Facsímil 02

Inteligencia clásica

Capítulo 09

Planificación automática: PDDL y modelado de dominios

686f6c61 · 2026

Capítulo 09: Planificación automática: PDDL y modelado de dominios

Entrando en el tema

Imagina una tarea aparentemente simple: “envía la factura al cliente”. Parece una instrucción de una sola línea. Pero si intentamos automatizarla de verdad, enseguida aparecen preguntas incómodas.

¿Existe el cliente? ¿La factura está completa? ¿El importe está calculado? ¿El correo está confirmado? ¿Hay permiso para enviarla? ¿Qué pasa si ya se envió ayer?

Un humano puede resolver esas preguntas con contexto y prudencia. Una automatización necesita otra cosa: un modelo explícito de estado, acciones, precondiciones y efectos. Eso es planificación automática.

Un plan no es una lista de tareas

Una lista dice qué pasos suenan razonables. Un plan dice qué pasos son ejecutables, en qué orden y bajo qué condiciones. La diferencia parece pequeña hasta que una acción tiene efectos reales.

En planificación clásica, el mundo se representa como un estado; las acciones solo se pueden aplicar si sus precondiciones se cumplen, y al aplicarlas cambian el estado. Ghallab, Nau y Traverso presentan esta idea como el modelo básico de la planificación automática: encontrar una forma de pasar de una situación inicial a una situación objetivo mediante acciones descritas formalmente.[1]

En agentes modernos pasa lo mismo, aunque el vocabulario sea distinto. El estado puede ser memoria, ficheros, base de datos, tickets, logs o respuestas de herramientas. Las acciones pueden ser tool calls. Las precondiciones son permisos, datos disponibles y reglas del negocio. Los efectos son cambios observables.

Russell y Norvig tratan la planificación como una extensión natural de la búsqueda: ya no buscamos solo un camino entre nodos, sino una secuencia de acciones que respete un modelo explícito del mundo.[2]

La planificación como sistema formal

Esta no es una notación que nos inventemos: es la formalización clásica de la planificación. Siguiendo a Ghallab, Nau y Traverso, una tarea se describe como un sistema de transición de estados (S, A, γ) más un estado de partida y una meta:[3]

$$\Pi = (S, A, \gamma, s_0, G)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
Π	Problema de planificación completo.	Enviar una factura validada.
S	Conjunto de estados posibles.	Todas las combinaciones de hechos sobre cliente, factura y envío.
A	Conjunto de acciones disponibles.	`validar_factura`, `enviar_email`, `registrar_log`.
γ	Función de transición: aplica una acción y devuelve el nuevo estado.	Si envío el email, aparece `email_enviado`.
s_0	Estado inicial.	Cliente identificado, factura preparada.
G	Objetivo: hechos que deben ser ciertos al final.	Factura enviada y log creado.

Una acción a tiene tres piezas:

$$a = (pre(a), add(a), del(a))$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$pre(a)$	Precondiciones: hechos requeridos antes de actuar.	`factura_validada` y `email_confirmado`.
$add(a)$	Hechos que pasan a ser verdaderos.	`email_enviado`.
$del(a)$	Hechos que dejan de ser verdaderos.	`factura_preparada`.

Esta forma de pensar viene de STRIPS, uno de los lenguajes clásicos para describir acciones mediante precondiciones y efectos de añadir o borrar hechos.[4]

Una acción es aplicable si todas sus precondiciones están en el estado actual:

$$\text{aplicable}(a, s) \Leftrightarrow pre(a) \subseteq s$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
a	Acción que queremos ejecutar.	`enviar_factura`.
s	Estado actual.	Conjunto de hechos verdaderos ahora.
$pre(a)$	Hechos requeridos por la acción.	`{factura_validada, email_confirmado}`.
$pre(a) \subseteq s$	Todas las precondiciones están presentes en el estado.	La factura está validada y el email confirmado.

Si la acción es aplicable, el nuevo estado se calcula así:

$$\gamma(s, a) = (s \setminus del(a)) \cup add(a)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\gamma(s, a)$	Estado resultante tras ejecutar la acción.	Estado después de enviar la factura.
$s \setminus del(a)$	Estado sin los hechos que la acción elimina.	Quitamos `factura_preparada`.
$add(a)$	Hechos nuevos que la acción añade.	Añadimos `email_enviado`.
$\cup$	Unión de hechos.	Juntamos lo que queda con lo nuevo.

