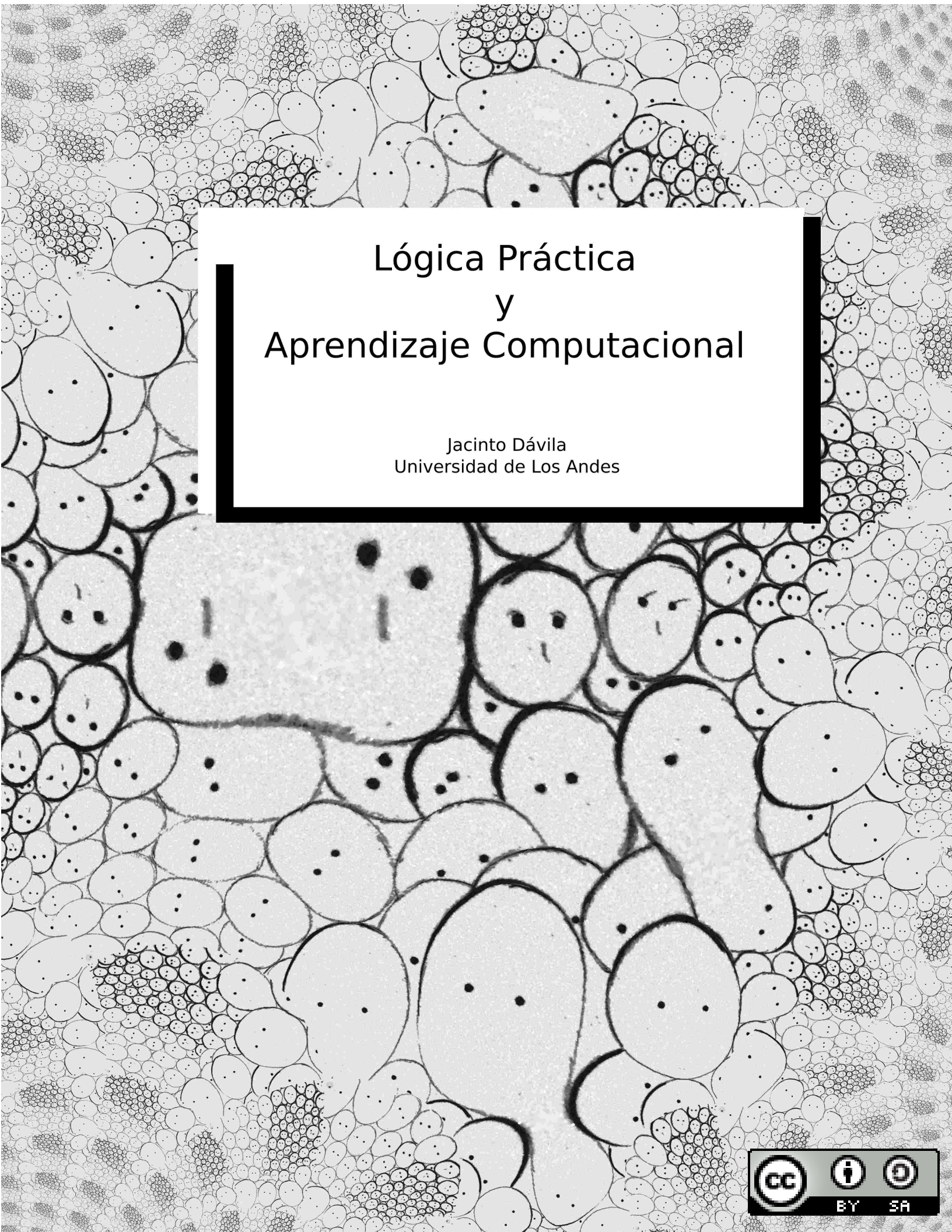


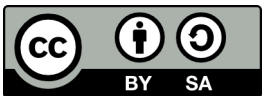
Lógica Práctica y Aprendizaje Computacional

Jacinto Dávila
Universidad de Los Andes

Lógica Práctica y Aprendizaje Computacional

Jacinto Dávila

8 de diciembre de 2009



Por las razones para explorar por nuevos caminos.

Índice general

Índice de figuras

Índice de cuadros

Introducción

¿Cómo evaluar los lenguajes que usamos para comunicarnos con el computador?. ¿Por qué tendríamos que hacer eso?. Los ejemplos de lógica que se presentan en este libro ilustran la sofisticación que ha alcanzado la disciplina y exponen una variedad de aplicaciones de la tecnología de la Inteligencia Artificial. Pero, ¿realmente estamos progresando? ¿Cómo sabemos que podemos comunicarnos cada vez más y mejor con las máquinas para lograr que hagan lo que queremos?. ¿Habrá algún paralelismo posible entre ese problema y el que ocurren entre humanos?.

Los lógicos computacionales parecemos decididos a enfrentar uno de los problemas más difíciles entre los legados por los padres fundadores de la disciplina. En el afán por construir un lenguaje perfecto[27], los fundadores diseñaron el lenguaje más simple que pudieron. El propósito principal parece haber sido "evitar la ambigüedad" o, cuando no se pueda, poder detectarla fácilmente. De allí viene la exigencia de "consistencia" que se le hace a cualquier código lógico y que, desde luego, obedece a la mejor de las razones: un código inconsistente permite derivar cualquier cosa, en particular una proposición y su opuesta. Es decir, es inútil. No tiene ningún valor práctico.

Ese esfuerzo para evitar la ambigüedad, sin embargo, parece haber ido muy lejos en muchos casos. Al despojar a los lenguajes lógicos de la riqueza estructural de los lenguajes naturales, pareciera que se los despoja también de su expresividad. Es decir, podría ocurrir que una verdad expresable en un lenguaje, puedo no serlo en otro. La falta de expresividad no es, técnicamente, fácil de establecer. Es un juicio que involucra a dos lenguajes (el "juzgado" y el de referencia) o, quizás más propiamente, a dos juegos lingüísticos [90].

Pareciera entonces que estamos condenados a un juicio relativo. Por ejemplo, uno puede reexpresar lo que se supone no es expresable en un lenguaje y "hacerlo expresable". Un ejemplo clásico de esos desafíos es la disputa entre lógica clásica y lógica modal. En esta última, los símbolos $\Diamond p$ para indicar lo que en Español diríamos "es posible p" o "posiblemente p es el caso". Según muchos lógicos (modales), eso no se puede expresar en lógica clásica. Pero en esta se puede decir $\exists U(cierto(p, U))$, queriendo decir que existe un universo en el que p es cierto. ¿No es esa una forma equivalente? [38].

Como ilustra brevemente esa caso, la lógica modal abrevia la sintaxis (usando símbolos más expresivos, porque dicen más con menos símbolos) en detrimento de la semántica que se torna más compleja (esto es, comparando sintaxis y

semánticas entre la lógica modal y la clásica). Pero la expresividad de ambos lenguajes, considerados como un todo (es decir, lenguaje = sintaxis + semántica) es la misma.

¿Será posible identificar una auténtica limitación en el lenguaje? (es decir, las condiciones de posibilidad para que sintaxis + semántica "no alcancen").

Los lingüísticas modernos cuentan con una tercera categoría para evaluar la expresividad de un lenguaje (natural): la pragmática. La pragmática se refiere al irreductible contexto en el que se dice algo. Es decir, las oraciones (o cualquier otra estructura lingüística) adquieren uno u otro significado de acuerdo al contexto en el que se insertan. Este es el elemento más básico o crudo de lo que Wittgenstein llama juego lingüístico[90] y es lo mejor, según el filósofo, que podemos acercarnos a una caracterización del "lenguaje siendo usado" (o dinámica lingüística).

La lógica computacional, tal como se le muestra en este texto, está conectada a esa posibilidad de "lenguaje siendo usado". El humano y la máquina se enfrentan en una forma degenerada de juego lingüístico[1]. Un juego degenerado por la declarada falta de consciencia en la máquina, pero un juego en el que los contextos también determinan significados.

De esta manera, para evaluar un lenguaje computacional uno puede recurrir a un modelo de lenguaje (él mismo escrito en algún lenguaje) que dé cuenta del cómo sus formas (las del lenguaje) significan (en su semántica) respecto a cada contexto de uso posible. El usuario del lenguaje, lector comprometido, juzgará, tan sistemáticamente como quiera, si el lenguaje "alcanza" para tal o cual uso.

[1] El verdadero juego ocurre, quizás, entre quienes diseñaron la máquina y quienes la usan

Capítulo 1

Lógica Práctica

1.1. Introducción

Para decidir si la lógica es útil o no uno tendría que producir un **argumento** (en favor o en contra). Entre lógicos se dice, a modo de broma, que esta es la mejor **prueba** de la utilidad de la lógica. Muchas veces, sin embargo, no ocurre que se dude de la utilidad de la lógica en sí misma, sino que las formas de lógica disponibles tienen tal nivel de complejidad que su utilización **práctica** (distinta de su utilización en la especulación teórica) es muy limitada.

La lógica que se propone en este texto tiene 2 características que nos permitimos ofrecer para justificarle el atributo de práctica o útil:

1. Una **semántica** relativamente simple.

2. Una realización computacional.

1.2. La Semántica de la Lógica Práctica

En la **lógica matemática** clásica, el mundo (el universo entero y toda otra posibilidad) se ve reducido a un conjunto de objetos y a un conjunto de relaciones (entre los mismos objetos o entre las mismas relaciones). Eso es todo lo que existe (o puede existir): objetos y relaciones.

Esta **ontología** degenerada (respecto a la nuestra) de la lógica clásica es frecuentemente criticada por **reduccionista**. Los **argumentos** para esas críticas, muchos ampliamente aceptados como válidos, son tolerados (o mejor, ignorados) por los lógicos en virtud de una gran ventaja que ofrece esa ontología: Es menos compleja y, por tanto, es más fácil pensar con ella (o con respecto a ella).

