden usar comodines y variables nuevas. Cabe mencionar que el alcance de las variables utilizadas en una red de Petri es el de la regla en la que están incluidas. Es decir que si una variable aparece más de una vez en los lugares de una regla, se referirá al mismo valor, no siendo así entre distintas reglas.

Para darse una mejor idea de cómo funcionan los patrones de las precondiciones se ha incluido una precondición relativamente compleja, que utiliza un valor constante, un comodín sencillo, un comodín múltiple y una variable sencilla. Así, la primera precondición del ejemplo podría habilitarse con cualquiera de los siguientes hechos de CLIPS:

```
(precondition_fact const1 2 3 4 5 6) ; ?last_val tendría el número 6 asignado.
(precondition_fact const1 2 x) ; ?last_val tendría el símbolo x asignado.
(precondition_fact const1 2 3 "several words")
    ; en este caso ?last_val tendría el string "several words" asignado.
```

Este esquema genérico funciona para crear reglas de CLIPS sencillas, sin embargo hay aún un par de puntos donde se puede mejorar: 1) el lenguaje de CLIPS permite construir precondiciones más complejas, y 2) se desea que las reglas sean fáciles de diseñar por los usuarios.

Por ejemplo, sabiendo qué tipo de regla se está creando, no sería necesario que el usuario incluyera todas las precondiciones estándar que necesita; al momento de generar el código de CLIPS esas precondiciones podrían ser agregadas por el programa. De la misma manera, no debería ser necesario especificar manualmente todos los campos de una tarea. Mientras menos *verbosos* tengan que ser los diagramas, más fácil será diseñar los planes. Con este fin, se han creado distintos tipos de lugares y transiciones que simplifican el diseño.

4.1.1. Lugares para la planeación.

Un diagrama de redes de Petri sirve para modelar cambios en un sistema. Las reglas de planeación generan cambios en la representación actual del mundo. Por esta razón tiene sentido modelarlas con redes de Petri, donde los lugares corresponden con la información del sistema (los hechos), mientras que las transiciones corresponden con las reglas que accionan esos cambios.

Para facilitar el diseño de las reglas de planeación descritas en este trabajo se han creado distintos tipos de lugares, que a la vez servirían para diferenciar gráficamente distintos tipos de hechos, y permitirían agregar expresividad a las precondiciones de las reglas. Distintos lugares tendrían un color distinto en la herramienta gráfica, además de que tendría restricciones distintas respecto a con qué otros nodos se pueden conectar, o a qué tipos de reglas pueden pertenecer. Estos lugares se describen a continuación.

Fact Place. Son los lugares presentados hasta ahora, sus etiquetas son similares a enunciados de lógica de primer orden.

Structured Fact Place. Estos lugares corresponden con los hechos estructurados de CLIPS. Su sintaxis es similar a la de los lugares `fact`, con la diferencia de que todos los campos deben ser nombrados anteponiendo un símbolo que corresponde con el nombre del campo, seguido de dos puntos (:) y finalmente, de la constante o variable que corresponde con ese argumento. Por ejemplo: `waiting(symbol:primitive_task_symbol)`

Task Place. Representa una tarea. Para facilitar el diseño de los planes, la etiqueta de este hecho contiene el `action_type` de la tarea y los parámetros de la tarea como argumentos del enunciado de FOL. Los campos `plan`, `step` y `parent` son llenados con la información de la tarea que los genera, como se describe en el apartado de este capítulo llamado Reglas de planeación. Cuando se crea una regla, el lugar en las precondiciones correspondiente con la tarea a la que pertenece dicha regla sería creado automáticamente. Al crear una tarea en el explorador del proyecto, el nombre de la tarea debería ser también un enunciado de FOL, pues sería la etiqueta que tendría esta precondición.

Command Place. Se utilizan para hacer llamados a procedimientos remotos en las reglas. La etiqueta de este nombre corresponde con el nombre del comando que se llama, y sus argumentos son: 1) un string con los parámetros que el comando necesita, 2) un símbolo para identificar que este comando está esperando una respuesta, y opcionalmente 3) y 4) un entero que representa el timeout de este comando (si no, se utilizaría un timeout predeterminado), y un entero que representa el número de intentos de realizar el comando en caso de que falle o se agote el tiempo del timeout.

Task Status Place. Sirve para establecer el estado final de una tarea y que el planeador proceda a la actualización del plan, o bien, para señalar precondiciones de una regla de finalización. La etiqueta del lugar es `task_status` seguida de paréntesis conteniendo el estado final de la tarea, es decir, alguno de los símbolos: `successful` o `failed`. Cuando funge como precondición puede tener un comodín en lugar del símbolo correspondiente con el estado final.

Function Place. CLIPS permite realizar algunas funciones estándar en los tipos de datos que maneja, como operaciones aritméticas o con cadenas de caracteres, y usarlos para restringir las precondiciones de una regla. La etiqueta de estos hechos inicia con las letras `fnc`, y le siguen paréntesis con el nombre de la función que se quiere utilizar, los argumentos de la función separados por comas, y finalmente una variable que alojaría el resultado. Esta variable podría después usarse en otras funciones, para tener funciones anidadas, en lugares de comparación, o en los argumentos de otras precondiciones. Para más información sobre las funciones de CLIPS puede consultar el manual de referencia [26].

Function Call Place. No confundir con el lugar anterior. Estas son funciones que serían llamadas en los efectos de la regla, pero que no regresan un valor que sería sustituido en otro hecho, comparación o función, sino que su ejecución es lo que es importante. Su uso más evidente es para poder imprimir cosas a consola, aunque podría utilizarse cualquier función. La sintaxis de estos lugares incluye sólo el símbolo `fncCall`, seguido de paréntesis, entre los cuales estarían los parámetros de la función, separados por comas. Los parámetros, de igual forma, pueden ser solamente constantes o variables aterrizadas en las precondiciones. Estas funciones no deberían suponer ningún orden particular en el flujo de ejecución, pues los diagramas de redes de Petri no especifican ningún orden entre los hechos que produce una transición.

Comparison Place. Además de los patrones de hechos, CLIPS permite realizar comparaciones entre valores. A diferencia de los lugares anteriores, donde la función arrojaría un resutlado que podría ser numérico o cualquier otro tipo de dato aceptado por el intérprete de CLIPS, las comparaciones solamente validarían una restricción adicional. Su etiqueta consiste en las letras `cmp`, seguidas de paréntesis con tres argumentos: el primero es la función de comparación, que puede ser alguna de: `>`, `>=`, `<`, `<=`, `=`, `<>`, `eq` o `neq`. Los argumentos segundo y tercero corresponden con los dos valores que estarían comparándose. La diferencia entre los comparadores `=` y `eq`, es que el primero se utilizaría para comparar valores numéricos, y el segundo serviría para comparar símbolos o cadenas de caracteres. Lo mismo sucede con los comparadores `<>` y `neq`.

OR Place. Además de los patrones, las funciones y las comparaciones, CLIPS permite establecer disyunciones en las precondiciones. Esto se logra conectando dos o más transiciones a un lugar de tipo OR, que a su vez se conectaría a la transición de precondiciones de la regla. Una condición NOR se puede lograr conectando el lugar de tipo OR con un arco inhibidor.

NAND Place. De la misma manera, se pueden crear conjunciones de precondiciones. De manera predeterminada, una transición representan la conjunción de todas las precondiciones que se conecten a ella; sin embargo, para agregar una negación a un conjunto de precondiciones, sería necesario agregar una transición conectada a un lugar de tipo NAND, que se conectaría con un inhibidor a la transición de las precondiciones de la regla.

4.1.2. Transiciones para la planeación.

De igual manera, se han definido distintos tipos de transiciones para diseñar las reglas. A diferencia de los lugares, el usuario no tendría que crearlas directamente, sino que serían creadas dependiendo de los tipos de lugares que se incluyan en la regla. Los tipos de transiciones que existen se describen a continuación.

Rule Transition. Esta transición corresponde con la transición de una regla, como se utilizó en el ejemplo anterior. Su función es separar las precondiciones de las acciones de la regla. Debe haber exactamente una transición de este tipo en cada diagrama de redes de Petri que corresponda con una regla de planeación. Dependiendo del tipo de regla del que se trate, ésta transición puede tener conectado (o estar conectada a) casi cualquier lugar. Sólo no puede estar conectada a un lugar de tipo `OR`, `NAND`, `Function` ni `Comparison`.

And Transition. Éste tipo de transición sería la que se conecta a un lugar OR o a un lugar NAND. Sirve para agregar una conjunción dentro de una negación, o bien, una conjunción dentro de una disyunción. Puede haber tantas como el usuario necesite para diseñar las precondiciones de las reglas.

Sequence Transition. Las reglas de tipo dexec pueden generar nuevas tareas. Todas las tareas creadas en una regla tienen en su campo `step` la jerarquía y el orden relativo con respecto a las otras tareas de la misma jerarquía. Éste tipo de transiciones sirve para asignar ese orden relativo a las tareas creadas. Así, las tareas generadas por una regla se representarían como una secuencia de lugares de tareas conectados a través de transiciones de tipo `sequence`. En la figura 4-1 se puede apreciar una regla de dexec que utiliza estas transiciones (blancas) para definir el orden relativo de las tareas creadas.

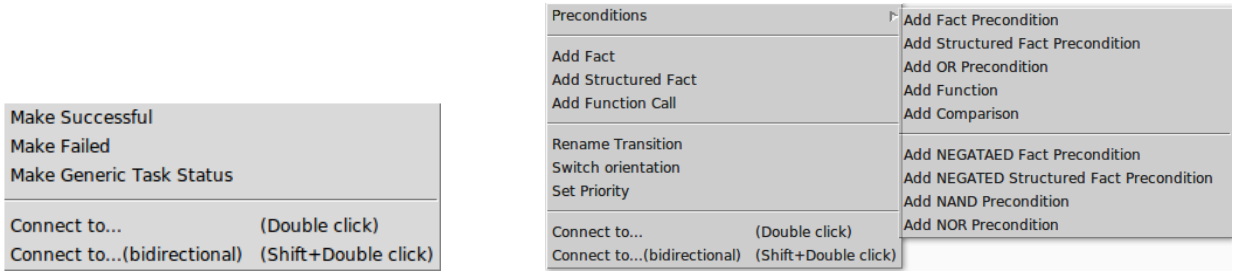
4.1.3. Reglas de planeación.

Los tipos de reglas que se pueden generar en la herramienta corresponden exactamente con los tipos de reglas de planeación descritas en el capítulo anterior: reglas de dexec, reglas de cancelación y reglas de finalización.

Al crear una regla en el explorador del proyecto, automáticamente son añadidos una transición tipo `Rule` y un lugar de tarea conectado con un arco bidireccinoal. Adicionalmente, las reglas de finalización incluirán una precondición de tipo `Task Status`.

Para añadir o modificar elementos de la regla, el usuario tendría que hacer click derecho sobre algún elemento, y seleccionar la opción adecuada. Los lugares de tarea cuya etiqueta corresponda con el nombre de la tarea a la que la regla pertenece (el lugar que se agrega automáticamente) no podrían modificarse a través del diagrama, sino a través del explorador de proyecto. Esto es así, porque ese lugar se utiliza en todas las reglas de la tarea, por lo tanto, tiene sentido hacer modificaciones a nivel de la tarea, y no a nivel de la regla.

La figura 4-3 muestra los menús contextuales de un lugar de tipo `Task Status` cuando es una precondición, y de una transición de tipo `Rule`. Todos los elementos de un diagrama de Petri tienen opciones comunes al tipo de elemento genérico que sean (lugares, transiciones, arcos), que se muestran al final del menú contextual; y cada tipo de elemento en concreto podría tener además opciones específicas.



(a) Menú contextual de un lugar `Task Status`

(b) Menú contextual de una transición `Rule`

Figura 4-3: Menús contextuales de la herramienta de PNPDT.

El menú del lugar de tipo `Task Status` es bastante sencillo, sólo permite especificar si la tarea debió ser exitosa o fallida, o bien, que no importa el estado final, señalado con un comodín sencillo. En el menú de la transición de la regla se pueden observar 2 secciones distitnas (además de las opciones genéricas de una transición): la primera sirve para añadir precondiciones a la regla y la segunda sirve para crear hechos como efectos de la regla (hechos nuevos que serían creados). En las reglas de planeación de tipo dexec se incluye una tercera sección que corresponde con opciones para agregar una secuencia de lugares de tareas a los efectos de la regla.