Esta fórmula tan compacta resuelve, casi sin que se note, uno de los problemas más antiguos de la inteligencia artificial: el **frame problem**, el problema del marco.[5] Cuando ejecutas una acción, ¿qué cambia y, sobre todo, qué **no** cambia? Si envías un email, el correo pasa a estar enviado, pero el nombre del cliente, el importe y mil hechos más siguen igual. Enumerar en cada acción todo lo que no cambia sería interminable. STRIPS lo zanja con una regla simple, la **suposición STRIPS**: lo único que cambia es lo que aparece en `add(a)` y `del(a)`; todo lo demás se queda exactamente como estaba. Por eso la transición solo quita `del(a)` y añade `add(a)`, y deja intacto el resto del estado. Es una de esas ideas que parecen obvias hasta que intentas programar un agente sin ella y descubres que recuerda hechos que ya no son ciertos.

Por último, un plan es una secuencia de acciones:

$$\pi = \langle a_1, a_2, \ldots, a_k \rangle$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
π	Plan completo.	Validar, enviar, registrar.
a_i	Acción en la posición i.	$a_2 =$ `enviar_factura`.
k	Longitud del plan.	Tres acciones.

El plan es válido si, al aplicar sus acciones una a una desde s_0, llegamos a un estado final s_k donde el objetivo se cumple:

$$G \subseteq s_k$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
G	Objetivo que queremos lograr.	`{email_enviado, log_creado}`.
s_k	Estado después de ejecutar las k acciones.	Estado final tras validar, enviar y registrar.
$G \subseteq s_k$	Todos los hechos objetivo son verdaderos al final.	El email se envió y quedó registrado.

Un ejemplo pequeño: enviar una factura

Tomemos este estado inicial:

$$s_0 = \{cliente\_identificado,\ factura\_preparada,\ importe\_calculado,\ email\_confirmado\}$$

Y este objetivo:

$$G = \{factura\_validada,\ email\_enviado,\ log\_creado\}$$

ACCIÓN	PRECONDICIONES	AÑADE	ELIMINA
`validar_factura`	`cliente_identificado`, `importe_calculado`, `factura_preparada`	`factura_validada`	`factura_preparada`
`enviar_factura`	`factura_validada`, `email_confirmado`	`email_enviado`	Nada
`registrar_envio`	`email_enviado`	`log_creado`	Nada

El plan válido es:

$$\pi = \langle validar\_factura,\ enviar\_factura,\ registrar\_envio \rangle$$

No porque suene bien, sino porque cada paso se puede comprobar.

Si intentamos `enviar_factura` primero, falla: `factura_validada` todavía no pertenece a s_0. Si intentamos `registrar_envio` primero, falla: `email_enviado` todavía no es cierto.

Hacia delante y hacia atrás

¿Cómo encuentra el planificador ese plan? Hay dos direcciones, y conviene conocer las dos.

La **progresión** busca hacia delante: parte del estado inicial s_0, mira qué acciones son aplicables, genera los estados que producen y repite hasta llegar a un estado que contiene G. Es lo que hace el lab de este capítulo con una búsqueda en anchura. Es intuitiva, pero desde s_0 suele haber muchas acciones aplicables que no llevan a ninguna parte.

La **regresión** busca hacia atrás: parte del objetivo G y se pregunta «¿qué acción podría haber hecho cierto esto, y qué tendría que ser verdad antes de ejecutarla?». De `log_creado` retrocede a `registrar_envio`, cuya precondición es `email_enviado`; de ahí a `enviar_factura`, y así hasta encajar con s_0. La ventaja es que solo considera acciones **relevantes** para el objetivo e ignora todo lo que no acerca a G.

Ninguna gana siempre. La progresión va bien cuando hay pocas acciones aplicables en cada estado; la regresión, cuando el objetivo es específico y la mayoría de las acciones son irrelevantes. Los planificadores modernos eligen la dirección, o combinan ambas, según la forma del problema. Lo importante es ver que «buscar un plan» no es una receta única: es una búsqueda, con las mismas decisiones de dirección y orden que vimos en los primeros capítulos.