En definitiva, para darle significados a los símbolos en lógica contamos con:

OBJETOS designados por sus nombres, llamados también **términos** del lenguaje y

RELACIONES cuyos símbolos correspondientes son los nombres de predicados o, simplemente, predicados.

Por ejemplo, el símbolo `ama(romeo, julieta)` es un literal positivo o átomo constituido sintácticamente por el predicado `ama` y sus dos argumentos, los términos `romeo` y `julieta`.

Para saber qué significa esos términos y ese predicado, uno tiene que proveerse de una definición de la **semántica** de ese lenguaje. Una forma de hacerlo es usar (referirse a) una historia como la que sugieren esos nombres o, al estilo matemático, definirlos:

semántica

`romeo` se refiere al jovencito hijo de los Montesco, en la Novela de William Shakespeare.

`julieta` se refiere a la jovencita, hija de los Capuletos, en la misma Novela.

Hay muchas formas de hacer esa definición[1]. En cada caso, quien la hace tiene en mente su propia **interpretación** . Uno podría, por ejemplo, decir:

interpretación

`romeo` es el perrito de mi vecina.

y no habría razón para decir que esa interpretación es incorrecta.

Esas definiciones dan cuenta del nombre, pero no del predicado. Este se define también, pero de una forma diferente. Al predicado le corresponde una **relación**. Si hacemos la intepretación que pretende Shakespeare, por ejemplo, la relación correspondiente es aquella en la que el objeto en el primer argumento ama al objeto en el segundo. Es decir, los matemáticos definen la relación *amar* como la colección de todas las duplas (X,Y) en las que X ama a Y. Los computistas, siguiéndoles la corriente, dicen que eso es muy parecido a tener una **tabla** en una **base de datos** que describe todo lo que es **verdad**.

Para saber si Romeo (el de la historia) realmente ama a Julieta (la de la historia) uno tendría que revisar la base de datos (correspondiente a la historia) y verificar que contiene la tupla correspondiente. Noten, por favor, una sutileza. Esa no es una base de datos de palabras o términos electrónicos. Es, de hecho (dicen los lógicos clásicos), una reunión (¿imaginaria?) de **objetos reales**.

Por simplicidad (y sin olvidar lo que acabo de decir), uno puede representarla así:

	amante	*amado*
AMA	romeo	julieta
	julieta	romeo
	bolívar	colombia
	manuela	bolívar

y, colapsando todo en palabras (en la sintaxis), también se puede representar (y se suele hacer) como se muestra en el código 1 (noten que uno puede anotar esta información de muchas maneras diferentes, e.g. el primer hecho se puede escribir también así: *romeo ama a julieta*).

[1] vean `http://es.wikipedia.org/wiki/Romeo_y_Julieta`

Algorithm 1 Codificando una interpretación

```
ama(romeo, julieta).
ama(julieta, romeo).
ama(bolívar, colombia).
ama(manuela, bolívar).
```

p	q	p **y** q
v	v	v
f	v	f
v	f	f
f	f	f

Cuadro 1.1: La semántica de **y**

Esta explicación dispensa con algunos elementos básicos de la semántica de la lógica. Pero no con todos. Faltan los más útiles. Las llamadas **palabras lógicas**. En la lógica, uno puede crear **oraciones compuestas** a partir de oraciones simples (como `ama(romeo, julieta)`). Digamos que usamos los símbolos **y**, **o**, **no**, **si** (siempre acompañado de **entonces**) y **si y solo si**. (los símbolos empleados son también una decisión del diseñador del lenguaje. Algunas veces se usan $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$ y muchos otros).

Sabremos que `ama(romeo, julieta)` es verdadera al revisar la *base de datos de la interpretacion* que se haya seleccionado. Pero, ¿cómo sabemos que `ama(romeo, julieta)` y `ama(julieta, romeo)` es verdadera?

La respuesta está codificada en la **tabla de verdad** de la palabra **y** en tabla 1.1: **tabla de verdad**

Si podemos decir que `p = ama(romeo, julieta) = verdad` y `q = ama(julieta, romeo) = verdad`, entonces la **conjunción** es también verdad **en esa interpretación**.

Esas tablas de verdad 1.2, 1.3 y 1.4 son útiles porque nos permiten decidir si un discurso, visto como una (gran) fórmula lógica, es cierto o falso. Esto es importante para verificar si tal discurso es coherente o, dicho de otra manera, está razonablemente estructurado. En lógica, cuando el discurso es verdadero en alguna intepretación decimos que es un **discurso consistente** . Un discurso o teoría es **inconsistente** si permite derivar una proposición y su opuesta: e.g. **consistencia lógica** **inconsistente**

p	q	p **o** q
v	v	v
f	v	v
v	f	v
f	f	f

Cuadro 1.2: La semántica de **o**

q	no q
v	f
f	v

Cuadro 1.3: La semántica del **no**

p	q	**si** p **entonces** q
v	v	v
f	v	v
v	f	f
f	f	v

Cuadro 1.4: La semántica de **si**

te amo y no te amo.

1.2.1. Sobre la validez de un argumento

Esos ejercicios de análisis de significados con tablas de verdad tienen otro sentido práctico. Se les puede usar para evaluar la validez de un argumento.

validez

> Un argumento es válido si siempre que sus condiciones son ciertas, sus conclusiones también son ciertas.

argumento

Argumento es uno de los conceptos trascendentales en lógica. Se refiere a toda secuencia de fórmulas lógicas, conectadas entre sí (por relaciones lógicas), que establece (**prueba**) una conclusión. Un ejemplo muy conocido de argumento es este:

> Todo humano es mortal. Sócrates es humano. Por lo tanto, Sócrates es mortal.

que puede ser codificado y usado por un computador de esta forma:

```
mortal(X) :- humano(X).

humano(socrates).

?mortal(X)

X = socrates

true
```

¿Ve lo que ocurre en el ejemplo? [2]

[2] Hemos convertido en el argumento en un código Prolog que le permite a una máquina simular el razonamiento humano, en este caso con absoluta fidelidad.

Para verificar que un argumento es válido basta verificar que **CUANDOQUIERA** que sus condiciones son todas ciertas, sus conclusiones también (son todas ciertas). Vale la pena tomar un tiempo para aclarar el concepto. Algunas personas tienen dificultad para aceptar que un argumento puede ser válido (aún) si:

> sus condiciones son falsas y sus conclusiones son ciertas o
> sus condiciones son falsas y sus conclusiones son falsas tambien.

De hecho, lo único que **NO** puede ocurrir es que

> sus condiciones sean ciertas y su conclusiones falsas.

Recapitulando, un argumento es válido si cuandoquiera que sus premisas (condiciones) son conjuntamente ciertas, su conclusión (o conclusiones) también son ciertas. Esta definición, como suele pasar en matemática, sirve para describir, pero no para guiarnos al producir un argumento válido. El que podamos construir un argumento válido va a depender, siempre, de que podamos establecer la conexión adecuada entre sus premisas y sus conclusiones.

En particular, la validez de un argumento **NO DEPENDE** de los valores de verdad de sus componentes. Eso es lo que verificamos con los siguientes ejemplos (basados en los ejemplos del libro de Irving Copi y Carl Cohen[7]).

Todas las ballenas son mamíferos. Todos los mamíferos tienen pulmones. Por lo tanto, todas las ballenas tienen pulmones.	cierto, cierta, válido

Este es un argumento válido, con condición cierta y conclusión cierta.

Todas las arañas tienen diez patas. Todas las criaturas de diez patas tienen alas. Por lo tanto, todas las arañas tienen alas.	falso, falsa, válido

Este **también** es un argumento válido, con condiciones falsas y conclusión falsa. Noten que la falsedad de la primer condición y de conclusión es atribuída por nuestro conocimiento de biología (a *Animal Planet*, decimos en clase). Decir que la segunda condicion es falsa es un poco temerario, puesto que no conocemos criaturas de diez patas (o ¿sí?), pero realmente no importa porque la conjunción se falsifica con la primera condición falsa.

Si yo tuviera todo el dinero de Bill Gates, sería rico. No tengo todo el dinero de Bill Gates. Por lo tanto, no soy rico.	cierto, cierta, inválido

Este es un argumento **inválido** (es decir, no válido) con condiciones ciertas y conclusión cierta (y les juro que es absolutamente cierta si hablan del autor). ¿Por qué es inválido?

Si Bin Laden tuviera todo el dinero de Bill Gates, sería rico. Bin Laden no tiene todo el dinero de Bill Gates. Por lo tanto, Bin Laden no es rico.	cierto, falsa, inválido

Este también es **inválido**, pero sus condiciones son ciertas (suponiendo que Bill es él más adinerado o, incluso, que posee una fortuna diferente) y su conclusión es falsa (por lo que sabemos acerca del origen acomodado de Bin Laden. Alguien podría opinar lo contrario).

Piénselo con conjuntos

¿Será este argumento inválido por la misma razón que el anterior? .

falso, cierta, válido

Todos los peces son mamíferos. Todas las ballenas son peces. Por lo tanto, todas las ballenas son mamíferos.

Este es válido, con condiciones (ambas) falsas y conclusión cierta (de esta última también sabemos gracias a *Animal Planet*).

Piénselo

¿Podemos llegar a la conclusión de validez olvidándonos de peces, ballenas y mamíferos? .

falso, cierta, inválido

Todos los mamíferos tienen alas. Todas las ballenas tienen alas. Por lo tanto, todas las ballenas son mamíferos.

Este es inválido, con condiciones falsas y condición cierta.

falso, falsa, inválido

Todos los mamíferos tienen alas. Todas las ballenas tienen alas. Por lo tanto, todos los mamíferos son ballenas.

Este es inválido, con condiciones falsas y conclusión también falsa.

Ahora, ¿Qué podemos decir de esos 7 argumentos?. ¿En qué nos basamos para el análisis?. ¿Qué ocurre con la 8va. posibilidad?. ¿Cuál es la combinación faltante?. ¿ Cuál es la relación entre la validez de un argumento y la tabla de verdad de las fórmulas **si** .. **entonces**?. Siendo este un libro guía, es razonable que el lector reclame las respuestas a estas preguntas. Descubrirlas por sí solo o sola, sin embargo, será mucho más productivo. Veamos ahora la otra ventaja interesante de la lógica que se describe en este texto.