En los menús contextuales solamente hay opciones para agregar nuevos lugares. Las transiciones necesarias serían agregadas automáticamente cuando se cree un tipo particular de lugar que las requiera, como en los casos de las estructuras OR o NAND, y de las tareas generadas en las reglas. De la misma manera, cuando se agregue una precondición que involucre una negación, esa precondición sería conectada a través de un arco inhibidor.

Note que la estructura de los planes permite generar distintas ramas, que corresponderían con tareas de la misma jerarquía, de modo que una misma transición podría tener arcos a distintos lugares de tareas; sin embargo, una transición `Sequence` no puede tener más de un arco entrante. Si se requiriera sincronización de tareas, se necesitaría agregar un nivel jerárquico que incluya los dos caminos de tareas que deberían sincronizarse.

La figura 4-4 muestra un ejemplo de una regla de ejecución que utiliza una construcción NOR en las precondiciones, y genera dos secuencias de tareas a realizar.

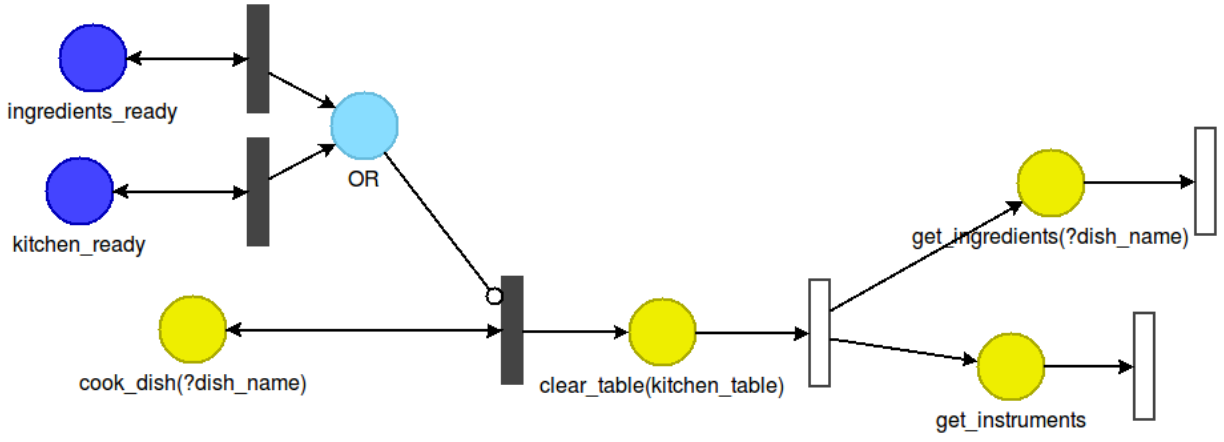


Figura 4-4: Ejemplo de regla con construcción NOR y generación de nuevas tareas.

4.2. Conversión a reglas de CLIPS

Una vez diseñadas las reglas de planeación, sería necesario generar el código de CLIPS equivalente para que el motor de planeación pueda ejcutarlas. La figura 4-5 muestra una plantilla de cómo es la estructura general del código generado de una regla. Los datos de la regla como el nombre de la tarea, el nombre de la regla y el salience vendrían de la información de la tarea a la que pertenecen, y la prioridad de la transición de la regla respectivamente. Evidentemente, una regla podría no tener alguno de esos elementos, como llamadas de comandos, creación de nuevos hechos o eliminación de hechos existentes.

```
(rule <task_name>-<rule_name>
        <salience>
        <preconditions>
        =>
        (retract <vars>)
        (assert
                <new_facts>
                <task facts>
        )
        <commands>
        ; (send-command "example_command" symbol "params"
                                <optional_ timeout> <optional_ attempts> )
)
```

Figura 4-5: Plantilla del código generado por una regla de planeación.

Inicialmente podría parecer una tarea sencilla generar el código de CLIPS, pues las etiquetas de los lugares serían prácticamente los hechos de CLIPS, sólo con un formato distinto. Sin embargo, varios problemas surgen si los hechos se agregan en un orden arbitrario, pues es necesario que las variables que se utilicen en lugares de tipo `Function` y `Comparison` estén aterrizadas, es decir, que aparezcan en otra precondición en donde se le asigne un valor, que debería ser procesada antes que estos lugares.

Así mismo, cuando existe una estructura (como una disyunción), dentro de una negación, las variables nuevas que aparezcan dentro de la negación serían unificadas solamente para el contexto de evaluación de la negación; es decir, las variables asignadas dentro de un elemento condicional `NOT`, no pueden ser utilizadas en precondiciones que estén después de ella, aunque tengan el mismo nombre. Esto puede producir resultados inesperados y contraintuitivos. En cambio, si la precondición no negada que incluye a la variable se encuentra antes de la negación, la variable ya estaría unificada al momento de evaluar la negación.

Los lugares conectados a una transicón en un diagrama de redes de Petri no tienen un orden asignado, y sería necesario evitar este tipo de conflictos. Por esta razón, el procedimiento para extraer las precondiciones procesa todas las precondiciones que sean positivas (que estén conectadas con un arco normal), y dejaría al último las precondiciones negativas (conectadas con un arco inhibidor).

El algoritmo de generación de código de CLIPS, mostrado en la figura **??**, obtiene una descripción estructurada de las precondiciones y de los efectos de las reglas, para generar el código. Esta descripción estructurada consiste básicamente en listas anidadas, donde el primer elemento de cada lista es un `string` identificador del tipo de elemento a procesar.

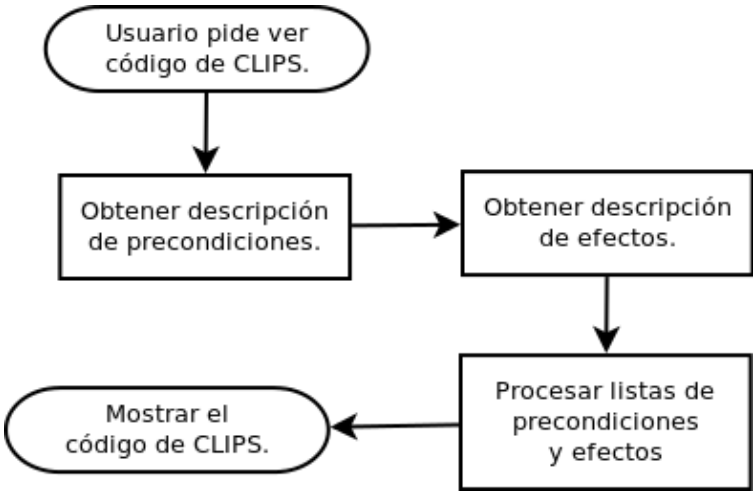


Figura 4-6: Algoritmo de generación de código de CLIPS.

La figura 4-7 muestra el código de CLIPS generado para el ejemplo de regla de planeación mostrada en la figura 4-4. Las precondiciones de esta regla fueron generadas a partir de una descripción similar a la mostrada a continueación. Los efectos o acciones se habrían generado a partir de una estructura similar.

```
[ ['task', 'cook_dish', ['?dish_name']],
  ['active_task'], ['not', ['cancel_active_tasks']],
  ['not', ['task_status', '?']],
  ['not', ['or', [['fact', 'ingredients_ready', []], ['fact', 'kitchen_ready', []] ] ] ]
]
```

CLIPS code - cook_dish - get_ready

```
(defrule cook_dish-get_ready
        (task (id ?pnpdt_task__) (plan ?pnpdt_planName__) (action_type cook_dish) (params ?dish_name) (step $?pnpdt_steps__) )
        (active_task ?pnpdt_task__)
        (not
                (cancel_active_tasks)
        )
        (not
                (task_status ?pnpdt_task__ ?)
        )
        (not
                (or
                        (ingredients_ready)
                        (kitchen_ready)
                )
        )
        =>
        (assert
                (task (plan ?pnpdt_planName__) (action_type clear_table) (params kitchen_table) (step 1 $?pnpdt_steps__) (parent ?pnpdt_task__) )
                (task (plan ?pnpdt_planName__) (action_type get_ingredients) (params ?dish_name) (step 2 $?pnpdt_steps__) (parent ?pnpdt_task__) )
                (task (plan ?pnpdt_planName__) (action_type get_instruments) (params "") (step 2 $?pnpdt_steps__) (parent ?pnpdt_task__) )
        )
)
```

Copy to clipboard Ok

Figura 4-7: Código de CLIPS generado para el ejemplo de la figura 4-4.

Note que además de las variables y los hechos introducidos por el usuario en el diagrama, la herramienta ha generado otras variables y otros hechos para completar la especificación de los planes presentada en el capítulo anterior. Las variables generadas por la herramienta, tienen el formato: `?pnpdt_<var_name>__`. En este caso corresponden con el identificador de la tarea, el nombre del plan, el padre de la tarea y el campo `step` de la tarea a la que pertenece la regla. Ésta es justamente la manera en que la herramienta PNPDT simplifica el trabajo del diseñador de los planes.

Las variables del campo `step` están divididas en una variable sencilla, seguida de una variable múltiple. Esto es así, por que dependiendo del tipo de regla, las tareas generadas serían heramans o hijas de la tarea que las genera. Para crear una tarea hermana, con un orden relativo a la tarea que la genera, habría que restar de la primera variable un número entero. No tendría mucho sentido sumarle, pues la tarea no debería conocer el contexto de ejecución, y por lo tanto, no sabría qué tareas hay después.

Así, en las reglas de descomposición, obviamente, se generarían tareas hijas, pues esa es su principal función. Lo mismo ocurriría en las de finalización, donde las tareas generadas harían el trabajo de un plan de verificación. En las reglas de ejecución, en cambio, fungirían como planes correctivos, y serían generadas como tareas hermanas, con una secuencia *anterior* al paso de la tarea actual. Las reglas de cancelación no pueden generar tareas en esta herramienta.

Cuando un lugar que es una precondición *no* tiene un arco de regreso desde la transición, significa que debería ser eliminado. De manera similar la herramienta genera también el código para identificar la dirección del hecho y eliminarlo.

CAPÍTULO

5

PRUEBAS Y RESULTADOS

El desempeño de un planeador de acciones como el desarrollado en esta tesis es un tanto difícil de cuantificar, pues se trata de procesos simbólicos con una gran parte de diseño humano que agrega una componente subjetiva por naturaleza (y desconocida *a priori*, pues los planes no forman parte del motor de planeación, sino que una persona los generaría para resolver un problema particular).

Además, como se mencionó anteriormente, la eficiencia computacional no es una de las principales preocupaciones de este trabajo, sino que los planes que se puedan diseñar para usarse con él deberían poder manejar situaciones que otros planeadores no podrían, o con los cuales sería más complicado hacerlo; o por lo menos facilitar la implementación de planes equivalentes en otros planeadores.

La manera en que se reportan los resultados de esta investigación, entonces, consiste en presentar un conjunto de casos o escenarios que exhiben determinadas circunstancias que con un diseño de plan adecuado el motor de planeación descrito puede manejar. En la primera sección de este capítulo se describen las pruebas realizadas, junto con algunos resultados particulares. Posteriormente se comentan algunas observaciones realizadas sobre el planeador y la especificación de los planes durante estos experimentos.

5.1. Pruebas

5.1.1. Mundo de cubos

Un problema de planeación icónico, utilizado como referencia para probar la efectividad de un planeador, es el de apilar un conjunto de cubos de acuerdo a una configuración objetivo. La idea es que sin importar cuál sea la configuración inicial, la pila de cubos especificada como obejtivo se encuentre en el estado final. Para este fin, cada cubo puede estar en la mesa o encima de otro cubo, y el robot puede ejecutar las tareas de analizar el ambiente para identidicar las pilas de cubos, tomar un cubo que no tenga otro cubo encima, y poner un cubo encima de otro o sobre la mesa.

Este problema ya ha sido resuelto por muchos planeadores; sin embargo, la mayoría toma en cuenta solamente el resultado de una simulación, o la formulación de un plan completo, sin llevar a cabo su ejecución. En la práctica, el llamado *reality gap* no permite que sea una tarea sin dificultad para un robot. Las fallas de los actuadores, aunado a las limitantes físicas del robot y del espacio en que debe desarrollar la prueba dificultan muchas veces que su ejecución sea exitosa.