PDDL: separar dominio y problema

PDDL, *Planning Domain Definition Language,* se creó para estandarizar cómo describir problemas de planificación en competiciones y sistemas de planning.[6]

La idea más importante no es memorizar la sintaxis. La idea importante es separar dos cosas:

PIEZA	QUÉ CONTIENE	QUÉ NO DEBERÍA CONTENER
Dominio	Reglas reutilizables: tipos, predicados y acciones.	Datos concretos del caso de hoy.
Problema	Objetos, estado inicial y objetivo concreto.	Lógica general de las acciones.

El dominio describe cómo funciona el mundo:

```
(define (domain facturas)
  (:requirements :strips)
  (:predicates
    (cliente-identificado)
    (factura-preparada)
    (importe-calculado)
    (factura-validada)
    (email-confirmado)
    (email-enviado)
    (log-creado))

  (:action validar-factura
    :precondition (and
      (cliente-identificado)
      (importe-calculado)
      (factura-preparada))
    :effect (and
      (factura-validada)
      (not (factura-preparada))))

  (:action enviar-factura
    :precondition (and
      (factura-validada)
      (email-confirmado))
    :effect (email-enviado))

  (:action registrar-envio
    :precondition (email-enviado)
    :effect (log-creado)))
```

El problema describe el caso concreto:

```
(define (problem envio-factura-123)
  (:domain facturas)
  (:init
    (cliente-identificado)
    (factura-preparada)
    (importe-calculado)
    (email-confirmado))
  (:goal (and
    (factura-validada)
    (email-enviado)
    (log-creado))))
```

Separar dominio y problema es parecido a separar código y configuración. No reescribes la acción `enviar-factura` cada vez que cambia el número de factura. Cambias la instancia.

PDDL de verdad: acciones con parámetros

El ejemplo de arriba está simplificado a propósito: cada predicado es un hecho suelto, sin variables. El PDDL que se escribe en la práctica es **parametrizado**. Los predicados llevan argumentos, `(factura-validada ?f)`, y las acciones se escriben una sola vez con variables que luego se instancian sobre objetos concretos.

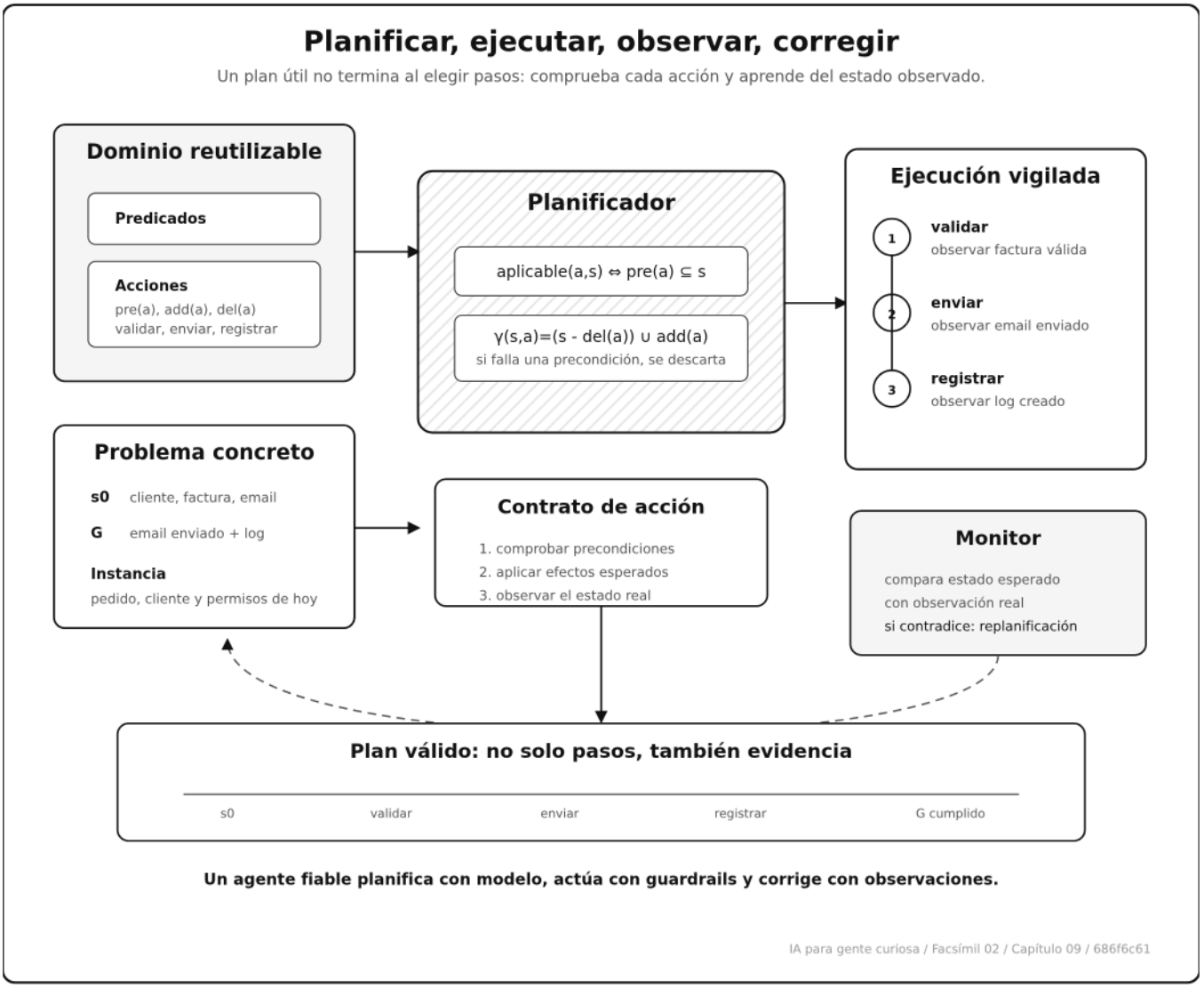
```
(:action enviar-factura
  :parameters (?f - factura ?c - cliente)
  :precondition (and (factura-validada ?f)
                     (email-confirmado ?c))
  :effect (email-enviado ?f ?c))
```

