Agentes Deliberativos Basados en Planificación Continua

Mario Moya and Claudio Vaucheret

Grupo de Investigación en Robótica Inteligente
Departamento de Ciencias de la Computación
Facultad de Economía y Administración
Universidad Nacional del Comahue
Buenos Aires 1400 - Neuquén - ARGENTINA
{moya.mario,vaucheret}@gmail.com

Resumen En este trabajo se presenta un framework para la implementación de agentes deliberativos cuyo razonamiento está basado en planificación continua. El planificador continuo es definido como una especialización del planificador de orden parcial con el agregado de características que lo hacen apto para ambientes dinámicos. La arquitectura del framework contempla también la ejecución concurrente de los procesos del planificador y el controlador del agente, así como la sincronización de los mismos. Asimismo, se muestra el uso del framework en el dominio del fútbol con robots, y se resaltan importantes aspectos de la implementación.

Key words: agents, planning, continuous planning, agent controller, concurrent prolog

1. Introducción

En Inteligencia Artificial (de ahora en más IA), la planificación[1] intenta definir de manera natural un conjunto de acciones que pueden ser aplicadas a un conjunto discreto de estados y construir la solución a un problema dando una secuencia apropiada de dichas acciones. Un agente orientado a una meta puede llevar a cabo esta tarea mediante algoritmos de búsqueda o con representación lógica del problema, pero una aproximación más eficiente es mediante planificadores [1].

Se denomina planificación clásica a aquellos algoritmos de planificación que asumen entornos de agentes que sean totalmente observables, estáticos, determinísticos y que las descripciones de las acciones sean correctas y completas. En estas circunstancias, un agente puede planificar primero y luego ejecutar el plan sin ninguna preocupación. Por otro lado, en la planificación denominada no clásica, se trabaja en un entorno incierto donde el agente debe usar sus percepciones para descubrir qué está pasando mientras el plan es ejecutado, y

[1] En inglés, *planning*.

posiblemente modificar o reemplazar el plan si pasa algo inesperado. Un agente tiene que tratar con información a veces incompleta, como es el caso de entornos parcialmente observables o no determinísticos, o incorrecta, en el caso de que el modelo que tiene el agente de su entorno no coincida con el real. En estos casos el agente puede encontrarse con un plan correcto al principio, pero que en el momento de actuar, éste ya no es aplicable. Para tratar con la planificación no clásica, algunos de los métodos son: *Planificación sin Sensores* [2], *Planificación Condicional* [3], *Ejecución, Monitoreo y Replanificación* [4] y *Planificación Continua*.

En este trabajo presentaremos una implementación de la última opción, es decir, un planificador diseñado para persistir en el tiempo sugerido en [5], que se ejecuta en un proceso concurrente con el proceso del controlador del agente como fue mostrado en [6]. El controlador implementa los ciclos de percepción y acción en forma independiente del planificador y ambos procesos interactúan actualizando las percepciones que pueden modificar un plan y las acciones que modifican el funcionamiento del agente.

El artículo se organiza de la siguiente manera: en la sección siguiente introducimos los aspectos claves de la planificación en ambientes del mundo real. Seguidamente, en la sección 3, presentamos la arquitectura de agente como un framework y la herramienta utilizada en su implementación. En la sección 4 profundizamos detalles del algoritmo de planificación continua. Posteriormente, en la sección 5, mostramos como utilizar el framework para el ambiente del fútbol de robots. Finalmente presentamos las conclusiones y los trabajos a futuro.