Es precisamente la robustez de un planeador y del plan a ejecutar lo que permite que la tarea se lleve a cabo correctamente. En este caso, los planes correctivos y de verificación habilitan al robot para realizar la planeación de acciones necesaria para conseguir el objetivo, así como para detectar situaciones de error en las que necesita ayuda para resolver el problema o detectar fallas en las acciones de los actuadores y actuar en consecuencia.

Los planes para esta prueba fueron diseñados directamente en código de CLIPS (sin utilizar la herramienta gráfica PNPDT). Algunas tareas implementaron ciertas heurísticas que le permitían al robot tomar decisiones particulares, como por ejemplo, cuando se quisiera tomar un cubo, se utilizaba la representación del mundo y del objetivo actual para decidir con qué brazo tomarlo, asumiendo que ambos estuviera habilitados y el cubo en cuestión estuviera a su alcance.

La figura 5-1 muestra parte del contenido de la bitácora de ejecución de esta prueba. Para facilitar su lectura se eliminaron muchas líneas que no aportaban mucha información respecto a la planeación, como la mayoría de la emisión y recepción de comandos y respuestas, respectivamente, la limpieza de hechos que ya no son utilizados, mensajes de éxito de algunas tareas y otros mensajes del proceso de selección de tareas realizado por el planeador.

```
1  ==> f-4     (can_run_in_parallel la_goto ra_goto)
2  ==> f-5     (task (id gen1) (plan "Cubes plan") (action_type cubes_plan) (params "") (step 0) (parent nil))
3  ==> f-7     (cubes_goal green blue)
4  INFO: Task activated: (task (id gen1) (plan "Cubes plan") (action_type cubes_plan) (step 0) (params "") )
5  INFO: Task activated: (task (id gen2) (plan "Cubes plan") (action_type spg_say) (step 1 0) (params "I'm going to execute
   the plan:" "Cubes plan") )
6  INFO: Task activated: (task (id gen3) (plan "Cubes plan") (action_type cubes_do_cubes) (step 1 0) (params "") )
7  INFO: Task activated: (task (id gen4) (plan "Cubes plan") (action_type cubes_get_info) (step 1 1 0) (params "") )
8  INFO: Task activated: (task (id gen5) (plan "Cubes plan") (action_type arms_goto) (step 1 1 1 0) (params "navigation") )
9  INFO: Task activated: (task (id gen7) (plan "Cubes plan") (action_type ra_goto) (step 1 1 1 1 0) (params "navigation") )
10 INFO: Task activated: (task (id gen6) (plan "Cubes plan") (action_type la_goto) (step 1 1 1 1 0) (params "navigation") )
11
12 INFO: Sent command: 'detectcubes' - id: 4 - timeout: 10000ms - attempts: 1 - time: 129.51 - params: all
13 INFO: Answer received: 'detectcubes' - id: 4 - successful: 1 - response: green 0.19655108698 0.202005107074246 0.
   636161451165 blue 0.51766167278 0.0350056267930247 0.565450185865 orange 0.77321793428 -0.526993814883361 0.489483658665
14
15 INFO: Successful task of plan 'Cubes plan' with action_type: 'cubes_get_info' and params: '"" was deleted with no other
   tasks of same hierarchy. Parent task with action_type: 'cubes_do_cubes' and params: '"" is now active.
16 INFO: Task activated: (task (id gen8) (plan "Cubes plan") (action_type cubes_stack_cubes) (step 1 1 0) (params "") )
17 INFO: Task activated: (task (id gen9) (plan "Cubes plan") (action_type cubes_move_cube) (step 1 1 1 0) (params blue green
   ) )
18 INFO: Task activated: (task (id gen10) (plan "Cubes plan") (action_type spg_say) (step 1 1 1 1 0) (params "I will move
   cube " blue " on top of cube " green) )
19 INFO: Task activated: (task (id gen11) (plan "Cubes plan") (action_type cubes_take_cube) (step 1 1 1 1 0) (params blue) )
20 INFO: Task activated: (task (id gen12) (plan "Cubes plan") (action_type spg_say) (step 1 1 1 1 1 0) (params "I couldn't
   take the cube " blue ". It's probably to far for me to get. I will try again.") )
21 INFO: Task activated: (task (id gen13) (plan "Cubes plan") (action_type cubes_get_info) (step 1 1 1 1 1 0) (params "") )
22 INFO: Task activated: (task (id gen14) (plan "Cubes plan") (action_type cubes_delete_info) (step 1 1 1 1 1 1 0) (params
   "") )
23 INFO: Successful task of plan 'Cubes plan' with action_type: 'cubes_get_info' and params: '"" was deleted with no other
   tasks of same hierarchy. Parent task with action_type: 'cubes_take_cube' and params: 'blue' is now active.
24
25 INFO: Sent command: 'takexyz' - id: 10 - timeout: 60000ms - attempts: 2 - time: 138.49 - params: left 0.13911970233 0.
   0751391189476418 0.658099927165
26 INFO: Answer received: 'takexyz' - id: 10 - successful: 1 - response: left 0.13911970233 0.0751391189476418 0.
   658099927165
27
28 INFO: Task activated: (task (id gen15) (plan "Cubes plan") (action_type cubes_get_info) (step 1 1 1 1 1 0) (params "") )
29 INFO: Task activated: (task (id gen20) (plan "Cubes plan") (action_type cubes_put_cube) (step 1 1 1 1 0) (params blue
   green) )
30 INFO: Task activated: (task (id gen21) (plan "Cubes plan") (action_type cubes_get_info) (step 1 1 1 1 1 0) (params "") )
31 INFO: Successful task of plan 'Cubes plan' with action_type: 'cubes_put_cube' and params: 'blue green' was deleted with
   no other tasks of same hierarchy. Parent task with action_type: 'cubes_move_cube' and params: 'blue green' is now active.
32 INFO: Successful task of plan 'Cubes plan' with action_type: 'cubes_move_cube' and params: 'blue green' was deleted with
   no other tasks of same hierarchy. Parent task with action_type: 'cubes_stack_cubes' and params: '"" is now active.
33 INFO: Successful task of plan 'Cubes plan' with action_type: 'cubes_stack_cubes' and params: '"" was deleted with no
   other tasks of same hierarchy. Parent task with action_type: 'cubes_do_cubes' and params: '"" is now active.
34 INFO: Successful task of plan 'Cubes plan' with action_type: 'cubes_do_cubes' and params: '"" was deleted with no other
   tasks of same hierarchy. Parent task with action_type: 'cubes_plan' and params: '"" is now active.
35 INFO: Top level task of plan 'Cubes plan' with action_type: 'cubes_plan' and params: '"" succeeded!
36 INFO: Task activated: (task (id gen26) (plan "Cubes plan") (action_type spg_say) (step 0) (params "I have finished the
   task " "Cubes plan") )
```

Figura 5-1: Bitácora de ejecución de la prueba de los cubos.

Los hechos iniciales (mostrados en las primeras líneas de la figura) incluían la tarea que debía ejecutarse y el objetivo que se quería lograr, que en este caso era formar una pila que tuviera al cubo verde como base, y el cubo azul estaría sobre él. Además de estos hechos existían otros que indicaban el estado inicial de los brazos, y otros hechos necesarios para la ejecución del intérprete de CLIPS.

Note que la tarea `cubes_get_info` se activa varias veces, en las líneas 7, 21, 28 y 30. Sin embargo, de acuerdo al contexto de ejecución actual, la descomposición de tareas se hace de manera distinta, la segunda vez llamó a una rutina para eliminar la información actual del mundo, antes de volverla a obtener, mientras que la primera vez, al no tener información sobre el mundo, no fue necesario eliminar nada, y por lo tanto, no se activó la tarea `cubes_delete_info`. Las otras activaciones corresponden con los planes de verificación de las tareas `cubes_take_cube` y `cubes_put_cube`.

En esta ejecución, deliberadamente se colocó el cubo azul en una posición inalcanzable por el robot. En la línea 20 se puede observar que el plan fue capaz de detectar que no alcanzaba el cubo que pretendía tomar, y procedió a analizar el ambiente nuevamente, en espera de que una persona lo acercara. De la misma manera el plan podría haber estado diseñado para poner en marcha cualquier otro curso de acción para corregir esa situación, aunque las limitantes físicas del robot limitan también los planes que puede realizar.

Finalmente, podrá observar que todas las tareas tienen números 1 en cada nivel jerárquico nuevo, a pesar de que la especificación del plan permite generar tareas secuenciales en un sólo paso, utilizando números consecutivos en la posición de más a la izquierda. El problema de hacerlo así es que si el plan falla, no sería posible determinar en qué parte del plan completo falló. Incluso si las tareas tienen ciertos efectos que deberían verse reflejados después de su ejecución. Si no se obsevan esos efectos cuando la tarea padre obtiene el control de ejecución, podría deberse a que la tarea falló, o bien a que el plan falló antes de tratar de ejecutar esa tarea.

Para evitar este inconveniente, el plan se diseñó de forma tal que cada tarea generaría una tarea hija a la vez, y al obtener la tarea padre nuevamente el control de ejecución generaría otra, o declararía que la tarea ha terminado, según corresponda. En la figura se puede apreciar en las líneas 5 y 6, que ambas tareas pertenecen a la misma jerarquía, sin embargo, una fue creada y ejecutada antes que la otra.

5.2. Observaciones generales

Todas las reglas de planeación contienen un conjunto de precondiciones mínimas necesarias, dependiendo del tipo de regla de planeación de que se trate, que habilitan las reglas de planeación de una tarea. Es decir, el hecho de que una tarea esté activa para el planeador, simplemente corresponde con que esas precondiciones mínimas se cumplen. Sin embargo, esas precondiciones son también hechos, como cualquier otro que defina el usuario, y el intérprete de CLIPS (la máquina de inferencias) tendría que realizar una búsqueda incluyendo todas las reglas que estén registradas en el sistema, independientemente de si una tarea está activa o no.

Por otro lado, el control de ejecución de una tarea en muchos casos estará determinado por hechos que reflejen el estado actual del mundo, o bien, cuando haya tareas que no tienen un efecto tangible en el ambiente (como la tarea de emitir un mensaje con el generador de voz), por hechos que funcionan como banderas para que la tarea sepa en qué paso de la ejecución se encuentra.

Esto quiere decir que al final, el motor de ejecución realiza en realidad una búsqueda greedy en el espacio-estado, en donde un estado corresponde con una configuración de la base de conocimiento (los hechos que existan en un momento dado), y cada transición a otro estaría determinada por el cambio que produzca (en la base de conocimiento) la ejecución de una regla habilitada en ese estado. En lugar de que el usuario defina explícitamente las transciones entre un estado y otro, éstas serían determinadas por la máquina de inferencias. No obstante, la conceptualización de las tareas y la forma en que se estructura y ejecuta un plan se realiza en un nivel de abstracción más alto que facilita el diseño de los planes.

El utilizar hechos como precondiciones en las reglas para controlar el flujo de ejecución es un tanto difícil de lograr. Si no se tiene cuidado, pueden estar habilitadas simultáneamente reglas que no deberían estarlo, de modo que el flujo de ejecución sería distinto al esperado. Por esta razón es importante mantener en las reglas una descripción explícita y lo más completa posible de las precondiciones (circunstancias) necesarias para su activación, así como también criterios explícitos y bien definidos para determinar el éxito o fracaso de una tarea.

De la misma manera, conforme aumenta de tamaño y complejidad un plan, es más probable que se cometan errores al escribir las reglas. Los más sencillos de identificar son los errores de sintaxis; pero ocurren otros más complicados, como confundir las variables que se están utilizando, o repetir los nombres de las reglas, que efectivamente producen la eliminación de la primera definición. Estos últimos son más frecuentes cuando se realizan operaciones de copiado y pegado de precondiciones y reglas, y son más difíciles de identificar.

Por esta razón, el uso de una herramienta gráfica como la desarrollada en esta tesis, que pueda generar código automáticamente y realizar algunas validaciones para no cometer estos errores, sería una herramienta muy valiosa para agilizar el desarrollo y la detección y corrección de errores. Sin embargo, la expresividad de la herramienta gráfica no es la misma que la del lenguaje completo de CLIPS, por lo que en ocasiones sería necesario diseñar los planes directamente en el lenguaje de programación. Además, si bien es un instrumento más intuitivo que un lenguaje de programación, aún existe una curva de aprendizaje para construir las reglas utilizando los hechos que utiliza el planeador para ejecutar los planes.