Esta única acción sirve para cualquier factura y cualquier cliente: el planificador la *instancia* sobre los objetos del problema (la factura 123, el cliente Ana) y obtiene tantas acciones concretas como combinaciones válidas haya. Es la diferencia entre escribir la regla una vez y copiarla a mano para cada caso. Esa capacidad de hablar de objetos y tipos (`?f - factura`) es lo que hace de PDDL un lenguaje práctico para dominios grandes, no solo para el ejemplo de juguete. La versión proposicional que hemos usado es ese mismo modelo «aplanado»: si tienes 3 facturas y 2 clientes, la acción parametrizada se expande a las combinaciones concretas, y volvemos a hechos sueltos como los del principio.

PDDL también nos enseña una disciplina útil aunque nunca lo usemos en producción: cada acción debe declarar qué espera del mundo y qué promete cambiar. Ese contrato permite detectar tres clases de errores que un texto libre disimula muy bien.

ERROR	CÓMO LO DETECTA EL MODELO DE PLANIFICACIÓN	EJEMPLO
Paso imposible	Falta una precondición.	Intentar enviar sin `factura_validada`.
Paso inútil	El efecto no acerca al objetivo.	Consultar tres veces el mismo pedido.
Paso peligroso	El efecto rompe una regla o requiere aprobación.	Enviar una factura de alto importe sin revisión.

Por eso PDDL encaja tan bien con tools y agentes: convierte una acción de “parece razonable” en una acción de “puedo comprobar si es legal”.



En el día a día

PDDL puede parecer antiguo, pero su disciplina es muy moderna. Cuando diseñas una tool para un agente, estás definiendo algo muy parecido a una acción de planificación.

PREGUNTA DE DISEÑO	EN PDDL	EN UNA TOOL MODERNA
¿Qué recibe?	Parámetros.	JSON schema o tipos.
¿Cuándo puede ejecutarse?	Precondiciones.	Permisos, estado y validadores.
¿Qué cambia?	Efectos.	Base de datos, fichero, ticket, email o log.
¿Cómo sé que funcionó?	Nuevo estado.	Observación verificable, test o evento registrado.

El capítulo anterior hablaba de guardrails. Este capítulo dice dónde viven muchos de esos guardrails: antes y después de cada acción. Antes, para comprobar precondiciones. Después, para comprobar efectos.