1.3. Una realización computacional.

teoría de modelos

En lógica clásica se cultivan separadamente esas estructuras semánticas que hemos discutido en la sección anterior, bajo el nombre de **Teorías de Modelos**. Un **modelo** es una interpretación lógica que hace cierta a una fórmula. Por ejemplo, `p y q` es satisfecha si `p = verdad` y `q = verdad`. Por tanto, **esa asignación** es un modelo.

teoría de prueba

La otra parte de la lógica matemática clásica la constituyen las llamadas **Teorías de Prueba** (*Proof Theories*). En lugar de girar en torno a la noción de **DEDUCCION**, como se hace al analizar argumentos y tablas de verdad, las teorías de prueba se concentran en describir ejercicios y estrategias de **DERIVACION**. Derivar es obtener una fórmula a partir de otras a través de transformaciones sintácticas. También se usa el término **INFERIR**.

X es P y P es M

entonces

X es M

Una realización computacional de un razonador lógico es un programa que hace justo eso: inferir. Deriva unas fórmulas a partir de otras. Así, para el computista lógico, derivar es computar. El razonamiento lógico es una forma de computación (si bien, no la única). A los razonadores lógicos se les llama de muchas maneras: motores de inferencia, probadores de teoremas, máquinas lógicas, método de prueba. En todo esos casos, el razonador tiene sus propias reglas para saber que hacer con las fórmulas lógicas (que también suelen ser reglas). A las reglas de un motor de inferencia se les llama: reglas de inferencia. He aquí una muy popular que hasta tiene nombre propio: **modus ponens**

motores de inferencia

reglas de inferencia

Con una fórmula como:

Si A entonces B

y otra formula como:

A

Derive

B

¿Puede ver que se trata de una regla para procesar reglas?. Por eso, algunas veces se dice que es una **meta-regla**[3]

Las teorías de modelos y las teorías de prueba normalmente caracterizan a la lógica clásica (a una lógica o a cada lógica, si uno prefiere hablar de cada realización por separado). Han sido objeto de intenso debate durante largo tiempo y mucho antes de que existieran los actuales computadores. La lógica computacional, sin embargo, ha abierto un nuevo espacio para describir el mismo lenguaje. En la siguiente sección conoceremos de un primer intento por explicar el alcance expresivo de la lógica computacional y conoceremos también una lista de ejemplos de agentes modelados con y en lógica.

1.3.1. Una Teoría de Agentes en Lógica

Hace más de una década, en la escuela de la **programación lógica**, emprendimos un proyecto colectivo para caracterizar a la nueva noción de Agente que irrumpió en varios escenarios de la Inteligencia Artificial. Robert Kowalski es uno de los mejores exponentes de ese proyecto, con su libro *Cómo ser artificialmente inteligente*[4]. El objetivo de Kowalski es explorar el uso de la lógica para modelar agentes. Ese objetivo lo ha llevado a bosquejar una teoría para

programación lógica

[3] Sin más preámbulo, les invito a que prueben uno de los mejores motores de inferencia que se conocen: Prolog. Ya saben como. Por cierto, Gloria "contiene" su propio motor de inferencia (construido sobre Prolog).

[4] publicado libremente en Internet y disponible en español en http://webdelprofesor.ula.ve/ingenieria/jacinto/kowalski/logica-de-agentes.html y como anexo de este texto

explicar qué es un agente y cómo se le puede implementar para efectos de computación. Kowalski ha tratado de aproximar esa misma noción de agente que se ha hecho muy popular en el mundo tecnológico en las últimas décadas.

agente

Resumiendo esa teoría, uno puede decir que un **agente** es una pieza de software para controlar un dispositivo capaz de interactuar con su entorno,

especificación

percibiendo y produciendo cambios en ese entorno. La **especificación lógica** es más general que esa visión resumida y orientada a la máquina. Un agente, en esa teoría lógica, es un proceso auto sostenido y auto dirigido de intercambios entre un estado interno y un estado externo. El proceso es modelado como un programa lógico que puede convertirse en código ejecutable sobre un hardware

bioware

o inclusive, decimos nosotros, sobre **bioware**, la máquinaria biológica de la inteligencia natural. En este sentido, la conceptualización alcanza a describir aspectos de la conducta autónoma de los humanos y otros seres en los que es posible o conveniente distinguir un estado interno de uno externo. Es el ahora

postura intencional

clásico ejercicio de *postura intencional*[21], según la cual la condición de agente es determinada por el ojo del observador.

La teoría renuncia a toda pretensión totalizante: un agente no es sólamente lo que allí se describe y, en particular, nunca se pretende que los humanos (y otros seres conscientes) seamos eso únicamente (pueden verse declaraciones al respecto al principio de los capítulos 1, 6 y 7 del texto de Kowalski). Sin embargo, esa teoría sí pretende restaurar aquella intención originaria de la lógica: *La lógica simbólica fue originalmente desarrollada como un herramienta para mejorar el razonamiento humano.* Ese es el sentido práctico, de utilidad, que se pretende rescatar con la teoría y que, necesariamente, debe estar asociado a una práctica.

1.3.2. Una Práctica de Agentes en Lógica

Inspirados por la teoría de Kowalski y por su afán de demostrar que transciende los símbolos y las operaciones matemáticas, hemos venido recolectando ejemplos particulares de especificaciones, siempre parciales, de agentes en lógica. Cada uno de los ejemplos trata de capturar antes que un agente completo, el rol (papel, partitura) que desempeña cierto agente en cierta circunstancia de interés. El objetivo es, siempre, explicar a un agente y a su forma de pensar sin que la representación matemática interfiera (demasiado).

Burocratín

En la figura 1.1 se muestran las reglas que cierto empleado público en Venezuela, llamado Burocratín, emplearía para decidir cómo atender a un usuario

burocratín

de su servicio:

La situación problema es una en la que Ud necesita que Burocratín le suministre información con cierta urgencia. Debemos explicar las reglas de conducta seguiríamos para hacer que Burocratín le responda en el menor tiempo posible.

antiburocratín

Algo como la figura 1.2:

Con estas reglas, el usuario (uno de nosotros), de *necesitar información de burocratín*, *ubicaría antes una referencia al tema en alguna de las leyes*, *le pediría*

```
1 Si alguien me pide algo entonces yo sigo el procedimiento.
2 Si alguien me pide algo y es algo muy importante, resuelvo
      inmediatamente.
3
4 Para seguir el procedimiento, consulto el manual si existe o
       invento un manual si no existe.
5 Para inventar un manual de procedimiento, solicite una carta
       de alguna autoridad y haga lentamente lo que allí dice (
      para evitar errores).
6 Todo procedimiento se sigue lentamente para evitar errores y
       trampas.
7 Algo es muy importante si se lo menciona en las leyes.
8 Para las cosas importantes existen manuales de procedimiento
      .
```

Figura 1.1: Burocratín

```
1 Si necesito información de burocratín entonces se la pido
      apropiadamente.
2
3 Para pedir información apropiadamente (a burocratín), se lo
      pido y le explico que es muy importante.
4 Para explicarle que es muy importante, antes encuentro una
      referencia del asunto en alguna de las leyes y la
      menciono con pasión.
```

Figura 1.2: El Usuario frente a Burocratín

```
1 Si se acerca un ciudadano en su vehículo entonces lo
      interpelo.
2
3 Para interpelar a un ciudadano, debo pedirle su cédula, su
      licencia, su certificado médico, el carnet de circulación
       y le pido que explique de donde viene y adonde va.
4
5 Si el ciudadano se pone nervioso, le retiro sus documentos
      por un tiempo, examino con cuidado el vehículo y los
      documentos y luego lo interpelo.
6 Si logro establecer una falta o violación de la ley, le
      explico el castigo y le permito negociar.
7 Si no negocia, ejecuto el castigo.
8 Si negociando, el ciudadano propone una ayuda mutua, la
      acepto y le dejo pasar.
```

Figura 1.3: Matraquín

la información a burocratín y *mencionaría con pasión la referencia legal*. Eso *obligaría* a Burocratín a *resolver inmediatamente*, una acción, por cierto, que no está definida en sus reglas (y que, por tanto, puede significar más problemas. Buena Suerte!)

Matraquín

matraquín

Matraquín es un agente que vive en un Universo similar al nuestro en donde sirve como fiscal de tránsito, mientras porta un arma. Sus reglas de conducta incluyen las que se muestran en la figura 1.3:

> ¿Qué pasaría si Ud, conduciendo y desarmado, se encuentra con Matraquín, tiene una falta evidente y Ud es de quienes creen que no se puede negociar desarmado con quienes portan armas?.

Digamos que esto es lo que me ocurre a mí: 1) (En cualquier caso) Matraquín me somete a la primera interpelación, al observar que me acerco. 2) Yo observo el arma y, siguiendo la mencionada creencia, no discuto (entiendo que eso significa que no negocio), pero (supongamos) tampoco me pongo nervioso. 3) Matraquín observa mi evidente falla y procede a explicarme el castigo y me deja negociar. 4) Yo ya habia decidido no negociar, así que guardo silencio. 5) Matraquín observa mi silencio y procede a ejecutar el castigo. El significado de esto último queda para la imaginación del lector.

desenlace

Obviamente esta no es la única forma de **desenlazar la historia**. Todo depende de cuáles otras suposiciones hagamos sobre Matraquín y nosotros mismos. Pero parece que ya hay razones suficientes para preocuparse por esta clase de sistema multi-agentes, ¿verdad?. Considere ahora este otro escenario:

```
METAS:
si hay hembras cerca, establece tu autoridad.
si alguna hembra necesita ayuda, sondéala.
si la hembra que necesita ayuda se muestra receptiva,
    lánzate.
si se aproxima una hembra, abórdala.
si luego de abordar a una hembra se muestra receptiva,
    lánzate.
si se quiebra, suéltaselo.
CREENCIAS:
Para establecer tu autoridad, haz una bravuconada.
Para abordar a una hembra, convérsale sobre cualquier cosa.
Para sondearla, abórdala.
Para lanzarte, invítala a salir.
Para soltárselo, invítala a ....
```

Figura 1.4: Machotín

> ¿Qué pasaría si Ud, conduciendo, se encuentra con Matraquín, tiene una falta evidente y tiene Ud mucha prisa?.