Como se mencionó anteriormente, el planeador implementa una búsqueda *greedy*, es decir que ejecuta la siguiente tarea inmediatamente, antes de generar el plan completo. Esto es debido a que los resultados de las acciones no siempre son los mismos, y es necesario planear de acuerdo a las circunstancias reales y no a las del resultado de una simulación o una suposición de que las acciones

se llevan a cabo correctamente. Desafortunadamente, esto implica que en ocasiones podría ejecutarse un plan subóptimo, por ejemplo, en la prueba de mundos de cubos, al decidir con qué brazo robótico tomar un cubo, asumiendo que se encuentre al alcance de ambos brazos, podría tomarlo con uno que no alcanza a poner el cubo en la pila que se está formando.

El motor de planeación no implementa una búsqueda *completa* antes de ejecutar el plan, ni cuenta con un razonamiento sobre las tareas que tiene que realizar, por lo tanto, la selección adecuada de las tareas debe ser codificada en heurísticas a través de las precondiciones de las reglas. En el ejemplo de con qué brazo tomar un cubo, se selecciona un brazo que tenga al alcance la pila que se está formando, cuando se trate de tomar un cubo que pertenece a la pila objetivo, o usar el otro brazo, cuando no sea así.

Por otra parte, el hecho de que la representación de las tareas y su estructura en los planes se realice utilizando hechos de la base de conocimiento, sí permitiría cierta manipulación de los planes. Por ejemplo, si hay una tarea registrada para traer una bebida a una persona, y la persona le comunica al robot que ya no quiere esa bebida. Podría existir un plan en que al recibir ese comando de voz y se genere la tarea "`cancel_get_drink`", en la regla de planeación se busque una tarea de tipo `get_drink` y se marque esa tarea como exitosa, o se elimine el hecho correspondiente, según se quiera manejar.

Esta misma razón, además de que el sistema esté implementado como un sistema experto basado en reglas, permitiría que puedan existir distintos planes para ejecutarse, codificados en hechos de tareas con distintas prioridades. Y que en consecuencia se pueda suspender una tarea para ejecutar otra más prioritaria. Por ejemplo, si un robot de servicio tiene un temporizador para recordar a una persona tomar una medicina a una cierta hora, en un momento dado el robot podría suspender lo que está haciendo para recordarle a la persona que debe tomar su medicina, y después continuar con la tarea que estaba desempeñando.

CAPÍTULO

6

CONCLUSIONES Y TRABAJO A FUTURO

Al momento de elegir un planeador sobre otro, por la naturaleza subjetiva que involucra la descripción de los planes, las acciones y el ambiente en el que se desarrollan, no se puede considerar absolutamente una alternativa mejor que otra. En cambio, habría que ponderar qué características presenta cada uno, que podrían en algún momento otorgarle más peso a la decisión de utilizar uno u otro, tales como la manera en que pueden considerar o administrar recursos necesarios para la planeación, o la facilidad de uso, entre otros. En esta sección se describen las implicaciones de utilizar el planeador presentado en los capítulos anteriores, cincundando las observaciones hechas al presentar los resultados.

La conceptualización de tener tareas activas y no activas permite separar y estructurar la solución a un problema complejo, y por lo tanto, diseñar planes más adecuados a las necesidades reales. De la misma manera, la necesidad de especificar explícitamente criterios de éxito o fracaso de una tarea, así como de las precondiciones en cada regla para controlar el flujo de ejecución ayuda a diseñar una solución efectiva a un problema de planeación, aunque no deja de ser un proceso complicado.

Así como en los planeadores tipo STRIPS resulta difícil describir acertadamente el mundo y los efectos de las acciones sobre él para conseguir un plan adecuado, en este caso, de manera similar, es complicado representar una situación particular en las precondiciones de cada regla, a fin de que se habilite solamente cuando debe. En muchas ocasiones ocurre que una regla es habilitada cuando no debería, o al contrario, que una regla que el usuario considera debería estar habilitada en un momento determinado, no lo está.

La interfaz gráfica no eliminaría por completo la curva de aprendizaje para poder diseñar un plan, pero sí la simplificaría, a cambio de una pequeña reducción en el espacio de planes que se pueden generar. Además puede ayudar a eliminar errores de sintaxis, e incluso (con desarrollo adicional al presentado en esta tesis, que se discutirá más adelante en este capítulo) realizar algunos análisis o simulaciones para revisar que cada regla se habilite cuando deba, antes de ejecutar el plan en el ambiente real.

La representación de los planes a través de hechos permite hasta cierto punto la manipulación de los planes, por ejemplo, una tarea puede cancelar a otra, o marcarla como exitosa o fallida, según convenga. Esto, aunado a la ejecución basada en reglas permite que puedan existir distintos planes, que serían ejecutados de acuerdo a una especificación de prioridad, es decir que el planeador soporta la ejecución multi-tarea.

Además, el sistema experto basado en reglas propicia un diseño de planes más orientado a objetivos que a procedimientos. Un diseño de planes orientado a objetivos, más que a procedimientos, sería más robusto para considerar escenarios iniciales distintos. La especificación de un conjunto de precondiciones en cada regla permite considerar situaciones distintas y realizar la planeación según corresponda.

A la vez permite codificar heurísticas en las reglas a través de precondiciones que defina el usuario, y más aún, permite separar la lógica de ejecución de más de una acción dentro de la misma tarea; un ejemplo muy común es separar la lógica de lo que el robot debe decir, de lo que debe hacer, por ejemplo: al momento de esperar una instrucción, el robot puede repetir cada cierto tiempo un mensaje solicitando una instrucción, mientras que otras reglas se encargarían de procesar las oraciones reconocidas.

Finalmente, es importante notar que para reproducir algunas de las funcionalidades mencionadas en este apartado, en un planeador/ejecutor procedural, como el de las máquinas de estados jerárquicas, requeriría de la implementación de técnicas especializadas para ese fin, que serían difíciles de manejar, tales como programación y ejecución multi-hilo, que debe ser sincronizada o coordinada de alguna forma, o bien, implementar un mecanismo de búsqueda, más o menos equivalente a la máquina de inferencias que incluye el intérprete de CLIPS.

6.1. Trabajo a futuro

Sobre la especificación de los planes y los hechos que utiliza el planeador, se podrían proponer algunos cambios o extensiones que faciliten el diseño de los planes. Por ejemplo: extender el sistema de prioridades para contemplar hechos que indiquen que una tarea puede correr en paralelo con cualquier otra tarea, y hechos que indiquen que una tarea *no* puede ejecutarse en paralelo con otro tipo de tarea. Así, por ejemplo, si se quisiera, se podría especificar que la tarea de generación de voz se puede ejecutar en paralelo con cualquier tarea, menos con ella misma.

El planeador descrito permite interrumpir tareas en cualquier momento de la ejecución cuando se dé de alta una nueva tarea que tenga mayor prioridad. Una alternativa sería que el usuario, diseñador de los planes, defina *checkpoints* en los cuales se pueda interrumpir la ejecución de una tarea.

En la herramienta gráfica también se podría incrementar la funcionalidad, de modo que el paso del diseño a la implementación o ejecución sea mucho más transparente o automático. El sistema actual permite sólo copiar al portapapeles el código correspondiente con una regla, pero se podría implementar un método para exportar en bulto varias reglas o todas las reglas del proyecto.

De manera similar, aún más transparente, se podría pensar en un sistema de comunicación entre el software PNPDT y el planeador, de tal forma que desde la interfaz gráfica se genere automáticamente el código de las reglas, se inserte en el intérprete de CLIPS y se ejecute sin ninguna intervención adicional del usuario. Para esto sería necesario validar que todas las tareas que incluye un determinado plan estén incluidas en el proyecto.

Adicionalmente, se podría implementar una carga modular a la herramienta PNPDT, de modo que conjuntos distintos de reglas se puedan combinar e integrar en distintos proyectos que hagan uso de ellas. Esto se puede lograr simplemente copiando las tareas en cada proyecto, pero no es escalable ni mantenible cuando ocurran cambios en una tarea.

En cuanto a los cambios para asegurar o proveeer una guía para un diseño de planes adecuado, se podrían utilizar las propiedades de las redes de Petri para realizar distintos análisis. Por ejemplo, se podría revisar que una tarea incluya necesariamente una regla que produzca un hecho `task_status`.

Más aún, se puede hacer una simulación para detectar reglas o hechos en conflicto y así corregir el diseño. Es decir, reglas que estén habilitadas cuando no deberían, o hechos que no deberían existir juntos. Se podría hablar de un ciclo diseño-simulación-diseño, que ahorraría mucho tiempo y esfuerzo en probar y corregir errores en el ambiente real.

Como se mencionó anteriormente, en este trabajo no se hace ningún supuesto respecto a cómo debería estar representado el conocimiento a través de los hechos. Sin embargo, el uso de una lógica de descripción con alguna ontología podría ser conveniente para mantener una representación adecuada del ambiente, e inclusive para modelar las tareas, sus recursos y sus efectos, de modo que se pueda realizar un manejo explícito de recursos y determinación automática de tareas que pueden ejecutarse en paralelo.

La interfaz gráfica podría modificarse para especificar los planes de forma tal que ayude a generar los hechos bandera que controlan el flujo de ejecución, y que el usuario se concentre en diseñar los planes de manera más intuitiva, sin preocuparse por esos detalles.

En un punto intermedio, que no requiera de una descripción detallada del ambiente, pero que permita hacer algún tipo de razonamiento para determinar qué acción se debería ejecutar a continuación, se podría extender el esquema de prioridades a uno más genérico que contemple ejecución de funciones o disponibilidad de recursos, tal vez a partir de una lógica de descripción.

Finalmente, se podría considerar ampliar el planeador a un sistema multi-agente, donde a través de un campo adicional en la descripción de la tarea, se indique a qué agente le corresponde ejecutarla.

APÉNDICE

A

SISTEMA EXPERTO - CLIPS

En este apéndice se presenta una pequeña introducción al lenguaje de CLIPS, utilizado para crear sistemas expertos basados en reglas. Para profundizar en las capacidades del lenguaje y su sintaxis, puede consultar la página `http://clipsrules.sourceforge.net/`, donde podrá encontrar información sobre el programa, así como la documentación y los manuales de referencia de CLIPS [26].

Un sistema experto es aquel que contiene, codificada de alguna forma, información sobre un dominio particular, introducida por una persona *experta* para que ayude a los usuarios a resolver un problema utilizando ese conocimiento. Los sistemas expertos surgieron cuando se quisieron resolver problemas que requieren una abstracción de mayor nivel que el de los problemas que usualmente se resolvían en una computadora (de manera procedural y enfocados a aplicaciones numéricas), y que normalmente involucran una representación simbólica, con relaciones y restricciones implícitas en los hechos que se conocen sobre el problema. Usualmente se le llama *mundo* al ambiente real en donde se desarrolla el problema, y se codifican hechos y reglas sobre el mundo.

Los lenguajes procedurales comunes como C, Java, etc. son poco prácticos para representar y resolver este tipo de problemas, por lo que usualmente se utilizan lenguajes que siguen un paradigma distinto, concretamente, los lenguajes de programación lógica como Prolog o CLIPS. Los lenguajes de programación lógica facilitan la representación simbólica de problemas y su manipulación a través de mecanismos de unificación e inferencia. Para esto, incorporan algunos principios como el *closed world assumption*, que dice que si un hecho no está incluido en la base de conocimiento entonces se considera falso; o el manejo del *frame problem*, que significa que cualquier cosa que no esté explícitamente especificado en las reglas u operadores del lenguaje se mantiene sin cambios.

Los intérpretes de estos lenguajes contienen lo que se conoce como una *máquina de inferencias*, que consiste en algoritmos de búsqueda y unificación de variables para establecer correspondencias entre los hechos de la situación actual del mundo, y los hechos y reglas del sistema experto, para encontrar una solución.

CLIPS es una herramienta para desarrollo de sistemas expertos basados en reglas de encadenamiento hacia adelante (*forward chaining*), esto quiere decir que usando los hechos que conoce sobre la situación actual y su conocimiento sobre el mundo, codificado en las reglas de ejecución, genera nuevos hechos. La herramienta de CLIPS consiste en un lenguaje con sintaxis similar a la del lenguaje funcional LISP, y un intérprete que ejecuta las reglas.

En el apéndice C puede consultar la documentación de la implementación del software BBCLIPS, que fue necesaria para conectar un intérprete de CLIPS con nuestro sistema de paso de mensajes y repositorio de variables compartidas *BlackBoard*.