2. Planificación Continua en el Mundo Real

Muchos de los trabajos recientes en planificación dentro de la IA están relacionados con extender la planificación clásica para ambientes dinámicos, e investigar su aplicación en problemas de navegación y exploración, planificación con múltiples robots, o evasión de obstáculos. En problemas del mundo real pueden ocurrir eventos inesperados durante la realización del plan o su ejecución. Cualquier plan generado tiene la desventaja de que al momento de su ejecución puede estar obsoleto. Mientras se está armando el plan y hasta que se pueda ejecutar, probablemente el mundo ya no será el mismo que se percibió en principio, y muchas de las acciones ya no serán viables o llevarán a una solución equivocada y de esta manera el plan fallará. Puede ocurrir que cuando se ejecute una acción ésta no tenga el resultado esperado, o bien que no se pueda ejecutar debido a que las precondiciones que permitían que ésta acción se realice ya no sean ciertas para el estado actual.

En este trabajo proponemos una arquitectura para agentes basada en un planificador que toma principalmente las ideas del CONTINUOUS-POP-AGENT propuestas en [7] y extiende esos conceptos para llegar a una implementación real del agente.

Este agente se basa en uno de los planificadores más populares, el de orden parcial [8], en adelante POP, que saca ventaja de la descomposición de un pro-

blema en submetas. POP tiene un carácter incremental, donde el algoritmo va completando un plan parcial resolviendo sus deficiencias.

El algoritmo comienza con un plan vacío, conteniendo sólo las acciones *Start* y *Finish*. *Start* es una acción sin precondiciones para que pueda ser ejecutada inmediatamente, y tiene como efectos las condiciones que describen el estado inicial del problema. De manera análoga *Finish* no tiene efectos y tiene como precondiciones los literales que describen la meta a alcanzar. Además, cada plan tiene un conjunto de restricciones de ordenamiento que establecen qué acción debe ejecutarse antes que otra, sin necesidad de que esté inmediatamente antes.

Uno de los principales componentes del algoritmo es el tratamiento de los enlaces causales del tipo $A \xrightarrow{p} B$, que indica que p es un efecto de la acción A necesaria para la acción B. Las precondiciones abiertas son las que aún no son resueltas por ninguna acción. El algoritmo de planificación intentará resolverlas insertando nuevas acciones que las satisfagan sin introducir conflictos.

Así como el algoritmo POP resuelve precondiciones abiertas y conflictos causales, nuestro agente continuo debe además tener capacidades para poder enfrentar con eficiencia los cambios propios de los ambientes dinámicos y no determinísticos. El agente puede decidir agregar nuevas metas al estado *Finish*, puede agregar una nueva acción que resuelva las precondiciones abiertas o puede poner restricciones en el orden de ejecución de las acciones para evitar conflictos causales. Como una característica adicional nuestro agente puede prevenir la ejecución de acciones cuyas precondiciones ya no son verdaderas, o tomar ventaja de cambios inesperados pero beneficiosos a la hora de alcanzar una de sus metas, eliminando acciones que puedan resultar redundantes.

3. Un Framework Desarrollado en Prolog

El sistema está diseñado con el intento de facilitar el desarrollo de agentes basados en planificación continua y adaptarlos a distintos ambientes dinámicos. Para ello provee una estructura organizada en módulos y una metodología de trabajo que permite al programador del agente concentrarse en los requerimientos del dominio específico y su representación, así como en la representación de las acciones que ejecutará en el ambiente. Como no intenta dar soluciones genéricas, sino facilitar la construcción de una solución personalizada de acuerdo a las necesidades, lo podemos considerar como un framework.

La plataforma de desarrollo elegida para este framework es el entorno de programación "Ciao" [9] desarrollado por el grupo CLIP de la Universidad Politécnica de Madrid. La razón de esta elección es que es un entorno multi-paradigma que ofrece un conjunto de características únicas que lo hacen muy apropiado para el problema a tratar. Ciao ofrece un sistema Prolog completo, cuyas propiedades declarativas lo hacen ideal para el desarrollo de sistemas de agentes, con la particularidad de tener un diseño modular novedoso que permite tanto restringir como extender el lenguaje. Ciao también soporta, a través de estas extensiones, concurrencia, ejecución distribuida y ejecución paralela, lo que facilita la implementación de los algoritmos de IA para sistemas multi-agentes.