Cuando el mundo no coincide con el plan

La planificación clásica suele empezar con un modelo limpio: sabemos qué hechos son ciertos, qué acciones existen y qué efectos tienen. El mundo real rara vez es tan educado. Una API puede fallar, una credencial puede caducar, otra persona puede cambiar el ticket o una herramienta puede devolver un resultado parcial.

Por eso, en sistemas modernos, planificar no debería significar “generar diez pasos y ejecutarlos sin mirar”. El patrón más sano es planificar, ejecutar un paso, observar, actualizar estado y decidir otra vez. Si la observación coincide con el efecto esperado, seguimos. Si no coincide, replanificamos.

MOMENTO	PREGUNTA	EJEMPLO
Antes de actuar	¿Se cumplen las precondiciones?	¿La factura está validada y el email confirmado?
Al actuar	¿La tool devuelve un resultado estructurado?	`send_email` devuelve `message_id`.
Después de actuar	¿El efecto esperado aparece en el estado?	Existe `email_enviado`.
Si falla	¿Qué acción sigue siendo legal ahora?	Reintentar, pedir dato, escalar o parar.

En un agente con LLM, esta tabla vale oro. El modelo puede proponer el siguiente paso, pero el sistema debe mirar el estado real antes de aceptarlo. Esa es la diferencia entre un plan textual y una automatización operable.

Modo ingeniero: del formalismo al código

Las fórmulas de este capítulo no son adorno: se traducen casi línea a línea a código. Veámoslo en dos pasos, primero el planificador clásico y luego el agente moderno.

El núcleo del planificador

Un estado es un conjunto de hechos y una acción es una tripleta `(pre, add, del)`. Las dos operaciones centrales del capítulo, «¿es aplicable?» y «¿qué estado resulta?», son dos funciones de una línea, la traducción literal de $pre(a) \subseteq s$ y $\gamma(s, a) = (s \setminus del(a)) \cup add(a)$.

```
def aplicable(accion, estado):
    return accion["pre"] <= estado          # pre(a) ⊆ s  (subconjunto entre frozensets)

def aplicar(accion, estado):
    return (estado - accion["del"]) | accion["add"]   # γ(s, a) = (s \ del) ∪ add
```

Con eso, encontrar un plan es una búsqueda en anchura sobre estados, la progresión del apartado anterior: desde s_0, expande cada acción aplicable y para en cuanto un estado contiene el objetivo.

```
from collections import deque

def planificar(estado_inicial, objetivo, acciones):
    cola = deque([(estado_inicial, [])])
    vistos = {estado_inicial}
    while cola:
        estado, plan = cola.popleft()
        if objetivo <= estado:              # G ⊆ s
            return plan
        for nombre, accion in acciones.items():
            if aplicable(accion, estado):
                nuevo = aplicar(accion, estado)
                if nuevo not in vistos:
                    vistos.add(nuevo)
                    cola.append((nuevo, plan + [nombre]))
    return None                             # no hay plan
```

Esto es, en esencia, un planificador STRIPS mínimo. Fíjate en que `aplicable` y `aplicar` no «entienden» nada del dominio de facturas: solo manipulan conjuntos de hechos. El conocimiento del dominio vive en los datos (las acciones con su `pre`, `add` y `del`), no en el algoritmo. Es la misma separación dominio/problema de PDDL, ahora en código.

El ciclo del agente: planificar, ejecutar, observar, replanificar

El planificador clásico asume un mundo perfecto. Un agente real ejecuta en un mundo que puede contradecirle, así que el bucle no es «planifica y dispara los diez pasos», sino «planifica, ejecuta un paso, **observa** el estado real y vuelve a decidir». Aquí el SDK de Anthropic encaja de forma natural: el modelo propone el siguiente paso, pero cada acción sigue siendo una herramienta con precondiciones que el código verifica antes y efectos que comprueba después, exactamente como en el capítulo 8.

```
import anthropic

client = anthropic.Anthropic()

def precondiciones_ok(accion, estado):
    return ACCIONES[accion]["pre"] <= estado      # el mismo pre(a) ⊆ s

def ejecutar_paso(estado, objetivo):
    # 1. El modelo propone la siguiente accion como tool call.
    respuesta = client.messages.create(
        model="claude-opus-4-8",
        max_tokens=512,
        tools=TOOLS,
        messages=[{"role": "user", "content":
                   f"Estado: {sorted(estado)}. Objetivo: {sorted(objetivo)}. "
                   f"Propon la siguiente accion."}],
    )
    for bloque in respuesta.content:
        if bloque.type == "tool_use":
            accion = bloque.name
            # 2. El codigo verifica precondiciones ANTES de actuar (el guardrail).
            if not precondiciones_ok(accion, estado):
                return estado, "bloqueada: falta una precondicion"
            # 3. Se ejecuta y se OBSERVA el estado real, no el esperado.
            estado_real = observar(ejecutar(accion))
            # 4. Si el efecto esperado no aparece, toca replanificar.
            esperado = aplicar(ACCIONES[accion], estado)
            if not esperado <= estado_real:
                return estado_real, "replanificar: el mundo no coincide"
            return estado_real, f"ok: {accion}"
    return estado, "sin propuesta"
```