A.1. Elementos básicos de CLIPS

Un programa en CLIPS está compuesto por construcciones de distintos tipos, las más comunes son `deffacts`, que sirve para especificar los hechos inciales de un programa; `deffunction`, utilizada para definir funciones que podrían ser llamadas cuando se disparen las reglas del programa; `deftemplate`, para especificar hechos estructurados (vea el apartado de hechos en esta sección); y `defrule`, que sirve para diseñar las reglas del programa. La figura A-1 contiene un ejemplo de un programa en CLIPS sencillo que contiene unos hechos iniciales, una pequeña función y una regla.

```
1  (deffacts hechos_iniciales
2      (hecho_inicial 1)
3      (hecho_inicial 2)
4      (hecho_inicial 3)
5  )
6
7  (deffunction imprime
8      ($?elementos)
9      (progn$ (?el $?elementos)
10         (printout t ?el)
11     )
12     (printout t crlf)
13 )
14
15 (defrule imprime_hecho_inicial
16     (hecho_inicial ?x)
17     =>
18     (imprime "hecho inicial: " ?x)
19 )
20
```

(a) Programa ejemplo.

```
         CLIPS (V6.24 06/15/06)
CLIPS> (load CLIPS_ex.clp)
Defining deffacts: hechos_iniciales
Defining deffunction: imprime
Defining defrule: imprime_hecho_inicial +j
TRUE
CLIPS> (reset)
CLIPS> (run)
hecho inicial: 3
hecho inicial: 2
hecho inicial: 1
CLIPS>
```

(b) Ejecución del programa ejemplo en una sesión interactiva en consola.

Figura A-1: Ejemplo de programa sencillo en CLIPS.

A.1.1. Tipos de datos

CLIPS provee ocho tipos de datos primitivos que pueden se usados en los hechos: *float*, *integer*, *symbol*, *string*, *external-address*, *fact-address*, *instance-name* e *instance-address*. *float*, *integer* y *string* representan números de tipo flotante, enteros y cadenas de caracteres respectivamente, como es común en otros lenguajes de programación. *instance-name* e *instance-address* son para utilizar la capacidad de CLIPS de utilizar el paradigma orientado a objetos, pero ni estos ni el tipo de dato *external-address* son utilizados en este trabajo.

Un dato de tipo *symbol* es una secuencia de caracteres que empieza con cualquier caracter ASCII imprimible seguido por cero o más caracteres ASCII imprimibles; el símbolo termina cuando encuentra algún: caracter no imprimible (incluyendo espacios, tabuladores, retorno de carros y saltos de línea), comillas dobles, paréntesis abriendo y cerrando "(" y ")", un ampersand "&", una barra vertical (pipe) "|", un caracter de menor que "<", o una tilde "∼".

Un dato de tipo *fact-address* es un identificador que utiliza CLIPS para referirse a un hecho particular. El identificador contiene la dirección en memoria del hecho la que se está refiriendo. En la sección de las reglas de CLIPS se explica cómo obtener este identificador de un hecho; se utiliza para realizar operaciones con hechos existentes, como retractarlos o modificarlos.

Dependiendo de la sintaxis que tengan los elementos incluidos en la definición de un hecho, CLIPS los interpretará como uno de estos tipos de datos, y se podrán utilizar funciones correspondientes con cada uno, por ejemplo, operaciones numéricas con los tipos *float* e *integer*.

A.2. Hechos

Los hechos son una forma de representar inforamción sobre el mundo. Cada hecho representa un pedazo de información que ha sido incluido en la lista de hechos del intérprete. La cantidad de hechos y la cantidad de información que pueden contener, está limitada únicamente por la cantidad de memoria en la computadora. Si se intenta crear un hecho que sea exactamente idéntico a otro, se ignorará la creación del "nuevo" hecho.

Algunos comandos, como **retract** o **modify** requieren que se les especifique un hecho como parámetro. Un hecho se puede especificar por un *fact-index* (índice de hecho) o un *fact-address* (dirección de hecho). Cuando un hecho es agregado a la lista de hechos (o modificado), se le asigna un número entero único, que representa el índice de ese hecho. Este número es simplemente un número que comienza en uno y se incrementa conforme se creen nuevos hechos (en realidad inicia en cero, pero ese índice es ocupado por un hecho inicial que crea el intérprete). Sin embargo, es difícil controlar en tiempo de ejecución cuál es el índice que corresponde a cada hecho, por eso se utilizan direcciones de hechos. Una fact-address puede ser obtenida capturando el valor de retorno de alguna función que regresa direcciones de hechos (como **assert** o **modify**), o asignando a una variable la dirección de un hecho que corresponde con un patrón en el LHS (*left hand side*, lado izquierdo) de una regla. (Vea el apartado de reglas de CLIPS en esta sección)

Hechos ordenados (Ordered facts). Consisten en un símbolo seguido de una secuencia de cero o más campos separados por espacios y delimitados por un paréntesis abriendo a la izquierda, y un paréntesis cerrando a la derecha. El primer campo de un hecho ordenado especifica una "relación" entre los demás elementos del hecho ordenado.

Algunos ejemplos de hechos ordenados son:

```
(objeto cubo1 cubo 0.0 0.0 0.0)
(objeto cubo2 cubo 0.0 1.0 0.0)
(objeto silla_favorita mueble 5.0 2.0 0.0)
(lista-del-super leche huevo pan cafe)
```

Hechos sin orden o estructurados (Non-ordered facts). Los hechos ordenados codifican información posicionalmente, es decir, cada posición corresponde con un atributo y el usuario debe saber además de qué campos están incluidos en un hecho, el orden en que están contenidos. Los hechos no ordenados (non-ordered facts), que también se pueden llamar hechos estructurados (*template facts*) le permiten al usuario abstraer la estructura de un hecho asignándole nombres a cada campo del hecho.

Algunos ejemplos de hechos estructurados son:

```
(objeto (nombre cubo1) (categoria cubo) (pos_X 0.0) (pos_Y 0.0) (pos_Z 0.0) )
(objeto (nombre cubo2) (categoria cubo) (pos_X 0.0) (pos_Y 1.0) (pos_Z 0.0) )
(objeto (nombre silla_favorita)
    (categoria mueble) (pos_X 5.0) (pos_Y 2.0) (pos_Z 0.0)
)
(lista-del-super (faltan leche huevo) (comprados pan) )
```

A.2.1. Hechos estructurados - `deftemplate`

Los hechos estructurados permiten utilizar campos nombrados en los patrones de precondiciones de reglas para facilitar su uso, ya que no es necesario recordar ningún orden en particular, ni especificar todos los campos de un mismo hecho. Los he traducido como hechos estructurados, porque son análogos a la definición de una estructura en un lenguaje de programación como C.

La gramática de una construcción `deftemplate` se muestra a continuación:

```
<deftemplate> ::= (deftemplate <nombre> [<comentario>]
                                <definicion_de_slot>*)
<definicion-de-slot> ::= <definicion-de-slot-sencillo> |
                                        <definicion-de-slot-múltiple>
<definicion-de-slot-sencillo> ::= (slot <nombre>
                                                <atributo-de-template>*)
<definicion-de-slot-múltiple> ::= (multislot <nombre>
                                                <atributo-de-template>*)
<atributo-de-template> ::= <atributo-de-default> |
                                        <atributo-de-restricciones>
<atributo-de-default> ::= (default ?DERIVE | ?NONE | <expresión>*) |
                                        (default-dynamic <expresión>*)
<nombre> ::= SYMBOL
```

Los atributos de default y de restruicciones permiten especificar un valor por default y asociar restricciones a los valores que se pueden asignar, ya sea de tipo o de valor, respectivamente. Para más información consulte el manual de CLIPS [26].

Un ejemplo de `deftemplate`:

```
(deftemplate objeto
    (slot nombre  (type SYMBOL)  )
    (slot categoria  (type SYMBOL)  )
    (slot pos_X  (type FLOAT) (default 0.0) )
    (slot pos_Y  (type FLOAT) (default 0.0) )
    (slot pos_Z  (type FLOAT) (default 0.0) )
)
```

A.3. Hechos iniciales - deffacts

La construcción `deffacts` sirve para crear los hechos iniciales de un programa. Cuando se ejecute la función **reset** se borrarán todos los hechos en la lista de hechos y se crearán los hechos contenidos en una construcción `deffacts`.

La sintaxis de esta construcción se muestra a continuación:

```
<deffacts> ::= (deffacts <nombre> [<comentario>]
                                <hecho>*)
```

Los hechos en una construcción `deffacts` pueden incluir llamadas a funciones para que se generen de manera dinámica al momento de realizar un **reset**. A continuación se muestra un ejemplo:

```
(deffacts inicio "Hechos que inicializan el estado de un cuarto"
    (numero_de_objetos 2)
    (objeto (nombre silla) (categoria mueble) (pos_X 5.0) )
    (refrigerador temp (obten-temp))
)
```

A.4. Funciones de usuario - deffunction

Las `Deffunction`s le permiten al usuario definir nuevas funciones directamente en CLIPS. El cuerpo de una `deffunction` es una serie de funciones que son ejecutadas en orden por CLIPS cuando la función es llamada, similar a las acciones de una regla. El valor de retorno de un `deffunction` es el valor de la última expresión evaluada dentro de la función. La llamada a un `deffunction` es idéntica a llamar cualquier otra función en CLIPS.

En el ejemplo presentado en la figura A-1, se muestra una función que llama a la función estándar de CLIPS **printout**, que con el parámetro t especifica que se debe imprimir a la salida estándar, y con el símbolo $crlf$ indica un salto de línea, de modo que la función *imprime* sirve para imprimir algo por la salida estándar y agregar un salto de línea al final.

La sintaxis de una `deffunction` es la siguiente:

```
(deffunction <nombre> [<comentario>]
    (<parametro-regular>* [<parámetro-comodín>])
    <acción>*
)
<parámetro-regular> ::= <variable-sencilla>
<parámetro-comodín> ::= <variable-múltiple>
```

En CLIPS, una variable sencilla se representa por un *símbolo* precedido por un signo de interrogación cerrando "?", y una variable múltiple es precedida además por un signo de pesos o dólares "$". Por ejemplo, $?x$ es una variable sencilla, y $?variables$ representa a una variable múltiple. Una variable múltiple puede unificar con un conjunto de cualquier longitud de valores separados por espacios. Es decir que una función definida por el usuario podrá reccibir cero o más argumentos posicionales y obligatorios, y opcionalmente cualquier cantidad de parámetros que serán asignados a la variable múltiple.

A.5. Reglas de CLIPS - `defrule`

Uno de los métodos principales para representar conocimiento en CLIPS es una regla. Las reglas de clips están formadas por dos partes: las precondiciones y las acciones. El desarrollador de un sistema experto define las reglas que describen cómo resolver un problema. Las reglas se ejecutan (o disparan) basándose en la existencia o no existencia de hechos o instancias de clases definidas por el usuario. CLIPS provee un mecanismo (máquina de inferencias) que intenta unificar las reglas con el estado actual del sistema, y aplicar las acciones correspondientes.

Las precondiciones de una regla, especificadas en el lado izquiedo (*left hand side*, LHS) de la regla, se forman con patrones que pueden contener variables y conectivos lógicos con estructuras complejas. Las acciones, el lado derecho (*rigt hand side*, RHS) de la regla, consisten en el llamado de funciones, generalmente se utiliza `assert`, para crear nuevos hechos; y `retract`, para retractar (eliminar) hechos que dejan de ser ciertos sobre el mundo; aunque en general en esta parte se puede llamar cualquier número de funciones que se ejecutan de manera secuencial. La creación y retracción de hechos habilita otras reglas (hace sus precondiciones verdaderas), de modo que se produce un desencadenamiento de reglas que va determinando la ejecución del programa.

Note que puede haber más de una regla habilitada, y la elección de cuál sería ejecutada podría ser arbitraria o no determinista. El diseñador de un sistema experto basado en reglas en general no debe presuponer nada sobre el orden de ejecución de las mismas cuando haya más de una regla habilitada a la vez (exceptuando el mecanismo de prioridades de reglas que se mencionará más adelante). De la misma manera, si existe más de un hecho que unifique con un patrón, el hecho que se use en la ejecución de la regla será arbitrario o no determinista. Si una regla sigue habilitada después de ser ejecutada (existen otros hechos que unifican con las precondiciones), es posible que se vuelva a ejecutar utilizando un hecho distinto, sin embargo, no se ejecutará nuevamente utilizando el mismo conjunto de hechos (Ver el ejemplo A-1b).