El framework está presentado en un estructura modular, siguiendo los principios del modelo *Belief-Desire-Intention* [10]. Nuestra arquitectura se compone de los módulos que a continuación detallamos y se pueden observar en la figura 1:

- continuouspop: este módulo actúa como planificador y obtiene las intenciones del agente, representando el estado deliberativo del mismo (lo que el agente ha escogido ejecutar).
- perceptions: este módulo es el encargado de traducir las percepciones obtenidas de los sensores y para ello implementa el predicado `get_perceptions`. El resultado de invocar este predicado es el conjunto de creencias que tendrá el agente sobre el mundo.
- actions: traduce las intenciones del agente a primitivas o comandos de bajo nivel que interactúan directamente con los efectores, como las velocidades en los motores de un robot.
- representation: contiene la representación de las acciones, sus precondiciones y sus efectos. Dicha representación también integra el conjunto de creencias del agente.
- goalgen: el módulo de generación de metas. Una meta es un deseo que se ha adoptado para que sea alcanzado activamente por el agente.

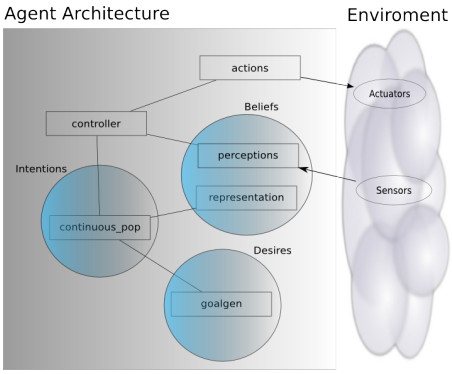

Figura 1. Arquitectura del Framework

Las implementaciones clásicas de Prolog contienen el concepto de predicados dinámicos, es decir, predicados que pueden ser inspeccionados o modificados en tiempo de ejecución, agregando o eliminando cláusulas. En Ciao el concepto de *data predicate* es una opción más eficiente que sirve a este propósito. Un *data predicate* es un predicado compuesto exclusivamente por hechos que pueden ser observados, agregados o eliminados dinámicamente. Usar estos predicados en lugar de los predicados dinámicos normales conlleva grandes beneficios en términos de velocidad y permite una mejor optimización. Un caso especial de

estos predicados permite implementar la sincronización y comunicación entre hilos[2] de ejecución diferentes. Predicados de este tipo se declaran con la directiva `:- concurrent pred/aridad.`

En el framework mantenemos dos hilos principales de ejecución, uno para el controlador o programa principal, encargado de tomar las nuevas percepciones del ambiente y actualizar el estado vigente del agente, y otro para el planificador que intentará encontrar una acción que satisfaga el conjunto de metas y percepciones actuales. Para la comunicación y sincronización entre estos dos hilos, utilizamos los predicados definidos como concurrentes `perception` y `action`. En el código que mostramos a continuación, podemos observar que el controlador cicla actualizando las nuevas percepciones con `assertz_fact(perception(Percepts))`, recuperandolas en la variable `Percepts`. Asimismo, recupera una acción, que el planificador deja a disposición del controlador con `retract_fact(action(Act))` para luego intentar ejecutarla.

```
:- concurrent perception/1.
:- concurrent action/1.

controller:-
  get_perceptions(Percepts),
  assertz_fact(perception(Percepts)),
  retract_fact(action(Act)),
  execute_action(Act),!,
  controller.
```

4. Implementación del Algoritmo de Planificación Continua

El planificador continuo es invocado desde el programa principal mediante la orden `eng_call` de Ciao Prolog generando un nuevo hilo de ejecución y recibe como parámetro un plan inicial vacío, conteniendo sólo las acciones *Start* y *Finish*.