1) Matraquín me somete a la interpelación (de rigor). 2) Esta vez no tengo aquella regla que me impide negociar con una persona armada. Supongamos que trato de explicar mi situación y, supongamos también, que tengo suerte de que Matraquín no interprete que estoy nervioso. 3) Matraquín observa mi evidente falla y procede a explicarme el castigo y me deja negociar. 4) Yo le explico (supongamos que calmadamente) que necesito continuar mi viaje y le pido que me ayude!. 5) Matraquín entiende que estoy negociando y me propone que lo ayude a ayudarme. 6) Yo me muestro dispuesto a la ayuda mutua. 7) y 8) *(estas acciones deben ser censuradas)* . 9) Matraquín acepta mi parte de la ayuda mutua y me deja pasar.

> ¿Cuál es la MORALeja de la historia?

Machotín

Considere la siguiente especificación parcial e informal de un agente que llamamos Machotín en figura 1.4: machotín

Imagine que queremos escribir una historia en la que participe Machotín y en la que se activen, cada una en algún momento, todas sus **metas de mantenimiento**. Nos piden que expliquemos, en esa historia, qué está pensando, observando y haciendo Machotín en cada momento.

Considere la siguiente respuesta:

> "Machotin es un agente en busca de una hembra, bien sea para invitarla a salir o para lanzarse en una aventura con ella. Machotin anda por la calle y observa que hay hembras cerca y establece su

autoridad. Para establecer su autoridad, hace una bravuconada con su pinta de sobrado. Mientras camina con su actitud altiva, observa que una hembra necesita ayuda. La sondea, abordándola y sacándole conversación mientras le ofrece ayuda. La hembra en cuestión se muestra receptiva. Entonces, Machotin aprovecha la oportunidad de lanzarse con una invitación a salir. Luego de cuadrada la cita, Machotin continua su recorrido y observa que se aproxima una hembra. La aborda. Conversa con ella y observa que la hembra es receptiva. Luego se lanza con otra de sus cotorras para invitarla a salir. Mientra sigue con su labia, nota que la hembra se quiebra y le suelta una invitación a su casa para que le lave la ropa aprovechándo que ella hará todo lo que él le pida."[5]

¿Ve cómo y cuando se **dispara** cada regla?

Mataquín

Cierto agente llamado Mataquín está contemplando la posibilidad de cometer un crimen. Quiere vengarse de cierto enemigo suyo pero quiere, desde luego, cometer el crimen perfecto: venganza plena, impunidad total y mantener su imagen pública cubriendo bien sus huellas. Mataquín ha observado que los policías de su ciudad no tienen mucha capacidad de respuesta ante un crimen. No suelen atender sino emergencias extremas. No tienen recursos para investigaciones complejas que involucren a muchas personas, muchos lugares o que impliquen análisis técnicos de cierta sofisticación. Tienen una pésima memoria organizacional, pues llevan todos los registros en papel y los guardan en lugares inseguros. Además, los policias son muy mal pagados y se conocen casos de sobornos, especialmente en crímenes muy complejos o en los que los investigadores se arriesgan mucho y por mucho tiempo. Los ciudadanos de esa ciudad tienen tal desconfianza en su polícia que nunca les ayudan, aún cuando tengan información sobre un crimen.

Suponga ahora que, con esas consideraciones en mente, se nos pide proponer una especificación (informal, pero tan completa como sea posible a partir de la información dada) de las reglas de conducta que debería seguir Mataquín para alcanzar su meta.

mataquín

Considere la respuesta en la figura 1.5 [6]:

La historia resultante, aparte de repulsiva, es bastante obvia, ¿verdad?

Halcones, Burgueses y Palomas: Sobre las aplicaciones de la Teoría de Decisiones en lógica

Para ilustrar una posible aplicación de la teoría de Decisiones, un desarrollo matemático estrechamente asociado con los agentes y los juegos (ver en el libro

[5] Joskally Carrero, 2007

[6] Basada en una propuesta de Victor Malavé, 2007

```
1  METAS
2  si enemigo se acerca entonces establece contacto.
3  si contacto establecido y policía cercano, crea distracción.
4  si contacto establecido y policía distraída, conduce a
       enemigo a piso más alto de edificio cercano.
5  si enemigo en piso alto de edificio cercano, asesínalo.
6  si durante asesinato un policía observó, sobórnalo.
7  CREENCIAS
8  Para establecer contacto, invítalo a lugar público.
9  Para crear distracción, provoca emergencia extrema.
10 Para provocar emergencia extrema, contrata lugareños que
       incendien estación de servicio cercana.
11 Para conducir enemigo a piso alto de edificio cercano,
       propónle un negocio atractivo.
12 Para asesinarlo, provoca caída mortal desde ese piso alto.
13 Para sobornar policia, entrégale mucho dinero.
```

Figura 1.5: Mataquín

de Kowalski[57]), consideren el siguiente ejemplo tomado de un examen (originalmente en [42]). Considere los siguiente tres tipos de personas en una sociedad en la que los individuos compiten por recursos que siempre son de alguien:

La **paloma** nunca trata de hacerse con las posesiones de otros, sino que espera a que sean abandonadas y ella misma abandona un recurso propio tan pronto es atacada. Si dos de ellas compiten por el mismo recurso, entonces una de ella lo obtendrá (por suerte o paciencia) con la misma probabilidad para cada una. El **halcón** siempre trata de apoderarse de los recursos de otros por medio de la agresión y se rinde sólo si recibe graves lesiones. El **burgués** nunca trata de hacerse con las posesiones de otros, sino que espera hasta que son abandonadas, pero defiende su posesión contraatacando hasta que tiene éxito o es derrotado[62].

Cuando dos individuos se encuentran tienen idénticas probabilidades de ganar o perder y de ser, al inicio, dueños ó no del recurso. Por ejemplo, si dos halcones se encuentran, uno de ellos obtendrá el recurso con utilidad U, mientras el otro tendrá graves lesiones, con costo $-Cpelea$. Como ambos tienen la misma oportunidad de ganar, la utilidad esperada para cada uno es $(U-Cpelea)/2$.

¿Cuáles son las utilidades esperadas correspondientes cuando se enfrentan una paloma y un halcón? . Explique

Dadas esas reglas de conducta, la paloma nunca gana contra un halcón (y el halcón nunca pierde contra una paloma). Así que la utilidad para la paloma es 0 y para el halcón es U. Noten que U es el valor *puro* del recurso a conquistar (otra simplificación muy gruesa).

¿Cuáles son las utilidades esperadas correspondientes cuando se enfrentan una paloma y un burgués? [7] Explique

"Cuando un burgués encuentra a otro individuo, cada uno de ellos puede ser el dueño lícito del recurso rivalizado. Por ejemplo, si una paloma se encuentra con un burgués y ambos compiten por el mismo recurso, entonces tenemos dos posibilidades igualmente probables:

- Caso1: Si el burgués es el dueño legal del recurso, entonces conserva el recurso (*Beneficio=U*), y la paloma no se lleva nada (*Beneficio=0*).
- Caso2: Si la paloma es la dueña legal del recurso, ambos deben esperar hasta que alguno renuncie (lo cuál les cuesta digamos *Cespera*. Beneficio = -*Cespera*) y, entonces, cada uno de ellos tiene igual probabilidad de lograr el recurso. Así que el resultado local esperado es *U/2 - Cespera* para cada uno.

De esta manera, el resultado global es $\frac{U+(\frac{U}{2}-Cespera)}{2}$ para el burgués y $\frac{0+(\frac{U}{2}-Cespera)}{2}$ para la paloma" (.ibid).

Noten que la utilidad global (para cada agente) es igual a $Caso1 * Prob1 + Caso2 * Prob2$, donde $Prob1 = Prob2 = 1/2$, pues se nos dice que ambos casos son igualmente probables.

En el caso2, además, juega un papel la acción de esperar a que el recurso sea abandonado (sí, esperar también es una acción posible). Cómo no se sabe cuál de los dos va a renunciar primero en esa esperar, se dice que su probabilidad local asociada es *1/2* y como el beneficio en disputa es *U*, la utilidad esperada local es de *U/2*, faltando por descontar el precio de esperar que ambos pagan igualmente (*Cespera*).

utilidad esperada anidada

Este ejemplo es particularmente interesante porque muestra el uso del concepto de la utilidad esperada *anidada* (cálculo de utilidad esperada sobre otra utilidad esperada).

¿Cómo sería la utilidad global para ambos agentes?.

Agentes, Ontologías y la Web Semántica

Los ejemplos anteriores se referieren todos a descripciones de agentes en ciertos roles muy puntuales. Pareciera que para tener un agente completo y funcional se requiere de un esfuerzo mucho mayor. Lo cierto es que esto último, incluso para efectos de prestación de servicios, no necesariamente es el caso.

Lo que queremos es mostrar a continuación son dos ejemplos de modelos de agentes que implementan otras dos nociones que se han vuelto fundamentales en la Web: Los **metadatos** y el **razonamiento no monótono**.

Una de las primeras aplicaciones naturales de la tecnología de la **Web Semántica** es la búsqueda de documentos. Con recursos como repositorios **RDF** y **OWL** en la Web, uno puede imaginar un agente que nos ayude a ubicar documentos a partir del tipo de conocimiento que contienen (y que buscamos).