La sintaxis de una construcción `defrule` se muestra a continuación:

```
(defrule <nombre-de-regla> [<comentario>]
    [<declaración>]                    ; Propiedades de la regla
    <elemento-condicional>*            ; Lado izquierdo de la regla (LHS)
    =>
    <acción>*                          ; Lado derecho de la regla (RHS)
)
```

En CLIPS, el caracter de *punto y coma* ";" es utilizado para comentar el resto de la línea. Se puede incluir al inicio de una línea para comentar toda la línea, o al final de una línea en la declaración de alguna construcción, para añadir algún comentario relacionado con lo que se está declarando.

A las reglas se les puede incluir una declaración, que determina algunas propiedades de las reglas; la propiedad más frecuentemente utilizada es la de **salience**, un número entero que determina la prioridad de la regla; mientras mayor sea el *salience*, mayor prioridad tendrá, y por tanto, se ejecutará antes que otras reglas con un *salience* inferior.

El LHS y el RHS de una regla están separados por un símbolo de igual seguido de un símbolo de mayor que "=>". Las acciones de una regla corresponden a cero o más funciones que se ejecutan de manera secuencial, usualmente se utilizan **assert** y **retract** para crear y retractar hechos sobre el mundo respectivamente.

A.5.1. Elementos condicionales de una regla

El LHS de una regla se compone de una serie de elementos condicionales (ECs) que típicamente consisten en patrones de elementos condicionales (o simplemente patrones) que deben ser unificados con hechos (en realidad CLIPS también permite utilizar objetos en el LHS, pero en esta tesis me concentro únicamente en hechos).

Hay ocho tipos de ECs: **patterns**, **test**, **and**, **or**, **not**, **exists**, **forall**, y **logical**. Un EC **and** implícito siempre rodea a todos los patrones del LHS.

A.5.2. El EC *pattern*

Consiste en un patrón de un hecho, que puede contener constantes, comodines y variables sencillas y múltiples, y restricciones de tres tipos: restricciones conectivas, restricciones de predicado y restricciones de valor retornado. Los comodines pueden ser usados para indicar que pueden unificar con cualquier cosa (aunque no importe conocer el valor), mientras que las variables sirven para almacenar el valor con que unifica el patrón para poder ser usado después en otro EC, o en el RHS de la regla como argumento para alguna acción.

El primer campo de un patrón *debe* ser un símbolo (constante) y no puede ser usado con ningún tipo de restricción. Los nombres de lugares (*slots*) de un `deftemplate` también deben ser símbolos y no pueden contener ningún tipo de restricción. En el patrón más simple, que se le puede llamar restricción literal, incluye exactamente el valor con el que debe unificar el patrón.

Los patrones pueden incluir comodines sencillos y múltiples, denotados respectivamente por un signo de interrogación cerrando "?" y por un signo de pesos seguido de un signo de interrogación cerrando "$?". El comodín sencillo unifica con exactamente un campo, sin importar cuál sea su valor; mientras que un comodín múltiple unifica con cualquier cantidad de campos, también sin importar sus valores.

La máquina de inferencias se encargará de asignar al comodín los campos correspondientes. Cuando se utiliza más de un comodín múltiple la máquina de inferencias debe hacer la búsqueda de todos los posibles valores con que puede unificar y podría ejecutar la regla varias veces, una vez por cada posible asignación (en caso de que las demás restricciones se cumplan). Si es posible, se debe optar por reglas que utilicen el menor número de comodines múltiples posible.

El ejemplo A-3 muestra un ejemplo de reglas con restricciones literales, comodines y variables, así como sus activaciones, utilizando los hechos mostrados en la figura A-2.

```
1  (deffacts data-facts
2      (data 1.0 blue "red")
3      (data 1 blue)
4      (data 1 blue red)
5      (data 1 blue RED)
6      (data 1 blue red 6.9)
7  )
8
9  (deftemplate person
10     (slot name)
11     (slot age)
12     (multislot friends)
13 )
14
15 (deffacts people
16     (person (name Joe) (age 20))
17     (person (name Bob) (age 20))
18     (person (name Joe) (age 34))
19     (person (name Sue) (age 34))
20     (person (name Sue) (age 20))
21 )
22
```

(a) Programa con las construcciones.

```
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (data 1.0 blue "red")
f-2     (data 1 blue)
f-3     (data 1 blue red)
f-4     (data 1 blue RED)
f-5     (data 1 blue red 6.9)
f-6     (person (name Joe) (age 20) (friends))
f-7     (person (name Bob) (age 20) (friends))
f-8     (person (name Joe) (age 34) (friends))
f-9     (person (name Sue) (age 34) (friends))
f-10    (person (name Sue) (age 20) (friends))
For a total of 11 facts.
CLIPS>
```

(b) Índices de los hechos en una consola interactiva.

Figura A-2: Hechos de ejemplo.

Las reglas mostradas no contienen acciones, lo cual no es común en un programa real, pues no harían nada, pero en este caso sirven para mostrar la agenda, que es la lista de reglas activadas y que serían ejecutadas.

El primer tipo de restricción que puede tener el EC pattern (además de la restricción literal) es el de *restricciones conectivas*, que conectan entre sí a variables y restricciones individuales. Estas son & (and), | (or), y ∼ (not). La restricción & se satisface si las dos restricciones adjuntas se satisfacen, la restricción | se satisface si alguna de las dos restricciones adjuntas se satisface, y la restricción ∼ se satisface si la siguiente restricción no se satisface. Las restricciones conectivas se pueden combinar de cualquier manera. La restricción ∼ tiene mayor precedencia, seguida de la & y después la |; en cualquier otro caso, la evaluación se realiza de izquierda a derecha.

Sólo existe una excepción a las reglas de precedencia: si la primera restricción es una variable que no tiene un valor asignado y está seguida por un operador &, entonces la primera restricción se trata como una restricción separada que también debe ser satisfecha. Así, la restricción `?x&red|blue` es tratada como `?x&(red|blue)`, en vez de `(?x&red)|blue`, como indicarían normlamente las reglas de precedencia.

El segundo tipo de restricciones que puede tener el EC pattern es el de *restricciones de predicados*. Esta restricción permite restringir el valor de una variable basándose en la evaluación de alguna expresión booleana, expresada en una *función de predicado* que debe regresar el símbolo FALSE para que no sea satisfecha, y cualquier valor distinto de FALSE para que sea satisfecha.

```
3  (defrule match-all-persons
4      (person)
5      =>
6  )
7
8  (defrule find-data
9      (data ? blue red $?)
10     =>
11 )
12
13 (defrule match-1-blue
14     (data 1 blue)
15     =>
16 )
17
18 (defrule no-match
19     (data 1)
20     =>
21 )
22
23 (defrule find-data-1
24     (data ?x ?y ?z)
25     =>
26     (printout t ?x " : " ?y " : " ?z crlf)
27 )
```

(a) Reglas ejemplo 1

```
CLIPS> (agenda)
0      match-all-persons: f-10
0      match-all-persons: f-9
0      match-all-persons: f-8
0      match-all-persons: f-7
0      match-all-persons: f-6
0      find-data: f-5
0      find-data-1: f-4
0      find-data: f-3
0      find-data-1: f-3
0      match-1-blue: f-2
0      find-data-1: f-1
For a total of 11 activations.
CLIPS>
```

(b) Reglas activadas del ejemplo 1.

Figura A-3: Ejemplo 1. Restricciones literales, comodines y variables

Una restricción de predicado se invoca con dos puntos, seguidos del llamado de la función.

Típicamente, las restricciones de predicado se utilizan en conjunto con una restricción conectiva y una asignación de variable (i. e. se asigna un valor a una variable que será utilizada para realizar alguna evaluación en la función de predicado). A continuación se muestra el ejemplo de una regla que unifica si un campo es numérico, se muestran la regla se activará con los primeros dos hechos creados, y fallará con el tercero, pues no se trata de un dato numérico, sino un símbolo:

```
(assert (dato 1) (dato 2) (dato red) )


(defrule es_numerico
    (dato ?x&:(numberp ?x))
    =>
)
```

```
1  (deftemplate data-B (slot value))
2
3  (deffacts AB
4      (data-A green)
5      (data-A blue)
6      (data-B (value red))
7      (data-B (value blue))
8  )
9
10 (defrule rules_example2-1
11     (data-A ~blue)
12     =>
13 )
14
15 (defrule rules_example2-2
16     (data-B (value ~red&~green))
17     =>
18 )
19
20 (defrule rules_example2-3
21     (data-B (value green|red))
22     =>
23 )
```

(a) Programa ejemplo 2

```
CLIPS> (facts)
f-0     (initial-fact)
f-1     (data-A green)
f-2     (data-A blue)
f-3     (data-B (value red))
f-4     (data-B (value blue))
For a total of 5 facts.
CLIPS> (agenda)
0      rules_example2-2: f-4
0      rules_example2-3: f-3
0      rules_example2-1: f-1
For a total of 3 activations.
CLIPS>
```

(b) Hechos y reglas activadas del ejemplo 2.

Figura A-4: Ejemplo 2. Restricciones conectivas

Finalmente, la *restricción de valor retornado* valida que el valor de un campo sea igual al valor retornado por una función. El valor retornado debe ser de los tipos de datos primitivos. El valor retornado es incorporado directamente al patrón en la posición en que la función fue llamada como si se tratara de una restricción literal. Sin embargo, la función es evaluada cada vez que la restricción es evaluada (no sólo una vez).

Esta restricción se define utilizando un símbolo igual "=" seguido del llamado a la función, en el lugar donde se espera que aparezca el valor que coincide con el valor retornado. A continuación se incluye un ejemplo de un `deftemplate` que incluye dos valores y una regla que utiliza esta restricción para evaluar que el segundo sea el doble del primero:

```
(deftemplate dato (slot x) (slot y))


(assert (dato (x 2) (y 4) ) ; f-1
        (dato (x 3) (y 9) ) ; f-2
)


(defrule doble
    (dato (x ?x) (y =(* 2 ?x) )
    =>
)
```

Adicionalmente, un patrón puede incluir una variable que no corresponde con ningún campo, sino que representa la dirección (en memoria) del hecho al que se refiere y comúnmente es usada por funciones de CLIPS en el RHS, tales como **retract** o **modify**; aunque también es posible utilizarlas con **assert** como campo en un nuevo hecho y por ende, en el LHS de una regla para comparar con el campo de otro hecho. El motor de planeación descrito en este trabajo los utiliza de esta forma para crear hechos que identifican a otros hechos que representan planes como activos o habilitados.

Para obtener la dirección de un hecho se debe anteponer al patrón del hecho una variable sencilla y los símbolos "`<-`". A continuación se muestran un par de reglas que capturan la dirección de un hecho.

```
(defrule borra_hecho
    (dato 1)
    ?hecho <- (patron ejemplo)
    =>
    (retract ?hecho)
)

(defrule compara_hechos
    ?f1 <- (color ~rojo)
    ?f2 <- (color ~verde&:(neq ?f1 ?f2) )
    =>
    (printout t "La regla se dispara por hechos distintos" crlf)
)
```

A.5.3. El EC *test*

El CE *test* se satisface si una función que es llamada dentro de él regresa un valor distinto de FALSE. Funcionan como las restricciones de predicado y de la misma manera pueden utilizar de cualquier manera variables a las que ya haya sido asignado un valor. Las funciones que se pueden llamar pueden ser funciones estándar de CLIPS como funciones para realizar comparaciones entre números, para manipular cadenas de caracteres, e incluso las funciones *and* y *or* que funcionan para evaluar distintas expresiones compuestas por llamados a funciones; el único requisito es que la función de más alto nivel devuelva un valor.