El modelo aporta flexibilidad, elegir el siguiente paso ante un estado que quizá no previste, pero no aporta la garantía. La garantía la dan las mismas comprobaciones de siempre: `pre(a) ⊆ s` antes, y comparar el efecto esperado con el estado **observado** después. La planificación clásica y el agente con LLM no compiten: el formalismo del capítulo es justo el armazón que vuelve fiable al agente.

Por qué debería importarte

Porque muchos agentes fallan no por falta de lenguaje, sino por falta de modelo de mundo. Redactan pasos razonables, pero no saben qué pasos son aplicables, qué dependencias faltan, qué efecto real produjo cada herramienta o cuándo deben parar.

La planificación automática te da una forma de depurar esas automatizaciones: mira el estado inicial, las acciones disponibles, las precondiciones, los efectos y el objetivo. Si alguna pieza no está escrita, el sistema puede estar improvisando.

Además, la planificación crece rápido. Si en cada estado hay b acciones aplicables y buscamos planes de longitud d, el árbol bruto puede crecer como:

$$O(b^d)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
b	Factor de ramificación: acciones candidatas por estado.	4 acciones disponibles.
d	Profundidad o longitud máxima del plan.	5 pasos.
$O(b^d)$	Crecimiento aproximado de combinaciones a explorar.	$4^5 = 1024$ secuencias posibles.

Bylander mostró que la planificación proposicional STRIPS tiene una complejidad computacional dura incluso con representaciones relativamente simples.[7] Por eso el siguiente capítulo hablará de heurísticas, planificación con SAT y técnicas para no explorar planes absurdos. Graphplan fue una de las grandes ideas para usar grafos de planificación y restricciones mutuas de forma eficiente.[8] FF, más tarde, hizo popular el uso de búsqueda hacia delante con heurísticas derivadas de planes relajados.[9]