El algoritmo es un bucle que obtienen las nuevas percepciones, actualiza el estado actual e intenta reparar el plan para que sea útil en el nuevo estado. Está implementando con el predicado `continuouspop` que cicla indefinidamente, actualizando los efectos del estado inicial mediante el llamado a `update_effects` con las nuevas percepciones que el controlador ha puesto a disposición. La reparación del plan está a cargo del predicado `remove_flaw`. Si `remove_flaw` encuentra alguna acción en condiciones de ser ejecutada, ésta se deja a disposición del controlador. Este último recibe como parámetros un plan para ser reparado y el límite de profundidad o DB, que simboliza el límite de acciones que existirán en el plan, con el objetivo de reducir el costo computacional en planes largos. Asimismo, devuelve en sus dos últimos parámetros el nuevo plan corregido y una acción para su posterior ejecución. En caso que `remove_flaw` no encuentre ninguna acción para ejecutar devuelve una acción especial denominada `noop` que significa

[2] threads, en inglés

no operación. El plan está representado como una estructura de datos que sigue las definiciones de [11]. Mantiene un agenda, o lista de precondiciones abiertas, enlaces causales, una lista de restricciones de orden en las acciones y una lista de restricciones demoradas, para ser evaluadas una vez que todo el plan esté completo.

El fragmento de código para `continuouspop` y `remove_flaw` es el siguiente:

```
continuouspop(Plan,DB):-
  retract_fact(perception(Percepts)),
  update_effects(Percepts),
  remove_flaw(Plan,DB,NewPlan,Act),
  asserta_fact(action(Act)),!,
  continuouspop(NewPlan,DB).

remove_flaw(plan(As,Os,Ls,Ag,DC),DB,NewPlan,Act) :-
  add_new_goals(Ag,Ag0),
  remove_unsupported_links(Ls,L1s,Ag0,Ag1),
  remove_redundant_actions(plan(As,Os,L1s,Ag1,DC),
                           plan(A1s,O1s,L2s,Ag2,DC)),
  solve_open_preconditions(plan(A1s,O1s,L2s,Ag2,DC),
                           plan(A2s,O2s,L3s,Ag3,DC),DB),
  unexecuted_action(plan(A2s,O2s,L3s,Ag3,DC),Plan3,Act),
  remove_historical_goals(Plan3,NewPlan).
```

En primer lugar agregamos nuevas metas a la agenda. El predicado `add_new_goals` recibe como parámetro la agenda actual y luego de procesar los deseos del agente, devuelve una nueva agenda.

Como segunda acción removemos del plan los enlaces que ya no son soportados en el estado actual. Esto se hace para evitar que `unexecuted_action`, invocado más tarde, intente devolver una acción para ejecutarla si sus precondiciones son falsas.

A continuación se buscan y eliminan las acciones redundantes que ya no son necesarias en el plan, quizá debido al éxito accidental de alguna de las metas. El predicado `remove_redundant_actions` examina los efectos de la acción *Start*, es decir, las nuevas percepciones, y explora dentro de la lista de enlaces causales si alguno de esos efectos está como precondición de alguna acción, es decir, busca enlaces causales de la forma $Act1 \xrightarrow{p} Act2$, donde *Act1* y *Act2* son acciones del plan y p es la precondición de la acción *Act2* que ahora es alcanzada directamente desde *Start*. En estas situaciones, la acción *Act1* ya no es necesaria, se remueve de la lista de acciones, y los enlaces causales se reemplazan por enlaces directos desde *Start*, $Start \xrightarrow{p} Act2$. Este reemplazo se denomina extender un enlace causal.

El siguiente paso en el planificador continuo, que hereda del planificador tradicional POP, es resolver las precondiciones abiertas. El algoritmo selecciona de la agenda una de las precondiciones e intenta resolverla, agregando una nueva acción al plan, o utiliza una de las acciones que ya existen. El predicado que resuelve estas precondiciones abiertas es `solve_open_preconditions`.