Ejemplo:

```
(defrule evalua_numeros
    (dato ?x)
    (valor ?y)
    (test (>= (abs (- ?y ?x)) 3) )
    =>
)
```

A.5.4. El EC *or*

Si cualquier EC dentro de un EC *or* se satisface, entonces el EC *or* también se satisface. Permite a un EC habilitar una regla independientemente de que los demás ECs contenidos en él sean satisfechos o no. Note que una regla será activada por cada EC satisfecho dentro de un EC *or* (a no ser que las acciones deshabiliten la regla a través de la creación o retracción de otros hechos). Ejempo:

```
(defrule falla_del_sistema
    (estado_del_error desconocido)
    (or (temp alta)
        (valvula rota)
        (bomba (estado apagada))
    )
    =>
    (printout t "El sistema tiene una falla." crlf)
)
```

A.5.5. El EC *and*

Si todos los ECs dentro de un EC *and* se satisfacen, entonces el EC *and* también se satisface. CLIPS incluye un EC *and* implícito en todas las reglas, de modo que todas las precondiciones deben cumplirse para que se active una regla; sin embargo, existe un EC *and* explícito para hacer distintas combinaciones de ECs *and* y *or*. Ejemplo:

```
(defrule recetar_medicamento
    (or (and (paciente con_dolor)
            (paciente presenta_sintomas)
        )
        (paciente diagnosticado)
    )
    =>
)
```

A.5.6. El EC *not*

En algunos casos se quiere ejecutar una regla cuando *no* exista un hecho en particular, para esto sirce el EC *not*. El EC *not* se satisface solamente si el EC contenido en él **no** se satisface. Ejemplo:

```
(defrule agregar_elemento_a_lista
    ?lista <- (lista $?elementos)
    ?instruccion_agregar <- (agregar_elemento ?el)
    (not (test (member$ ?el $?elementos)))
    =>
    (retract ?lista ?instruccion_agregar)
    (assert (lista $?elementos elemento) )
)
```

A.5.7. El EC *exists*

El EC *exists* provee un mecanismo para determinar si un grupo de ECs especificados son satisfechos por al menos un conjunto de patrones. Debido a que CLIPS utiliza el principio del *closed world assumption*, se puede pensar que los patrones normales funcionan con un cuantificador existencial, ya que si no existieran los hechos, la regla no se activaría, sin embargo, es distinto en el sentido en que una regla puede ser activada varias veces por distintos hechos que unifiquen con sus precondiciones, mientras que utilizando el EC *exists*, se revisa solamente una vez si se cumple la condición, sin importar cuántos conjuntos distintos de hechos satisfacen las condiciones. La figura A-5 muestra dos reglas, una utilizando solamente un patrón, y otra utilizando el EC exist para evaluar si el dato existe.

```
1   (deffacts
2       (dato 1)
3       (dato 2)
4   )
5
6   (defrule existe_dato
7       (exists (dato ?))
8       =>
9       (printout t "Dato existe" crlf)
10  )
11
12  (defrule imprime_dato
13      (dato ?x)
14      =>
15      (printout t "Dato: " ?x crlf)
16  )
```

(a) Programa ejemplo con EC exists

```
CLIPS> (run)
Dato: 2
Dato existe
Dato: 1
CLIPS>
```

(b) Salida del ejemplo. Note que el número de activaciones de la regla es distinto.

Figura A-5: Ejemplo del EC exists.

A.5.8. El EC *forall*

El CE *forall* provee un mecanismo para determinar si un grupo de ECs especificados son satisfechos para cada ocurrencia de otro EC especificado. Ejemplo:

```
(defrule ejemplo_forall
    (forall (a ?x) (b ?x) (c ?x) )
    =>
    (printout t "Para cada hecho 'a' con valor 'x',
                existen los hechos 'b' y 'c' también con valores 'x'." crlf))
```

A.5.9. El EC *logical*

El EC *logical* provee una capacidad de *mantenimiento de la verdad*. Un hecho creado en el RHS puede ser lógicamente dependiente de hechos que unificaron con los patrones encerrados en un EC *logical* el el LHS de la regla. El soporte lógico quiere decir que cuando un hecho es retractado, los hechos que dependen lógicamente de él perderán el soporte lógico de ese hecho. Un hecho puede estar soportado lógicamente por más de un hecho, pero cuando pierde todos los soportes lógicos, será retractado también.

El soporte incondicional ocurre cuando el hecho no fue creado con soporte lógico (una regla como cualquiera de los ejemplos anteriores). Sólo los primeros n patrones de una regla pueden tener el EC *logical* aplicado; i. e. todos los ECs *logical* deben ir al principio de la regla. La figura A-6 muestra un ejemplo tomado del manual, donde un hecho pierde el soporte lógico y es retractado autmáticamente, mientras que otro que también tenía soporte lógico adquiere soporte incondicional al ser çreado"sin el soporte lógico.

```
1   (defrule rule1
2       (logical (a))
3       (logical (b))
4       (c)
5       =>
6       (assert (g) (h))
7   )
8
9   (defrule rule2
10      (logical (d))
11      (logical (e))
12      (f)
13      =>
14      (assert (g) (h))
15  )
```

(a) Programa ejemplo con EC logical

```
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (assert (a) (b) (c) (d) (e) (f) )
==> f-1     (a)
==> f-2     (b)
==> f-3     (c)
==> f-4     (d)
==> f-5     (e)
==> f-6     (f)
<Fact-6>
CLIPS> (run)
FIRE    1 rule2: f-4,f-5,f-6    ; la primera regla agrega soporte lógico
==> f-7     (g)
==> f-8     (h)
FIRE    2 rule1: f-1,f-2,f-3    ; la segunda regla agrega otro soporte lógico
CLIPS> (retract 1)              ; remueve el primer soporte lógico
<== f-1     (a)
CLIPS> (assert (h) )            ; soportada incondicionalmente
FALSE
CLIPS> (retract 4)
<== f-4     (d)                 ; remueve el segundo soporte lógico de g
<== f-7     (g)                 ; g ya no tiene soporte
```

(b) Salida del ejemplo. Note el comportamiento de los hechos con soporte lógico.

Figura A-6: Ejemplo del EC logical.

APÉNDICE

B

PYROBOTICS

B.1. Detalles del uso del BlackBoard

BlackBoard es una herramienta de paso de mensajes y respositorio de variables compartidas (más o menos equivalente a ROS: `http://ros.org`) desarrollada en el laboratorio de Bio-Robótica en una tesis de maestría [19]. Sirve para comunicar diferentes programas bajo los esquemas comando-respuesta y publicación-subscripción.

La manera en que está implementado es a través del esquema cliente-servidor, donde el BlackBoard toma el rol de cliente, y cada módulo que se quiere conectar, se comporta como servidor. El BlackBoard utiliza un archivo de configuración para conocer la ubicación de cada módulo (dirección IP y puerto) para establecer una conexión y poder comunicarse con ellos. Cada módulo puede entonces enviar mensajes que corresponden a comandos, que deberán ser ejecutados por otro módulo, o bien, por el mismo BlackBoard, en caso de ser comandos estándar que él entienda, por ejemplo el comando **write_var**, que sirve para escribir un valor a una variable compartida.

Los mensajes se envían en forma de texto y tienen el siguiente formato básico:

```
<nombre_comando> "<parametros>" @<id>
```

Donde el `<id>` es un número entero y sirve para identificar cada mensaje con su respuesta, `<nombre_comando>` puede ser cualquier cualquier cadena que pueda ser el nombre de una variable en el lenguaje C, y `<parametros>` es cualquier cadena de texto que el módulo destinatario tendría que interpretar.

Las respuestas incluyen adicionalmente, antes del id, un 1 en caso de que el comando haya sido exitoso, o un 0 en caso de que haya fallado, y pueden sustituir los parámetros por alguna respuesta que el comando deba generar, por ejemplo: una lista de objetos que el módulo de visión ha reconocido.

El esquema de comando-respuesta es adecuado para procesos que son ejecutados sobre demanda, es decir, en el momento en que se necesitan, se solicitan. Esto puede ser por varias razones, como que el robot necesite estar ubicado en alguna situación en particular, por ejemplo, un comando que sirva para abrir una puerta debería ser ejecutado solamente cuando el robot se encuentre frente a una puerta cerrada; o si el costo de procesamiento es muy alto para ejecutarlo continuamente, como los procesos de visión.

Sin embargo, muchas veces es conveniente reaccionar cuando algo en el ambiente sucede, independientemente del plan que se esté ejecutando, por ejemplo, cuando el robot escucha algún comando de voz. Para este tipo de problemas existe la comunicación de publicación-subscripción, en que cualquier módulo, que usualmente está sensando algo sobre el ambiente, puede escribir a una variable compartida, y todos los módulos que estén subscritos a ella recibirán una notificación de que el valor ha cambiado, para que puedan actuar en consecuencia si fuera necesario.

Para facilitar la implementación de nuevos módulos, se han desarrollado APIs de desarrollo que abstraen todos los detalles de conexión y manejo de mensajes de la aplicación para la que fue diseñada el módulo. Existen APIs para los lenguajes C#, C++ y python; en este apéndice se comenta un poco sobre el diseño de la API de python, llamada pyRobotics, y se muestran algunos ejemplos.

B.2. Documentación y ejemplos de pyRobotics

pyRobotics **no** está basada en las APIs de C# ni de C++, por lo tanto, su uso es diferente.

Un módulo desarrollado con la pyRobotics para conectarse con el BlackBoard necesita de dos partes básicas: [1]

1. Una rutina de inicialización donde:

   a) Proporciona la configuración necesaria para conectarse al BlackBoard y responder a los comandos que le corresponden.

   b) Deja todo listo para empezar a procesar comandos, dependiendo del módulo del que se trate.

   c) Inicia la comunicación con el BlackBoard.

   d) Opcionalmente crea y se subscribe a cualquier variable compartida que vaya a utilizar.

   e) Le avisa al BlackBoard que este módulo está listo para recibir comandos.

2. Un ciclo principal para hacer lo que sea que necesite hacer. Si el módulo sólo debería responder a comandos, se puede usar la función **BB.Wait()** para mantener un ciclo ocioso para prevenir que el programa termine (y por lo tanto, el ConnectionManager y el CommandManager, que se encargan de manejar la conexión y la recepción de comandos).

La rutina de inicialización debe incluir llamadas a las funciones **BB.Initialize**, **BB.Start** y **BB.SetReady**. *BB.Initialize* recibe básicamente un número de puerto y un mapa de funciones que define qué función ejecutará qué comando, *BB.Start* no recibe parámetros y sirve para inicializar la conexión con el BlackBoard; y *BB.SetReady* puede recibir un valor `booleano` (por default es `True`) indicando que el módulo está listo para recibir y procesar comandos[2].

[1] Realmente es el BB quien se conecta al módulo, pero como pyRobotics abstrae esto de la implementación, desde el punto de vista del desarrollador es más sencillo considerarlo como que los módulos se conectan a él.

[2] Esta función se puede llamar en cualquier momento para avisar al BlackBoard que su estado `ready` ha cambiado.

Los comandos más comunes, además de los arriba mencionados, que se pueden utilizar son los siguientes:

BB.Send Envía un comando SIN esperar por una respuesta. En general este comando es para uso interno, pero se provee la función en la API para casos en que realmente no importe la respuesta, o se trate de un módulo que contemple ejecución asíncrona de comandos y tenga otra forma de manejar las respuestas.

BB.SendAndWait Envía un comando y espera por la respuesta para continuar la ejecución.

BB.ReadSharedVar Es una función para consultar el valor de una variable compartida, sin estar subscrito ni tener que esperar por una notificación.

BB.CreateSharedVar Crea una variable compartida a la que se publicará información.

BB.WriteSharedVar Escribe (publica) un valor en una variable compartida.

BB.SubscribeToSharedVar Se suscribe a una variable compartida. Cuando la variable cambie su valor (alguien más escriba en ella), éste módulo recibirá una notificación.

La figura B-1 muestra un ejemplo de un módulo sencillo que responde al comando `tst_testfunction`, y envía dos comandos, uno con la función `BB.SendAndWait`, y otro con la clase `ParallelSender`, para después quedarse esperando en un ciclo ocioso para continuar recibiendo comandos.

```
import time
from pyrobotics import BB
from pyrobotics.parallel_senders import ParallelSender
from pyrobotics.messages import Command, Response

# These command handler functions could (and probably should) be defined in another module
# as to keep the main program file simple and clean.
def testFunction(c):
  print "testFunction called..."
  print 'Params: ' + c.params
  return Response.FromCommandObject(c, True, 'Command response')

fmap = {
     'tst_testfunction' : testFunction
}


def main():
  BB.Initialize(2000, fmap)
  BB.Start()
  BB.SetReady(True)

  print 'Sending command say...'
  print BB.SendAndWait(Command('spg_say', 'This is a test.'), 5000, 3)

  print 'Sending Async command...'
  ps = ParallelSender(Command('othertst_slowfunction', 'This is another test.'), 5000, 3)

  while ps.sending:
    print 'sending...'
    time.sleep(0.3)

  print 'Response received...'
  print ps.response

  BB.Wait()

if __name__ == "__main__":
  main()
```

Figura B-1: Módulo sencillo desarrollado en python con el uso de pyRobotics.