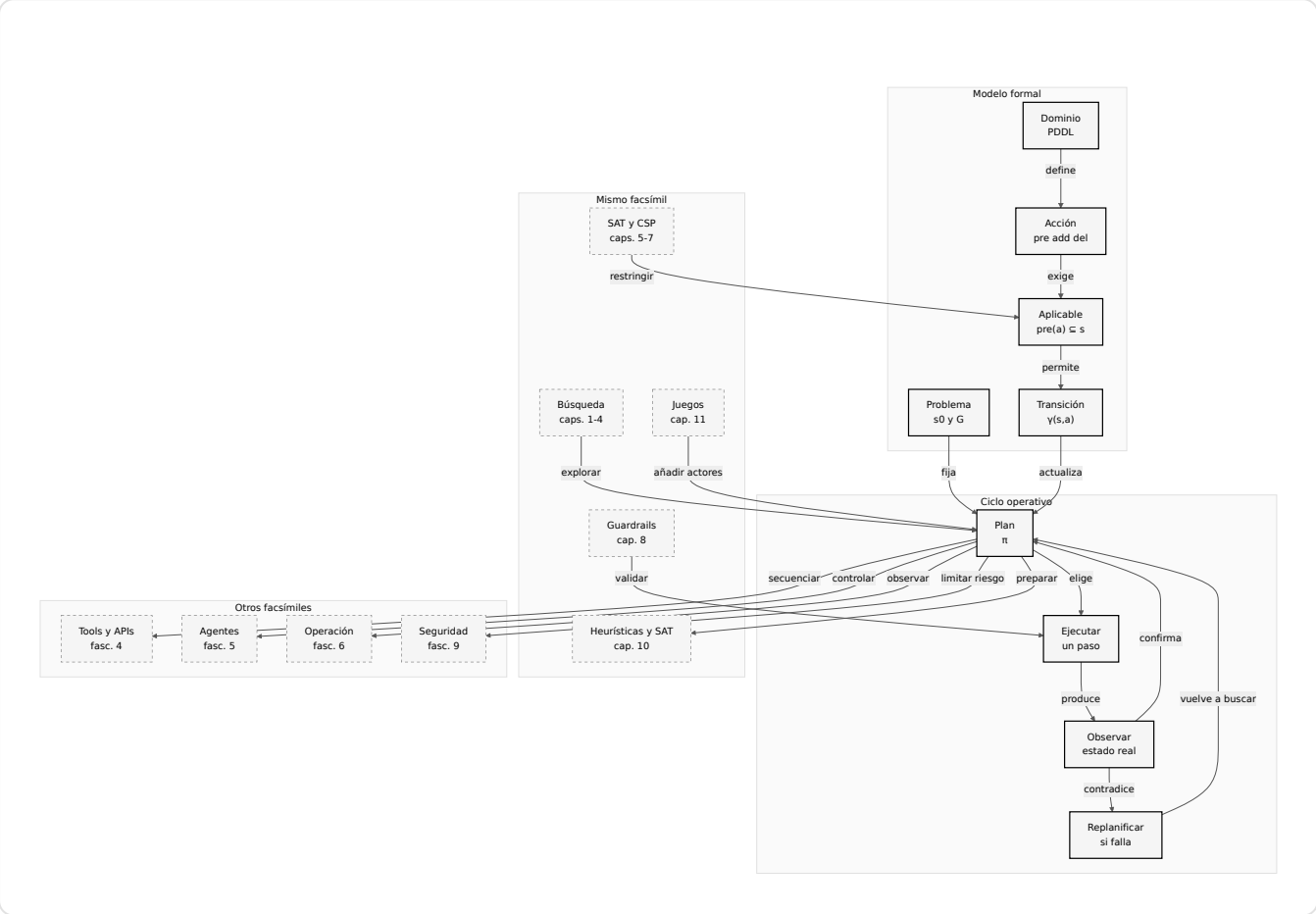
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Confundir plan con lista bonita	Una lista puede ignorar precondiciones y efectos.	Pregunta qué debe ser cierto antes y después de cada paso.
Meter el caso concreto dentro del dominio	Hace que cada instancia obligue a reescribir reglas.	Separa dominio reutilizable y problema de hoy.
No modelar efectos negativos	El sistema recuerda hechos que ya no son ciertos.	Escribe también qué se elimina: `del(a)` o `not (...)`.
Asumir que ejecutar es verificar	Una tool puede ejecutarse y aun así no lograr el objetivo.	Comprueba el estado resultante y registra evidencia.
Olvidar el coste de búsqueda	Los planes posibles crecen muy rápido.	Usa límites, heurísticas, SAT o descomposición.

Cómo encaja todo

Este mapa une búsqueda, restricciones y acción. Un dominio PDDL no es una lista bonita: define acciones con precondiciones y efectos. Un problema fija estado inicial y objetivo. El plan aparece cuando esas piezas encajan paso a paso.

La decisión nueva es tratar cada acción como contrato verificable. Esa idea prepara el capítulo 10, donde el plan se guía con heurísticas, SAT y observación real.



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Planificación automática	Búsqueda de una secuencia de acciones que transforma un estado inicial en un objetivo.
Predicado	Hecho verificable que puede ser verdadero o falso.
Precondición	Hecho que debe cumplirse antes de ejecutar una acción.
Efecto	Cambio que una acción produce en el estado.
Dominio PDDL	Descripción reutilizable de tipos, predicados y acciones.
Problema PDDL	Instancia concreta con objetos, estado inicial y objetivo.
Observación	Evidencia real que confirma o corrige el estado esperado tras actuar.
Replanificación	Construcción de un nuevo plan cuando la observación contradice el plan anterior.
Frame problem	Decidir qué hechos no cambian al actuar; STRIPS lo zanja suponiendo que solo cambia lo declarado en `add` y `del`.
Progresión	Buscar el plan hacia delante, de s_0 hacia G.
Regresión	Buscar el plan hacia atrás, de G hacia s_0.