[7] Cito la explicación dada en Simulación para las Ciencias Sociales de Nigel Gilbert y Klaus Troitzsch

```
si me preguntan ¿Quien Opina sobre Tema? y Quien es una
    variante gramatical de Quienes y Opina es una variante
    semántica de opinar y el análisis del Tema arroja estos
    Descriptores entonces Rastreo la Web buscando todos los
    Autores de documentos con esos Descriptores.
```

Figura 1.6: Agente Web Semántica

Por ejemplo, para responder preguntas como *¿Quienes han escrito sobre el legado epistolar del Libertador de Venezuela?*, uno puede pensar en un agente con una META[8] como la que se muestra en la figura 1.6. agente web semántica

Esta meta requerirá, desde luego, mecanismos de soporte (que podrían convertirse en creencias del agente) para realizar el análisis del Tema que produce los Descriptores y el Rastreo de la Web.

En la tarea de `análisis del tema`, sería muy útil contar con un mecanismo que le permita al agente, en el caso particular de nuestra pregunta, traducir *legado epistolar* en todos sus **sinónimos** e **hipónimos**. Así el agente sabría que quien quiera que declare haber escrito sobre *las cartas del Libertador* es un autor a revisar. Algo similar habría que hacer con *Libertador de Venezuela* para que la máquina entienda que se trata de Simón Bolívar. Esto último requeriría, desde luego, un mecanismo adecuado para el caso *venezolano-colombiano-ecuatoriano-peruano-boliviano*.

Esta clase de relaciones de **sinonímia**, tanto universales como locales, son posibles con sistemas como **WordNET** una especie de diccionario y tesauro automático. WordNET codifica una **ontología** que le permite relacionar los términos. Desafortunadamente, esa ontología no está escrita en ninguno de los lenguajes ontológicos de la Web Semántica, pero aún así puede ser útil.

La segunda tarea es también muy interesante. Requiere que nuestro agente disponga de medios para identificar entre la enorme cantidad de documentos contenidos en la Web, los autores y los descriptores de esos documentos. Esa información que refiere el contenido y otros atributos de los datos es conocida como **metadata** y es fundamental para que las máquinas puedan contribuir a la gestión de los **repositorios de conocimiento**. Hay muchos esfuerzos en la dirección de proveernos de un standard de metadatos en la forma de una ontología que explique que son y que contienen los documentos formales. El Dublin Core[1] es quizás el más avanzado de esos esfuerzos. **metadata**

Podemos ver, en ese primer ejemplo de agente para la Web Semántica, como se incorporaría el manejo de significado en un agente que atienda consultas en la Web. Hay, sin embargo, un siguiente nivel de complejidad en el manejo de significados para atender consultas. Algo que es bien conocido en Bases de Datos desde hace años: El manejo de datos temporales y el razonamiento no monótono.

Se llama **razonamiento no monótono** a aquel que cambia de opinión. **razonamiento no monótono**

[8] Agradecemos a Icaro Alzuru por motivar este ejemplo con su trabajo de maestría en Alejandría Inteligente: Un experimento en la Web Semántica con un sistema de gestión de

```
si me solicitan los Propietarios de terrenos en un Área
    entonces reportar Propietarios del Área.

Para reportar Propietarios del Área haga transforme la
    descripción del Área en un Área Geográfica y Los
    Propietarios son los sujetos para quienes se cumple (
    ahora) que poseen propiedades contenidas en el Área
    Geográfica.
```

Figura 1.7: Gea: El Agente Catastral

Típicamente, un agente con cierto cúmulo de creencias alcanza ciertas conclusiones, pero si esas creencias cambian (debido quizás, a un cambio en el mundo), las conclusiones también podrían cambiar.

Los lectores del libro del Profesor Kowalski podrán asociar esa forma de razonar con esos lenguajes lógicos que estudiaron en el capítulo sobre el cambiante mundo (**lógicas modales**, el **cálculo de situaciones** y el **cálculo de eventos**). Permítanme, sin embargo, un último ejemplo con un agente que implementa una forma de razonamiento no monótono indispensable para la aplicación a su cargo:

Imaginen un agente que maneja un repositorio de información catastral elemental[9]. Entre muchos otros tipos de consulta, este agente debe poder explicar quienes son los propietarios de los terrenos en un área dada. La meta de mantenimiento y creencia correspondiente para este agente es simple se muestra en fig 1.7.

gea

La última sub-meta (*Los Propietarios..*) es una consulta que se podría responder con una versión extendida del cálculo de situaciones, o el de eventos, configurada para lidiar con propiedades y áreas geográficas. Lo importante en este caso es que si a este agente se le hace la consulta en un momento dado, su respuesta bien puede variar respecto a otro momento si han ocurrido eventos que cambien las relaciones propiedad-propietario en ese área geográfica.

Desde luego, este agente tendría que operar con conocimiento que le permita asociar cualquier descripción de un área (por ejemplo, en términos políticos: parroquia, municipio, país, etc) con un conjunto de coordenadas standard. Más importante aún, el agente requeriría de un registro sistemático de los eventos de compra y venta de propiedades que incluya los detalles de cada propietario y propiedad (incluyendo la ubicación de cada una, desde luego). Todo este conocimiento bien puede estar almacenado en un repositorio asociado con una ontología de registro catastral (por ejemplo, un archivo, con marcaciones en **OWL** específicas para el problema).

Esos dos ejemplos, informales y muy simples, pueden servir para ilustrar la

documentos http://webdelprofesor.ula.ve/ingenieria/jacinto/tesis/2007-feb-msc-icaro-alzuru.pdf

[9]Agradecemos a Nelcy Patricia Piña por motivar este ejemplo con su trabajo de maestría en Una Ontología para Manejo de Información Catastral http://webdelprofesor.ula.ve/ingenieria/jacinto/tesis/2006-mayo-msc-nelcy-pina.pdf

```
si mepregunta(Preg), significa(Preg, Sig), not(Sig=[no,
    entiendo|]) entonces respondoa(Sig).
si mepregunta(Preg), significa(Preg, Sig), Sig=[no, entiendo
    |] entonces disculpas(Sig).
para respondoa(S) haga rastrear(S), mostrar(S).
```

Figura 1.8: Kally

```
pregunta([V|S]) --> prointerrog, vatributivo(V), satributivo
    (S).
pregunta([V|S]) --> advinterrog, sverbal(V), satributivo(S).
pregunta([V|S]) --> sverbal(V), satributivo(S).
pregunta([no, entiendo, tu, pregunta, sobre|S]) --> atributo
    (S).

sverbal(V) --> vmodal, vinfinitivo(V).
sverbal(V) --> vinfinitivo(V).
sverbal(V) --> vconjugado(V).
satributivo(S) --> especificador, atributo(S).
satributivo(S) --> atributo(S).
```

Figura 1.9: La Gramática Española de Kally

extraordinaria riqueza de posibilidades en este encuentro entre la Web, la lógica y los agentes.

Kally: Agente para atención a usuarios y enseñanza isocéntrica

Los agentes Web Semántica y Gea Catastral dejan ver someramente la importancia del procesamiento del lenguaje natural en la Inteligencia Artificial. Es, todavía, una de las fronteras más activas de investigación y promete grandes avances, siempre que logremos contener el entusiamo que produce la posibilidad de interactuar con el computador en los mismos términos (lenguaje) que usamos con otros humanos.

Kally[10] es un pequeño agente diseñado para asistir a los humanos en el uso de una herramienta Ofimática, OpenOffice, interactuan en una forma controladada de lenguaje natural. En fig 1.8, 1.9 y 1.10 se muestra el código de Kally en una versión preliminar de Gloria (ver capítulo **??**).

Kally tiene sólo un par de reglas simples, pero tiene también una gramática interconstruída (el predicado `pregunta`) que le permite entender ciertas preguntas simples en Español y establecer "su significado" (es decir, lo que debe hacer al respecto). La gramática está formalizada usando un lenguaje conocido como DCG, *Definite Clause Grammar*.

[10] `http://kally.sourceforge.net`

```
prointerrog --> ['qué'];[que];[cual];[cuales]; ['cuáles']; [
    cuantos];[cuantas].
especificador --> [el];[la];[lo];[los];[las];[un]; [una];[
    unos];[unas];[mi];[mis].
vatributivo(es) --> [es];[son];[significa].
advinterrog --> ['cómo'];[como];[cuando]; [donde];[por, que
    ]; [por, 'qué'].
vmodal --> [puedo];[puede];[podemos].
vinfinitivo(utilizar) --> [utilizar].
vinfinitivo(abrir) --> [abrir].
vinfinitivo(salvar) --> [guardar].
vinfinitivo(salvar) --> [salvar].
vinfinitivo(crear) --> [crear].
vinfinitivo(instalar) --> [instalar].
vinfinitivo(instalar) --> [reinstalar].
vinfinitivo(definir) --> [definir].
vinfinitivo(realizar) --> [realizar].
vinfinitivo(agregar) --> [agregar].
vinfinitivo(agregar) --> [anadir].
vinfinitivo(exportar) --> [exportar].
vinfinitivo(insertar) --> [insertar].
vconjugado(guardar) --> [guarda].
vconjugado(desinstalar) --> [desinstalo].
vconjugado(ubicar) --> [existe].
vconjugado(usar) --> [uso].
vconjugado(cambiar) --> [cambio].
vconjugado(instalar) --> [instalo].
vconjugado(crear) --> [creo].
vconjugado(estar) --> [esta].
vconjugado(actualizar) --> [actualizo].
prep-->[a];[como];[con];[de];[desde];[durante];[en];[entre
    ];[hacia];[mediante];[para];[por];[sin];[sobre].
atributo(T,T,).
```

Figura 1.10: El Léxico Español de Kally

El Cálculo de Eventos y El disparo de Yale

El **sentido común**, esa intuición básica que nos permite actuar correctamente sin, aparentemente, pensarlo demasiado, se ha constituído en uno de los mayores desafíos en la IA, al punto que algunos problemas referenciales al tema se han vuelto muy populares. sentido común

En **la historia del Disparo de Yale**, una persona es asesinada de un disparo. Para una posible formalización uno considera tres acciones: *cargar (el arma)*, *esperar* y *disparar*; y dos propiedades: *vive* y *cargada*. La acción cargar coloca una bala en el arma. La víctima muere después del disparo, siempre que el arma esté cargada en ese instante. Se asume que la víctima vive al principio y también que el arma está descargada entonces.

cálculo de eventos

Considere el axioma central del Cálculo de Eventos :

0) Se cumple un Hecho en un Momento si un Evento ocurrió antes y ese Evento inició el Hecho y no hay Otro evento que ocurra luego del Evento iniciador, antes del Momento y que termine el Hecho.