En este punto del algoritmo resta analizar si existe alguna acción que aún no ha sido ejecutada y que esté en condiciones de serlo. El agente continuo verifica si una acción A tiene sus precondiciones satisfechas en *Start* y ninguna acción además de *Start* está ordenada antes que A, entonces puede remover A y sus enlaces causales del plan, devolviendo ésta como la próxima acción a ejecutar. El predicado que verifica las acciones no ejecutadas es `unexecuted_action`.

Por último, con el predicado `remove_historical_goals` verificamos si se ha alcanzado el conjunto actual de metas. Esto se cumple cuando la agenda está vacía, es decir cuando no hay más precondiciones abiertas, y cuando no hay otras acciones en el plan más que *Start* y *Finish*, de manera que todos los enlaces causales vayan directamente de *Start* a *Finish*. En este momento el agente puede buscar un nuevo conjunto de metas para resolver en los siguientes ciclos.

5. Aplicación del Framework al Problema de Fútbol con Robots

Con el objeto de evaluar al framework del planificador continuo desarrollado, elegimos al fútbol de robots como caso de estudio. Esta elección está basada, principalmente, en dos características que presenta el ambiente en que se desenvuelve el juego: continuo y no determinístico.

En particular, utilizamos al software Rakiduam [12], que consiste en una arquitectura de agentes para equipos de fútbol de robots, que permite aplicar la misma estrategia del agente tanto a ambientes simulados como reales. Rakiduam fue implementado para actuar sobre el simulador de FIRA [13] y sobre robots físicos Lego [14]. En este último caso, se trabajó con el servidor de video Doraemon [15] como sistema de procesamiento de imágenes y un servidor de comandos que envía las órdenes a los robots que están en la cancha.

En primer lugar debemos identificar qué comportamiento pretendemos en nuestro jugador según la situación en la que se encuentre, con el fin de definir cuáles van a ser sus deseos. Seguidamente debemos analizar el dominio e identificar qué acciones serán necesarias para poder alcanzar cada una de la metas.

A través de un análisis restringido del dominio del fútbol, identificamos algunas estrategias que, de acuerdo a la posición de la pelota y el jugador en la cancha, le darán a este último una meta diferente. Queda fuera de esta discusión el rol del arquero por necesitar de un tratamiento diferente al resto de los jugadores.

De acuerdo a la arquitectura descripta en la sección anterior el módulo `goalgen` comprende los deseos del agente, y de acuerdo a las creencias actuales, generará un nuevo conjunto de metas consistentes entre sí. Por ejemplo, el agente no tendrá dos metas concurrentes para moverse hacia el arco propio y para moverse hacia la zona de ataque. En el caso del fútbol con robots, este módulo contendrá la estrategia del equipo y de cada jugador de acuerdo a su rol. Nosotros reutilizamos el trabajo hecho en Rakiduam, y creamos a partir de su estrategia una estrategia similar adaptada al planificador que revisa las creencias actuales y si las condiciones se cumplen devuelve alguna meta a cumplir por el agente.

En esta primera etapa de pruebas la generación de metas implementada consiste en adoptar un deseo, si y sólo si todas las condiciones que llevan a su generación se satisfacen. El hecho que sea un módulo aparte hace posible que siga creciendo con mejores herramientas de IA orientadas a la generación de metas.

Example 1 (Ejemplo de cómo el jugador obtiene metas que hacen su juego defensivo). En Rakiduam las acciones de cada robot se representan con el predicado acción(Rol,Robot,Izq,Der). En nuestra implementación utilizamos el predicado `get_desire_for_player(Name,Role,Goal)`, cuyos parámetros de entrada son el identificador del jugador y su rol y el parámetro de salida es la meta que desea lograr.

```
get_desire_for_player(Name,jugador,G):-
     holds(waiting_at(ball,Cell),init), cell_in_own_field(Cell),
     set_defence(Name,G).

set_defence(Name,carrying(Name,ball)):-
     behind_ball(Name).

set_defence(Name,waiting_at(Name,Cell)):-
     centro_arco_propio(X,Y),
     point_to_cell_position(X,Y,Cell).
```