Este ejemplo fue tomado de la documentación y está incluido con el código fuente de la librería. Para consultar el código fuente y otros ejemplos puede acceder a la página de GitHub del proyecto: `http://www.github.com/BioRoboticsUNAM/pyRobotics/`.

La liga a una documentación un poco más extensa está incluida en el README del proyecto, pero se incluye también aquí por completez: `http://bioroboticsunam.github.io/pyRobotics/`.

APÉNDICE

C

BBCLIPS

Este software depende de la librería pyCLIPS (`http://pyclips.sourceforge.net/`), que sirve para embeber un intérprete de CLIPS en un programa de python; y de la librería pyRobotics, presentada en el apéndice B, que sirve para crear fácilmente módulos en python que se comuniquen con la herramienta *BlackBoard* desarrollada en el mismo laboratorio. Se creó con la intención de desarrollar módulos programados en CLIPS para nuestro robot de servicio en el Laboratorio de Bio-Robótica, abstrayendo los conceptos de comunicación implementados en el *BlackBoard*. Para entender mejor esta documentación se recomienda consultar la sección "Detalles del uso del BlackBoard" del apéndice B, así como la introducción a CLIPS presentada en la sección **??**.

BBCLIPS consiste básicamente en dos partes: la parte de CLIPS y la parte de python. La parte de python es básicamente un módulo que inicializa la conexión con BlackBoard y el intérprete de CLIPS. También define funciones que serían llamadas desde CLIPS, particularmente las funciones para enviar y recibir comandos, y para el manejo de las variables compartidas. La parte de CLIPS es un programa que se carga en el intérprete de CLIPS al iniciar BBCLIPS, para que al cargar otros programas hechos en clips se tenga acceso a las funciones que lo comunican con el BlackBoard.

El proyecto incluye una interfaz gráfica que permite controlar algunos parámetros del intérprete, como las cosas qué mostrar, así como el número de reglas a ejecutar cada vez que se presione el botón Run. También se puede correr en modo consola especificando parámetros en la invocación del programa. El modo consola es útil en plataformas como Raspberry Pi, donde los recursos son un tanto más limitados y donde adicionalmente es común que no se tenga un monitor conectado al sistema, sino que se realiza la comunicación a través de algún protocolo como secure shell (SSH). La figura C-1 muestra una captura de pantalla de la interfaz gráfica de BBCLIPS.

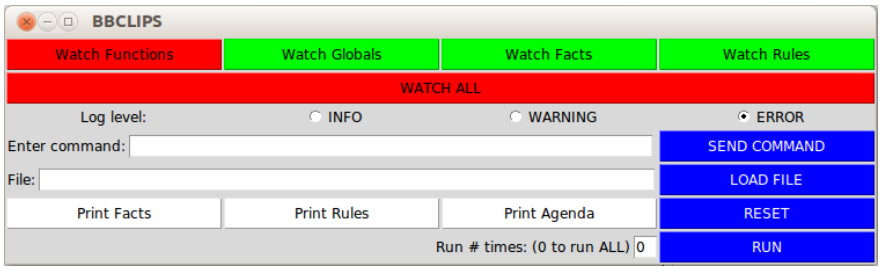


Figura C-1: GUI desarrollada para el programa de BBCLIPS.

Los botones `watch` determinan qué información interna del intérprete de CLIPS se mostrará en la consola, el control de `log level` está relacionado con una función utilitaria incluida en la parte de CLIPS para que los programas incluyan llamadas a la función `log`, y dependiendo del nivel actual se desplegará, o no, el mensaje en consola. El campo de `enter command` y el botón `SEND COMMAND` son para manualmente introducir un comando al intérprete de CLIPS. Los botones de `print` muestran los datos actuales del intérprete de CLIPS, y los botones `RESET` y `RUN` sirven para realizar las funciones correspondientes en el intérprete.

Los archivos que se pueden cargar en el campo `File` con el botón `LOAD FILE` son de extensión **.clp** para cargar programas de CLIPS, y **.lst** o **.dat** para cargar listas de archivos recursivamente. Un archivo con extensión *.lst* (o *.dat*) debe contener una lista de archivos con rutas relativas a su ubicación, que serían cargados recursivamente (i. e. los archivos *.lst* pueden incluir otros archivos *.lst*, además de archivos *.clp*).

C.1. Funciones y reglas de CLIPS

Los archivos de CLIPS que se cargan al incializar BBCLIPS incluyen funciones que se pueden utilizar en las reglas de otros programas, además de algunas reglas que manejan la llegada de mensajes o la limpieza de hechos para le manejo de los mensajes. Aquí se presentan las funciones que puede utilizar un desarrollador con otros programas de CLIPS, y se describen los hechos generados cuando se recibe un comando, una respuesta, o una notificación de variable compartida.

log-message Recibe un nivel y pedazos de mensajes que serán concatenados. El nivel es uno de los siguientes símbolos: `INFO` | `WARNING` | `ERROR` | `DEBUG` (siempre imprime, independientemente del nivel de log actual).

setTimer Inicializa un hilo con un timer. Recibe el tiempo en milisegundos y un símbolo para identificar el hecho que indica que el timer se consumió. Después de que el tiempo haya transcurrido, genera el hecho `(BB_timer <timer_symbol>)`, donde `<timer_symbol>` es el símbolo que se usó al llamar la función, y debería ser usado como precondición por otra regla.

sleep Recibe el tiempo en milisegundos. Esta función evita que python ejecuta al intérprete de CLIPS durante ese tiempo, incluso si recibe mensajes. En ese caso, hará el assert de los mismos, pero no se ejecutará ninguna regla hasta que el período de sleep haya terminado. No se recomienda el uso de esta función, a no ser que el usuario sepa lo que está haciendo.

send-command Recibe el nombre del comando, símbolo identificador, parámetros del comando y opcionalmente timeout y número de intentos en caso de que falle o transcurra el tiempo de timeout. El símbolo identificador (diferente al id de mensaje que utiliza el BlackBoard) sirve para rastrear respuestas a través de distintas reglas.

send-response Recibe el nombre del comando, id (de BlackBoard), resultado (1 ó 0) y parámetros de respuesta.

create-shared_var Recibe tipo y nombre. Tipo es una de: `byte[]` | `int` | `int[]` | `long` | `long[]` | `double` | `double[]` | `string` | `matrix` | `RecognizedSpeech` | `var`

write-shared_var Recibe tipo y datos. El tipo es alguno de la lista mencionada arriba en `create-shared_var`.

subscribe_to-shared_var Recibe nombre y opcionalmente tipo de subscripción y tipo de reporte. Tipo de subscripción es uno de: `creation` | `writemodule` | `writeothers` | `writeany`. Tipo de reporte es uno de: `content` | `notify`.

Adicionalmente, BBCLIPS genera los siguientes hechos, correspondientes a distintos eventos:

1. Cuando recibe una respuesta:

   ```
   (BB_answer ?cmd ?sym ?result ?params)
   ```

   Las primera variable y la última variable son cadenas de texto, la segunda variable corresponde con el símbolo que se usó al enviar el comando y result es un uno o un cero.

2. Cuando recibe un comando:

   ```
   (BB_cmd ?cmd ?id ?params)
   ```

   La variable `?id` es el identificador que las APIs de BlackBoard asignan a los mensajes, y debe incluirse en la respuesta con la función `sendresponse`.

3. Cuando se actualiza una variable compartida:

   ```
   (BB_sv_updated ?var $?values)
   ```

   La variable múltiple `$?values` dependerá del tipo de variable compartida de que se trate.

4. Cuando transcurre el plazo de un timer:

   ```
   (BB_timer ?sym)
   ```

   El símbolo corresponde con el símbolo utilizado en el llamado de la función `setTimer`.

Para más información, puede consultar la página del proyecto en GitHub: `http://www.github.com/BioRoboticsUNAM/BBCLIPS/`, donde se incluye el código fuente y otros datos.

BIBLIOGRAFÍA

[1] Bruno Siciliano and Oussama Khatib, editors. *Springer Handbook of Robotics*. Springer, 2008.

[2] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pages 608–620, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[3] Earl D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'75, pages 206–214, San Francisco, CA, USA, 1975. Morgan Kaufmann Publishers Inc.

[4] J. Scott Penberthy and Daniel S. Weld. Ucpop: A sound, complete, partial order planner for adl. pages 103–114. Morgan Kaufmann, 1992.

[5] Kutluhan Erol, James Handler, and Dana S. Nau. Semantics for Hierarchical Task-Network Planning. Technical Report CS-TR-3239, UMIACS-TR-94-31, ISR-TR-95-9, University Of Maryland, 1994.

[6] Dana S. Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. SHOP and M-SHOP: Planning with ordered task decomposition. Technical Report CS TR 4157, University of Maryland, jun 2000.

[7] J. Bohren and S. Cousins. The smach high-level executive [ros news]. *Robotics Automation Magazine, IEEE*, 17(4):18–20, Dec 2010.

[8] Luis A. Pineda, Lisset Salinas, Ivan V. Meza, Caleb Rascon, and Gibran Fuentes. Sitlog: A programming language for service robot tasks. *International Journal of Advanced Robotic Systems*, october 2013.

[9] Vittorio Amos Ziparo and Lucca Iocchi. Petri net plans. In *Proceedings of the Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA'06)*, 2006.

[10] Hugo Costelha and Pedro Lima. Robot task plan representation by petri nets: modelling, identification, analysis and execution. *Autonomous Robots*, 33(4):337–360, 2012.

[11] Bruno Lacerda and Pedro Lima. Designing petri net supervisors from ltl specifications. In *Proceedings of Robotics: Science and Systems*, Los Angeles, CA, USA, June 2011.

[12] J.L. Fernandez, R. Sanz, E. Paz, and C. Alonso. Using hierarchical binary petri nets to build robust mobile robot applications: Robograph. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 1372–1377, May 2008.

[13] W.J. Knottenbelt N.J. Dingle and T. Suto. Pipe2: A tool for the performance evaluation of generalised stochastic petri nets. *ACM SIGMETRICS Performance Evaluation Review (Special Issue on Tools for Computer Performance Modelling and Reliability Analysis)*, 36(4):34–39, March 2009.

[14] M. Baláž M. Riesz and G. Juhás. Petriflow: A petri net based framework for modelling and control of workflow processes. From the website: http://www.pneditor.org/.

[15] Jörg Desel, Gabriel Juhás, Robert Lorenz, and Christian Neumair. Modelling and validation with viptool. In WilM.P. van der Aalst and Mathias Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 380–389. Springer Berlin Heidelberg, 2003.

[16] Wil M.P. van der Aalst Christian W. Günther. Modeling the case handling principles with colored petri nets. In *Proceedings of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 211–230. Department of Computer Science, University of Aarhus, October 2005.

[17] Tapaal: Tool for verification of timed-arc petri nets. Website: http://www.tapaal.net/.

[18] Jesus Savage, Adalberto Llarena, Gerardo Carrera, Sergio Cuellar, David Esparza, Yukihiro Minami, and Ulises Peñuelas. Virbot: A system for the operation of mobile robots. In Ubbo Visser, Fernando Ribeiro, Takeshi Ohashi, and Frank Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI, July 9-10, 2007, Atlanta, GA, USA*, volume 5001 of *Lecture Notes in Computer Science*, pages 512–519. Springer, 2007.

[19] Mauricio Matamoros. Análisis de extensibilidad, reestructuración y desempeño de software para robots móviles. Master's thesis, UNAM, january 2013.

[20] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2 edition, 2003.

[21] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[22] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[23] H.J. Genrich and K. Lautenbach. System modelling with high-level petri nets. *Theoretical Computer Science*, 13(1):109 – 135, 1981. Special Issue Semantics of Concurrent Computation.

[24] H. J. Genrich. Predicate/transition nets. In *Advances in Petri Nets 1986, Part I on Petri Nets: Central Models and Their Properties*, pages 207–247, London, UK, UK, 1987. Springer-Verlag.

[25] W. A. Woods. Transition network grammars for natural language analysis. *Commun. ACM*, 13(10):591–606, oct 1970.

[26] *CLIPS Reference Manual. Volume I, Basic Programming Guide*, 2007.

ÍNDICE DE FIGURAS