Suponga que nos piden completar este axioma con las reglas y hecho necesarios para describir el problema del Disparo de Yale. Con todas esas reglas, debemos probar (con un argumento formal) que: no se cumple que (la víctima) vive en en último momento del cuento.

La historia se puede describir, con algo de formalización, así:

1) La (presunta) víctima nace en el momento 0.

2) No es cierto que el arma está cargada en el momento 0.

3) El (presunto) asesino carga el arma en el momento 1.

4) El asesino espera entre el momento 1 y el momento 2.

5) El asesino dispara en el momento 2.

Con la misma terminología podemos escribir las reglas complementarias específicas a esta situación:

6) El evento `Un agente (le) dispara` en el momento T termina el hecho `la víctima vive` en T si se cumple que el arma está cargada antes de T.

7) El evento `Un agente (le) dispara` en el momento T termina el hecho `el arma esta cargada` en T si se cumple que el arma está cargada antes de T.

8) El evento `Un agente carga el arma` en el momento T inicia el hecho `el arma está cargada` en el momento T.

9) El evento `Un agente nace` en el momento T inicia el hecho `el agente vive` en el momento T.

Con esas reglas razonamos hacia atrás a partir del hecho de que no se cumple que la víctima vive en el último momento del cuento. Digamos que ese momento es momento 3.

10) no se cumple que la víctima vive en el momento 3.

de 10) y 0) se obtiene:

11) no (es cierto que) un evento ocurrió antes del momento 3 y ese evento inició (el hecho de) víctima vive y no hay otro evento que ocurra luego de ese evento iniciador y antes del momento 3 y que termine el hecho de que la víctima vive.

Pero 11) se puede reescribir (por equivalencia lógica `no (a y b) = (no a) o (no b)`, así:

11') no es cierto que un evento ocurrió antes del momento 3, ó no inició ese evento víctima vive, ó hay otro evento que ocurra luego de ese evento iniciador y antes del momento 3 y que termine el hecho.

11') se transforma en 12) al considerar a 1),

12) hay otro evento que ocurra luego del evento iniciador `víctima nace` en momento 0 y antes del momento 3 y que termine el hecho de que `la víctima vive`.

Pero, por 5), sabemos que hay un candidato a posible terminador, con lo que 12) se reduce a 14), luego de considerar a 13)

13) `el asesino dispara` en momento 2 termina el hecho de que `la víctima vive`.

Así, por 6), pasamos a preguntarnos si 14)

14) el arma está cargada en el momento 2.

y gracias a 0), 3) y 2) (y que la acción de esperar no cambia nada), podemos probar a 14 con un par de pasos similares a los anteriores.

Fin de la prueba

Visto así parece increíble que un computador lo pueda resolver. Pero lo resuelve.

El Agente Artista

Considere las reglas en la figura1.3.2 que son parte de un agente artista holístico.

Suponiendo que este agente observa `inspiracion_sistemica`, vemos como, paso a paso, este agente artista produciría un plan para la creación de una obra con la mayor cantidad de elementos posibles. Asegúrese de explicar en que consiste la `Obra` planeada.

Razonando hacia adelante con la primer regla, el agente produce su primer meta de logro:

```
si inspiracion_sistemica entonces crear_obra_tipo(holistica,
    Obra).

para crear_obra_tipo(holistica, Obra) haga
    seleccionar_elementos(holistica, Elementos), plasmar(
    Elementos, Obra).

para seleccionar_elementos(Tipo, [Elemento]) haga
    tomar_elemento(Tipo, Elemento).
para seleccionar_elementos(Tipo, [R|Resto]) haga
    tomar_elemento(Tipo, R), seleccionar_elementos(Tipo,
    Resto).

para tomar_elemento(holistica, nota(do)) haga true.
para tomar_elemento(holistica, color(azul)) haga true.
para tomar_elemento(holistica, textura(suave)) haga true.

para plasmar([nota(do)], sonidobajo) haga true.
para plasmar([nota(do), color(azul)], azulprofundo) haga
    true.
para plasmar([nota(do), color(azul), textura(suave)],
    extasis ) haga true.
```

Figura 1.11: El Artista

1) `crear_obra_tipo(holistica, Obra)`

noten que la escribimos en notación Prolog para hacer más fácil la representación subsiguiente. Razonando hacia atrás a partir de 1) usando segunda regla obtenemos:

2) `seleccionar_elementos(holistica, Elementos)` y
`plasmar(Elementos, Obra)`.

Para resolver la primer submeta de 2) tenemos dos reglas (la tercera y cuarta arriba), por lo que 2) se convierte en 3)

3) `tomar_elemento(holistica, [Elemento])` y
`plasmar([Elementos], Obra)`, o
`tomar_elemento(holística, R)` y
`seleccionar_elementos(holistica, Resto)` y
`plasmar([R|Resto], Obra)`.

Uno podría, si no es cuidadoso con lo que se nos pide, terminar la prueba con la primera opción (antes del ,o). Pero eso produciría una obra con un solo elemento. Eso es justamente lo que NO se nos pide. Por esta razón, un agente inteligente buscaría la segunda alternativa y, luego de sucesivos refinamientos obtendría algo como 4) búsqueda

4) `tomar_elemento(holistica, [Elemento])` y
`plasmar([Elementos], Obra)`, o
`tomar_elemento(holistica, nota(do))` y
`tomar_elemento(holistica, color(azul))` y
`tomar_elemento(holistica, textura(suave))` y
`plasmar([nota(do), color(azul), textura(suave)], Obra)`

el cual, al ser ejecutado, implicaría *Obra* = *extasis* que consiste en esta lista de elementos:

```
[nota(do), color(azul), textura(suave)]
```

El truco acá era observar que una reducción hacia atrás, izquierda a derecha (tal como lo hace Prolog) no nos conduce a la obra pretendida (que debe tener el máximo numero de elementos posibles).

Fin de la prueba

Como lidiar con la inflación

Presentamos el siguiente ejemplo en este capítulo para ilustrar un caso mucho más complejo (y humano) del modelado lógico de un agente. El Prof Carlos Domingo se planteó el siguiente conjunto de reglas para enfrentar el problema de la inflación que afectaría sus ingresos como académico[11]

> "Meta: *Evitar las pérdidas en mis ahorros causadas por la inflación de este año.*
>
> Creencias:
>
> 1. Durante el año en curso, se espera que los precios aumenten en un 30 por ciento en artículos como libros, del hogar, medicinas y ropas. En los artículos básicos (comida y transporte nacional), el incremento puede ser menor al 10 por ciento debido al control de precios y, además, es probable que sea compensado con un aumento salarial. Pero mis ahorros anteriores podrían ser afectados por la inflación.
> 2. Las tasas de interés estimadas para este año en un 13 por ciento no compensarán la inflación. Por lo tanto, depositar el dinero el banco no es una buena estrategia para proteger mis ahorros. Otras alternativas deben ser consideradas.
>
> Restricciones:
>
> 1. No debería tomarme mucho tiempo para encontrar una solución. Tengo trabajo pendiente.
> 2. No debo comprometerme a trabajos posteriores a este año.
> 3. Cualquiera que sea la decisión no deben disminuir mi ingreso total anual."

[11] Tomado de las notas de curso del Prof. Carlos Domingo, ULA. 2005

Tomando eso en consideración, Carlos Domingo decide que sus metas inmediatas son:

1. "Conseguir un ingreso adicional que no dependa de sus ahorros para compensar por la inflación. Para ello puedo:

   a) Participar en proyectos universitarios extra que le den un ingreso extra (obvenciones). Esto podría chocar con la restricción 1 porque es dificil conseguir proyectos en mi área de trabajo.

   b) Ocuparme en negocios privados que siempre están disponibles. Esto choca con todas las restricciones.

2. Invertir mis ahorros (total o parcialmente) para obtener beneficios equivalentes a las pérdidas esperadas. Para ello puedo:

   a) Comprar acciones (pero esto requiere tiempo y puede contrariar a 1 y probablemente a 2).

   b) Comprar bienes 'durables' para vender en el futuro. Tendría que tener cuidado con los precios futuros y la obsolescencia.

   c) Mudar mis ahorros a moneda extranjera. Difícil debido al control cambiario y muy riesgoso debido a una posible devaluación de esa moneda extranjera. "

El Prof. Carlos ha concluído que una combinación de 1a y 2b es la mejor decisión posible (una combinación de ambas estrategias minimizaría el riesgo de fallar completamente).

Suponga que Carlos Domingo posee una cierta cantidad en AHORROS al INICIO del año y monto total por su SALARIO en todo el año. Ayudemos a Carlos a proponer una justificación formal de su decisión. Un análisis de utilidad completo supondría estimar las utilidades esperadas de cada opción. Esto puede consumir mucho tiempo y, en algunos casos, aún así no mejoraría la evaluación. Así que, atendiendo a las conclusiones del mismo Profesor, uno se puede concentrar en las dos alternativas de acción que le parecen las mejores decisiones.

Carlos fallaría si sus ahorros son consumidos por la inflación. Es decir, si al final del año sus AHORROS no satisfacen:

$$AHORROS \leqq AHORROS\_INICIALES * 1{,}3 \tag{1.1}$$

El valor a la derecha de la desigualdad es el mínimo valor razonable para la utilidad esperada global de las acciones que se plantea el profesor. Esas acciones son:

1. ganar obvenciones por proyectos extras y

2. invertir en durables.