En el ejemplo anterior, la primer cláusula del predicado `get_desire_for_player` analiza la posición de la pelota en el campo de juego para el rol `jugador`. Si resulta verdadero que la pelota se encuentra en el campo propio, el predicado `set_defence(Name,G)` retorna en el parámetro `G` alguna de las metas que hacen la estrategia defensiva. Si el jugador está detrás de la pelota, retorna `carrying(Name,ball)`, que simboliza el deseo del robot `Name` de adquirir la pelota identificada como `ball`. En caso contrario, la estrategia defensiva es que el robot se dirija hacia el centro del arco propio, representada por el parámetro `Cell` dentro de la meta `waiting_at(Name,Cell)`.

A partir del análisis hecho sobre el dominio, surgen las acciones que el agente debe realizar. Tales acciones las representamos en STRIPS [16] con predicados Prolog. Para las precondiciones de una acción utilizamos el predicado `preconditions(Action,PreconditionList)`. Para representar la lista de agregados luego de ejecutar una acción utilizamos el predicado `achieves(Action, Effect)`. Finalmente, la representación de la lista de borrados se realiza a través del predicado `deletes(Action,DeleteEffect)`.

Example 2. Consideremos un agente `Agent` que se mueve desde una posición `Pos1` a otra posición `Pos2`. El predicado `move(Agent,Pos1, Pos2)` representa esta situación. Las precondiciones y las listas de agregados y borrados son las que se detallan a continuación:

```
preconditions(move(Ag,Pos,Pos_1), [player(Ag),waiting_at(Ag,Pos),
                                   valid_move(Pos,Pos_1)]).
achieves(move(Ag,_Pos,Pos_1),waiting_at(Ag,Pos_1)).
deletes(move(Ag,Pos,_Pos_1),waiting_at(Ag,Pos)).
```

La semántica de estas acciones las da el módulo `actions`, que es totalmente dependiente de la representación en STRIPS. En este módulo implementamos cada una de la acciones y las traducimos, valiéndonos de las primitivas de navegación de Rakiduam, a las velocidades de los motores izquierdo y derecho de cada uno de los robots. En Rakiduam el agente obtiene el nuevo estado en cada ciclo, que básicamente son coordenadas de los jugadores y de la pelota dentro de la cancha. El módulo `perceptions` recibe entonces el nuevo estado y lo traduce a una representación que entiende el planificador.

Para describir el dominio utilizamos un módulo denominado `field_grid` que representa la cancha de fútbol en forma de grilla, y provee una serie de utilidades para facilitar la implementación de la estrategia que tendrá el agente para jugar, y para convertir las coordenadas que da el simulador o bien el servidor de vídeo, en coordenadas válidas para la representación y viceversa. Este módulo utiliza las herramientas para representar programación lógica con restricciones sobre dominios finitos que brinda Ciao en su paquete `fd` [17].

6. Conclusiones y Trabajos Futuros

En este trabajo se ha presentado una descripción del framework general desarrollado para la implementación de sistemas multiagentes basados en planificación continua. Se ha descripto la arquitectura y sus aspectos más significativos y se ha detallado la implementación del modulo de planificación continua, como una especialización del planificador de orden parcial, aprovechando las herramientas que brinda Ciao Prolog. Asimismo se ha aplicado con éxito el framework sobre el dominio de fútbol de robots tanto en ambientes simulados como reales.

Un controlador reactivo, como el utilizado en Rakiduam, resulta eficiente en ambientes reales por su rapidez para responder ante una determinada percepción, pero se muestra débil cuando tiene que resolver ciertas acciones complejas que requieren algún tipo de deliberación, sobre cuál es la mejor de las acciones disponibles. El controlador presentado en este framework es más robusto para interactuar en ambientes complejos y dinámicos, debido a la capacidad del planificador continuo de adaptarse a los constantes cambios que se producen en el ambiente, la posibilidad de crear nuevas metas según nuevas necesidades mientras continúa su ejecución, y la posibilidad de continuar con una estrategia reactiva en caso de que el planificador no tenga acciones disponibles.