Antes de pasar página

- [ ] ¿Puedo explicar por qué un plan no es solo una lista de pasos? (Si no, vuelve a «Un plan no es una lista de tareas».)
- [ ] ¿Sé distinguir estado inicial, acciones y objetivo? (Si no, vuelve a «La planificación como sistema formal».)
- [ ] ¿Entiendo cuándo una acción es aplicable: $pre(a) \subseteq s$? (Si no, vuelve a «La planificación como sistema formal».)
- [ ] ¿Sé calcular el nuevo estado con $(s \setminus del(a)) \cup add(a)$, y por qué eso resuelve el frame problem? (Si no, vuelve a «La planificación como sistema formal».)
- [ ] ¿Distingo la progresión (hacia delante) de la regresión (hacia atrás)? (Si no, vuelve a «Hacia delante y hacia atrás».)

- [ ] ¿Distingo dominio PDDL de problema PDDL y sé qué aporta una acción con parámetros? (Si no, vuelve a «PDDL: separar dominio y problema».)
- [ ] ¿Entiendo por qué ejecutar un paso debe producir una observación verificable? (Si no, vuelve a «Cuando el mundo no coincide con el plan».)
- [ ] ¿Sé traducir $pre(a) \subseteq s$ y $\gamma(s, a)$ a código y montar el ciclo planificar-observar-replanificar? (Si no, vuelve a «Modo ingeniero: del formalismo al código».)
- [ ] ¿Entiendo por qué el orden del plan importa? (Si no, vuelve a «Un plan no es una lista de tareas».)

En resumen

IDEA FUERZA	DETALLE
Planificar es transformar estado.	Partimos de s_0, aplicamos acciones legales y buscamos llegar a G.
Una acción tiene contrato.	Precondiciones antes; efectos después. Sin eso, una tool opera a ciegas.
PDDL separa reglas e instancia.	Dominio es la lógica reutilizable; problema es el caso concreto.
Los planes se verifican y se corrigen.	Ejecutar, observar y replanificar evita seguir una ruta que el mundo ya contradijo.

Para saber más

Blum, A. L. y Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2), 281-300. https://doi.org/10.1016/S0004-3702(96)00047-1

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2), 165-204. https://doi.org/10.1016/0004-3702(94)90081-7

Fikes, R. E. y Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4), 189-208. https://doi.org/10.1016/0004-3702(71)90010-5

Ghallab, M., Nau, D. y Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Hoffmann, J. y Nebel, B. (2001). The FF planning system: fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14, 253-302. https://doi.org/10.1613/jair.855

McCarthy, J. y Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. En B. Meltzer y D. Michie (Eds.), *Machine Intelligence 4* (pp. 463-502). Edinburgh University Press.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D. y Wilkins, D. (1998). *PDDL: The Planning Domain Definition Language, Version 1.2*. Yale Center for Computational Vision and Control. https://www.isi.edu/results/publications/19837/pddl-the-planning-domain-definition-language-version-1-2

Russell, S. y Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4.ª ed.). Pearson.

Notas

1. Ghallab, M., Nau, D. y Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann. ↩

2. Russell, S. y Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4.ª ed.). Pearson. ↩

3. Ghallab, M., Nau, D. y Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann. La planificación clásica se formaliza como un sistema de transición de estados (S, A, γ) con un estado inicial y un objetivo. ↩

4. Fikes, R. E. y Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4), 189-208. https://doi.org/10.1016/0004-3702(71)90010-5 ↩

5. McCarthy, J. y Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. En B. Meltzer y D. Michie (Eds.), *Machine Intelligence 4* (pp. 463-502). Edinburgh University Press. ↩

6. McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D. y Wilkins, D. (1998). *PDDL: The Planning Domain Definition Language, Version 1.2*. Yale Center for Computational Vision and Control. https://www.isi.edu/results/publications/19837/pddl-the-planning-domain-definition-language-version-1-2 ↩

7. Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2), 165-204. https://doi.org/10.1016/0004-3702(94)90081-7 ↩

8. Blum, A. L. y Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2), 281-300. https://doi.org/10.1016/S0004-3702(96)00047-1 ↩

9. Hoffmann, J. y Nebel, B. (2001). The FF planning system: fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14, 253-302. https://doi.org/10.1613/jair.855 ↩