La probabilidad de ganar obvenciones por proyectos extras, $ProbObv$, es menor al 0.5. Así que la utilidad estimada de la acción 1 es:

$$utilidad_1 = ProbObv * OBVENCIONES \tag{1.2}$$

Invertir en durables significa: 1.- Adquirirlos a buen precio y, desde luego, con un costo no superior al monto en ahorros en ese momento y 2.- venderlos a un precio justo. La probabilidad de adquirirlos a buen precio es $ProbAdDur$. La probabilidad de venderlos es $ProbVenta$. Así, la utilidad de acción 2, suponiendo independencia entre compra y venta, puede ser aproximada por:

$$\begin{aligned} utilidad_2 = \quad & ProbAdDur * ProbVenda \\ & *(PRECIOJUSTO - \\ & AHORROS\_ESPERADOS) \end{aligned} \tag{1.3}$$

donde $AHORROS\_ESPERADOS$ es una variable que agrega el $SALARIO$, un probable aumento, la inflación y los gastos de cada tipo en que incurra el Profesor. Por ejemplo:

$$\begin{aligned} AHORROS\_ESPERADOS \quad = & SALARIO \\ & + ProbAumento * AUMENTO \\ & - GASTOS\_BASICOS * 1{,}10 \\ & - OTROS\_GASTOS * 1{,}3 \end{aligned} \tag{1.4}$$

Así que los escenarios que satisfacen la siguiente fórmula son los que justificarían la decisión de Carlos:

$$\begin{aligned} & ProbObv * OBVENCIONES + \\ & ProbAdDur * ProbVenda * (SALARIO \\ & + ProbAumento * AUMENTO \\ & - GASTOS\_BASICOS * 1{,}10 \\ & - OTROS\_GASTOS * 1{,}3) \quad > \\ & AHORROS\_INICIALES * 1{,}3 \end{aligned} \tag{1.5}$$

¿simple?

Rentín

Los ejemplos anteriores ilustran cómo la lógica combina muy bien con otras herramientas matemáticas para abordar problemas complejos. Permítannos completar esta colección de ejemplos con uno que, a nuestro juicio, pulsa la expresividad del lenguaje contra un tipo extremo de complejidad: la representación

```
1  Metas: Si puedo, abuso.
2       Si me da flojera, no trabajo.

1  Creencias: Abuso si gano sin merecerlo.
2      Puedo si nadie sospecha de mí.
3      Nadie sospecha de mí si me oculto tras la complejidad.
4      Me oculto tras la complejidad cuando toda evaluación es
           compleja y el sistema judicial no funciona.
5      Una evaluación se complica si no hay memoria.
6      No hay memoria si los datos no se colectan o se pierden.
7      No lo merezco si no trabajo.
8     El sistema judicial no funciona si no hay jueces confiables y
          las exenciones y excepciones son arbitrarias.
9      Las exenciones y excepciones son arbitrarias si la evaluación
           es compleja.
10     Los datos sólo se colectan y se preservan si hay un observador
           independiente encargado de cuidarlos.
```

Figura 1.12: Rentín

del discurso y la acción política. Tenemos, en Venezuela, un problema muy complejo al cual sólo se le ha llegado con discursos oscuros o con referencias claras pero muy tímidas: el problema del rentismo. Algunos lo caracterizan como un fenómeno económico que está asociado a la dependencia de la economía nacional de un producto de altísima renta, el petróleo, que genera unos beneficios extraordinarios respecto al esfuerzo de producción (lo cuál lo convierte en un fenómeno económico) y, según alegan, induce una conducta peculiar y una dudosa moralidad en muchos venezolanos. Rentín

El desafío que nos proponemos acá es triple. Primero, caracterizar, como hicimos al principio del capítulo, al agente responsable de tal conducta con la colección de reglas que se muestra en la figura 1.12. El desafío se duplica porque esperamos que el lector pueda separarse del contexto socioeconómico que dibuja el párrafo anterior y aún así ver la asociación con el agente. Pero, para que sea aún más interesante (el triple desafío), vamos a razonar como Rentín en una situación particular, ilustrándo aquella noción de pensamiento preactivo, intermedia entre pensamiento reactivo y pensamiento proactivo en la teoría de Kowalski.

Por simplicidad, sin embargo, permítannos también dispensar con el protocolo de observación, pensamiento y acción y suponer que el agente Rentín está, mayormente, pensando. Así, a partir de su primer meta:

```
Si puedo, abuso
```

y su segunda creencia:

```
Puedo si nadie sospecha de mí.
```

obtiene:

```
Si nadie sospecha de mí, abuso
```

y un paso más allá, gracias a la creencia:

```
Nadie sospecha de mí si me oculto tras la complejidad.
```

conduce a:

```
Si me oculto tras la complejidad, abuso.
```

Continuando con esa misma "línea de pensamiento" y usando la creencia:

```
Me oculto tras la complejidad cuando
    toda evaluación es compleja
 y el sistema judicial no funciona.
```

aquella regla se reduce a:

```
Si toda evaluación es compleja y
    el sistema judicial no funciona, abuso
```

La quinta creencia, con algo de ajuste sintáctico/semántico (¿complicado es igual que complejo?) permitiría:

```
Si no hay memoria y
    el sistema judicial no funciona, abuso
```

La sexta creencia, conduce a:

```
Si (los datos no se colectan o se pierden) y
    el sistema judicial no funciona, abuso
```

Esa disjunción en una de la condiciones de la regla podría requerir algo de tratamiento lógico. Sin embargo, en esta ocasión, otra regla, la última creencia (con algo de ajuste semántico), ofrece una forma rápida de reducir a:

```
Si no hay un observador independiente de los datos y
    el sistema judicial no funciona, abuso
```

Este es un buen lugar para detenerse a observar las transformaciones que deben tener lugar para acoplar (o aparear) esas dos reglas (usando la última creencia). Es importante notar que hay un "sólo" en el texto de la creencia.

El lector acucioso podrá verificar que la segunda condición de esta última regla se disuelve en la primera (pruebe reduciéndolas con las creencias penúltima y antepenúltima y, bueno, suponga que no hay jueces confiables). Con lo que la regla se transforma en:

```
Si no hay un observador independiente de los datos, abuso
```

Este es también es un buen lugar para plantearse preguntas sobre las capacidades del agente. Qué puede este agente hacer para alcanzar sus metas. Uno podría especular, por ejemplo, que "hay un observador .." es una observación posible para el agente (es una proposición **observable**). Si es así, esta regla quedaría latente como una meta derivada en espera de activarse. Cuandoquiera que "no haya un observador independiente de los datos" el agente podrá deducir la meta a simplemente:

observable

```
abuso
```

que puede ser entonces (o antes) sometida a su propio proceso de reducción usando la primer creencia (con algo de flexibilidad semántica también):

```
no lo merezco y gano
```

Por otro lado, quizás no sea demasiado exagerado suponer que "para que no haya ningún observador independiente de los datos", el agente dispone de la acción "eliminar a todos los observadores de los datos". En este caso, podría usar la regla para derivar:

```
Si elimino todos los observadores independientes, abuso
```

y desde allí, el plan parcial (porque podríamos pensar que no está completamente reducido a acciones):

```
elimino (todos los observadores) y no lo merezco y gano
```

La séptima creencia nos acerca un poco más a un plan completo:

```
elimino y no trabajo y gano
```

Este plan tiene el inconveniente de depender de una meta, "no trabajo", que quizás (sólo quizás) no está entre las capacidades del agente (quizás si entre las incapacidades, pero digamos que no es algo que el agente haga). Este es un plan preactivado o preactivo. Si existiera otra forma de establecer (alcanzar) esa meta, el plan podría completarse o reducirse solamente a acciones del agente. Para ello podría servir las segunda meta y la observación de que "me da flojera", con lo que el plan se reduciría a:

```
elimino y tengo flojera y gano
```

que incluye esa observación que, como acabamos de decir, puede provenir del ambiente (en este caso, del cuerpo del agente si se trata de una forma de cansancio).

Este es, se admite, un ejercicio de reducción extrema de una realidad muy compleja. Estamos seguros, sin embargo, que muchos lectores harán la conexión entre esta caricatura de plan y el común y sufrido concepto de "desidia" que contamina algunos espacios públicos con una combinación de pereza y un enorme desdén por los datos y la información.

Hay todavía varios otros ejemplos por discutir. En particular, hemos reservado espacio separado para los agentes del cambio estructural en el capítulo 2 y para los motores de minería de datos y los agentes aprendices, en el capítulo3. Ejemplos estos que cuentan ya con una realización computacional completa (disponible en el repositorio Galatea).

Capítulo 2

Simulación con Agentes y Lógica

Introducción

En capítulo anterior presentamos una práctica de lógica. En este presentaremos una práctica de simulación. Suponemos que el lector tiene algunos conocimientos mínimos del tema, que podrá refrescar con referencias que iremos dictando a lo largo del capítulo. Como quiera también que una práctica requiere conocimientos mínimos de un lenguaje, referimos al lector al apéndice A para una breve introducción al lenguaje de programación que empleamos en el texto.

2.1. Directo al grano

Este segundo capítulo tiene como propósito dirigir a un simulista en la creación de un modelo de simulación Galatea [1]. Nos proponemos hacer eso en dos fases. En la primera, mostramos paso a paso cómo codificar en Java un modelo básico para Galatea. En la segunda fase, repetimos el ejercicio, pero esta vez sobre un modelo de simulación multi-agente de tiempo discreto.

La intención trascendente del capítulo es motivar ejercicios de "hágalo Ud mismo o misma", luego de darle al simulista las herramientas lingüísticas básicas. Explicar la infraestructura de simulación que las soporta nos tomará tiempo en el resto del libro. El lector no debería preocuparse, entretanto, por entender los conceptos subyacentes, más allá de la semántica operacional de los lenguajes de simulación.