Como trabajo futuro queremos aislar la representación en STRIPS de Prolog y utilizar archivos externos escritos en PDDL [18]. Para ello es necesario construir un intérprete PDDL o reutilizar alguno de los que ya existen en la comunidad. Asimismo, se planea mejorar el módulo de generación de metas, con el objeto de que el agente se comporte con mejor rendimiento que un agente reactivo tradicional ante una misma situación. Otra alternativa es incluir en el módulo capacidad de interpretar metas dadas por el programador en tiempo de ejecución.

Referencias

1. Fikes, R.E., Nilsson, N.J.: Strips, a retrospective. Artificial Intelligence **59**(1-2) (1993) 227–232
2. Goldman, R., Boddy, M.: Expressive planning and explicit knowledge. In: Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems(AIPS-96), Edinburgh, Scottland, AAAI Press (1996) 110–117
3. Olawsky, D., Gini, M.: Deferred planning and sensor use. In Sycara, K.P., ed.: DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control, San Diego, California, Defence Advanced Research Projects Agency (DARPA), Morgan Kaufmann (1990)
4. Wilkins, D.E.: Can ai planners solve practical problems? Computational Intelligence **6**(4) (1990) 232–246
5. Moya, M., Vaucheret, C.: Planificador continuo como controlador de agentes robots. In: X Workshop de Investigadores en Ciencias de la Computación, General Pico, La Pampa, Argentina, Universidad Nacional de la Pampa (2008)
6. Moya, M., Vaucheret, C.: Un planificador continuo concurrente para agentes robots. In: V Workshop de Inteligencia Artificial aplicada a la Robótica Móvil, Buenos Aires 1400, Neuquén, Argentina, Universidad Nacional del Comahue (2008)
7. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Second edn. Prentice Hall Series in Artificial Inteligence. Prentice Hall (2003)
8. McAllester, D.A., Rosenblitt, D.: Systematic nonlinear planning. In Press, A., ed.: Ninth National Conference on Artificial Intelligence (AAAI-91). Volume 2., Anaheim, California (1991) 634–639
9. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López, P., Puebla, G.: The Ciao Prolog System - A Next Generation Multi-Paradigm Programming Environment. Grupo Clip, `http://www.ciaohome.org/`. The ciao system documentation series edn. (August 2004)
10. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In Allen, J., Fikes, R., Sandewall, E., eds.: Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufmann publishers Inc.: San Mateo, CA, USA (1991) 473–484
11. Poole, D., Mackworth, A., Goebel, R.: Computational Intelligence: A Logical Approach. Oxford University Press (1998)
12. Moya, M., Torres, G., Kogan, P., Vaucheret, C., del Castillo, R., Cecchi, L., Parra, G.: Framework para el desarrollo de equipos de fútbol de robots. In: VI Campeonato Argentino de Fútbol con Robots, Buenos Aires 1400, Neuquén, Argentina, Universidad Nacional del Comahue (2008)
13. of International Robot-Soccer Association, F.: Fira official website. `http://www.fira.net/` (2008)
14. LEGO: Lego mindstorms. `http://mindstorms.lego.com/`
15. Baltes, J.: Yue-fei: Object orientation and id without additional markers. In: RoboCup 2001: Robot Soccer World Cup V. (2002)
16. Fikes, R.E., Nilsson, N.J.: Strips: A new approach to the application of theorem proving to problem solving. Artificial Intelligence **2**(3-4) (1971) 189–208
17. Gomez, J., Carro, M.: Constraint programming over finite domains. `http://clip.dia.fi.upm.es/Software/Ciao/ciao_html/ciao_189.html`
18. Ghallab, M., Howe, A., Knoblock, C.A., D., M.: PDDL-the planning domain definition language. Technical report, Yale Center for Computational Vision and Control, New Haven, Connecticut (1998)