En otro lugar explicamos que Galatea es una *familia de lenguajes de simulación*. En estas primeras secciones, por simplicidad, sólo usamos Java.

[1] Repositorio Galatea `http://galatea.sourceforge.net`

2.2. El primer modelo computacional

Galatea heredó la semántica de Glider[22]. Entre varias otras cosas, esto significa que Galatea es, como Glider, *orientado a la red*. Esto, a su vez, refleja una postura particular de parte del modelista que codifica el modelo computacional en Galatea. En breve, significa que el modelista identificará componentes del sistema modelado y los ordenará en una red, cuyos enlaces corresponden a las vías por las que esos componentes se comunican. La comunicación se realiza, normalmente (aunque no exclusivamente como veremos en los primeros ejemplos) por medio de pase de mensajes.

La simulación se convierte en la reproducción del desenvolmiento de esos componentes y de la interacción entre ellos. Esta es, pura y simple, la idea original de la orientación por objetos que comentabamos al principio del capítulo anterior.

En Glider, y en Galatea, esos componentes se denominan **nodos**. Los hay de 7 tipos: `Gate`, `Line`, `Input`, `Decision`, `Exit`, `Resource`, `Continuous` y `Autonomous`[2]

Estos 7 tipos son estereotipos de componentes de sistemas que el lenguaje ofrece "ya pensandos" al modelista. El modelista, por ejemplo, no tiene que pensar en cómo caracterizar un componente de tipo recurso. Sólo debe preocuparse de los detalles específicos de los recursos en el sistema a modelar, para luego incluir los correspondientes nodos tipo `Resource` en su modelo computacional. Esta es la manifestación de otra idea central de la orientación por objetos: la reutilización de código.

Una de las líneas de desarrollo de Galatea apunta a la creación de un programa traductor para la plataforma, como explicaremos en un capítulo posterior. Ese interpretador nos permitirá tomar las especificaciones de nodos en un formato de texto muy similar al que usa Glider (lo llamaremos **nivel Galatea**) y vertérlas en código Java, primero y, por esta vía, en programas ejecutables después.

Aún con ese traductor, sin embargo, siempre es posible escribir los programas Galatea, respetando la semántica Glider, directamente en Java (lo llamaremos **nivel Java**). Hacerlo así tiene una ventaja adicional: los conceptos de la orientación por objetos translucen por doquier.

Por ejemplo, para definir un tipo de nodo, el modelista que crea una clase Java que es una subclase de la clase `Node` del paquete `galatea.glider`. Con algunos ajuste de atributos y métodos, esa clase Java se convierte en un `Node` de alguno de los tipos Glider. Para *armar* el simulador a nivel Java, el modelista usa las clases así definas para generar los objetos que representan a los nodos del sistema.

De esta forma, lo que llamamos un modelo computacional Galatea, a nivel Java, es un conjunto de subclases de `Node` y, por lo menos una clase en donde se definen las variables globales del sistema y se coloca el programa principal del simulador.

[2]Los primeros 5 dan cuenta del nombre del lenguaje GLIDER. Note el lector que Galatea también es un acrónimo, sólo que seleccionado para rendir tributo al género femenino como nos enseñaron varios maestros de la lengua española.

Todo esto se ilustra en las siguientes subsecciones, comenzando por las subclases de `Node`.

2.2.1. Nodos: componentes del sistema a simular

Tomemos la caracterización más primitiva de un sistema dinámico: Una máquina abstracta definida por un conjunto de variables de estado y una función de transición, normalmente etiquetada con la letra griega δ, que actualiza ese estado a medida que pasa el tiempo. Una caracterización tal, es *monolítica*: no hay sino un solo componente y sus régimen de cambio es el que dicta δ.

La especificación de ese único componente del sistema sería, en el Nivel Java, algo como:

Como el lector reconocerá (seguramente luego de leer el apéndice A), ese código contiene la especificación de una clase Java, `Delta`, del paquete `contrib.Grano`, que acompaña la distribución Galatea (como código libre).

Noten que esta clase "importa" (usa) servicios de los paquetes `galatea` y `galatea.glider`. El primero de estos paquetes, como se dice en el apéndice A, contiene servicios genéricos de toda la plataforma (como la clase `List`). Ese paquete no se usa en este ejemplo tan simple. Pero seguramente será importante en ejemplos más complejos.

El segundo paquete almacena toda la colección de objetos, clases necesarios para que la plataforma Galatea ofrezca los mismos servicios que el tradicional compilador Glider, incluyendo el núcleo del simulador (en la clase `galatea.glider.Glider`).

Los tres métodos que se incluyen en la clase `Delta` son paradigmáticos (es decir, un buen ejemplo de lo que suelen contener esas clases).

El primero es el constructor. Noten la invocación al constructor de la superclase `super("Delta", 'A')`. Los parámetros corresponden a un nombre para la "familia" de nodos y, el segundo, a un caracter que identifica el tipo de nodo en la semántica Glider[3]. Este es, por tanto, un nodo autónomo lo que, por el momento, significa que es un nodo que no envia, ni recibe mensajes de otros nodos.

La segunda instrucción del constructor agrega el nuevo nodo a la lista de todos los nodos del sistema.

El método con el nombre `f` es simplemente una implementación auxiliar que podría corresponder a la especificación matemática de la función de transición del sistema en cuestión.

Finalmente en esta clase `Delta`, el método `fact()` contiene el código que será ejecutado cada vez que el nodo se active. Es decir, en el caso de los nodos autónomos, cada vez que se quiera reflejar un cambio en el componente que ese nodo representa.

[3]Decimos "familia" cuando estrictamente deberíamos decir tipo. Esto para evitar una confusión con el uso de la palabra tipo que sigue. Lo que está ocurriendo es que tenemos tipos de nodos en Glider, tipos de objetos, clases en Java y, por tanto, tipos de tipos de nodos: las clases Java que definen un tipo de nodo Glider para un modelo particular. Aquí se abre la posibilidad de compartir tipos de tipos de nodos entre varios modelos computacionales. El principio que ha inspirado la **modeloteca** y que se explicará más adelante

Algorithm 2 Delta.java

```
/*
 * Delta.java
 *
 * Created on April 22, 2004, 10:18 PM
 */

package demos.grano;

//import galatea.*;
import galatea.glider.*;

/**
 *
 * @author  Jacinto Dvila
 * @version beta 1.0
 */
public class Delta extends Node {

    /** Creates new CambioEdo */
    public Delta() {
        super("Delta", 'A');
        Glider.nodesl.add(this);
    }

    /** una funcion cualquiera para describir un fenomeno */
    double f(double x) {
        return (Math.cos(x/5) + Math.sin(x/7) + 2) * 50 / 4;
    }

    /** funcion de activacion del nodo */
    public boolean fact(){
        Modelo.variableDep = f((double)Modelo.variableInd);
        Modelo.variableInd++;
        it(1);
        return true; }

}
```

Este código de fact merece atención cuidadosa. La instrucción:

```
Modelo.variableDep = f((double)Modelo.variableInd);
```

invoca a la nuestra función f para asignar lo que produzca a una variable de un objeto que no hemos definido aún. Se trata de un objeto anónimo asociado a la clase `Modelo` que es aquella clase principal de nuestro modelo computacional. Las variables globales de nuestro modelos son declaradas como atributos de esta clase que, desde luego, puede llamarse como el modelista prefiera. En la siguiente sección ampliaremos los detalles sobre `Modelo`. Noten que el argumento de entrada de `f`, es otra variable atributo de la misma clase.

La siguiente instrucción incrementa el valor de aquella variable de entrada, en 1 (Ver el manual de Java para entender esta sintaxis extraña. x++ es equivalente a x = x + 1).

Para efectos de la simulación, sin embargo, la instrucción más importante es:

```
it(1);
```

Se trata de una invocación al método `it` de la clase `Node` (y por tanto de su subclase `Delta`) cuyo efecto es el de reprogramar la ejecución de este código (el de la `fact`) para que ocurra nuevamente dentro de `1` unidad de tiempo. En simulación decimos que el tiempo entre llegadas (*interarrival time*, **it**) de estos *eventos de activación* del nodo `Delta` es de 1. Veremos en capítulos posteriores que estos eventos y su manejo son la clave del funcionamiento del simulador.

2.2.2. El esqueleto de un modelo Galatea en Java

Como el lector sabrá deducir, necesitamos por el momento, por lo menos dos clases para nuestro modelo computacional: `Delta`, con la función de transición del único nodo y `Modelo`, con las variables y el programa principal. Vamos a agregar la clase `Analisis`, que explicamos a continuación, con lo cual nuestro modelo queda compuesto por tres clases:

`Delta` Como ya hemos explicado, representa a todo el sistema e implementa su única función de transición.

`Analisis` Es un nodo auxiliar para realizar análisis sobre el sistema simulado mientras se le simula. Se explica a continuación.

`Modelo` Es la clase principal del modelo. Contiene la variables globales y el método `main()` desde el que se inicia la simulación. También se muestra a continuación.

Esta es la clase `Analisis`:

Como habrán notado, su estructura es muy parecida a la de la clase `Demo`. No hay atributos de clase (aunque podrían incluirse (No hay ninguna restricción al respecto) y se cuentan 3 métodos. Aparecen el constructor y `fact()`, como en

Algorithm 3 Analisis.java

```
/*
 * Analisis.java
 *
 * Created on April 22, 2004, 10:22 PM
 */

package demos.grano;

//
import galatea.glider.*;

/**
 *
 * @author  Jacinto Davila
 * @version beta 1.0
 */
public class Analisis extends Node {

    /** Creates new Analisis */
    public Analisis() {
        super("Analisis",'A');
        Glider.nodesl.add(this);
    }

    public void paint() {
        for ( int x = 0; x < (int) Modelo.variableDep ; x++
            ) {
            System.out.print('.');
        }
        System.out.println('x');
    }

    /** funcion de activacion del nodo */
    public boolean fact(){
        paint();
        it(1);
        return true; }
}
```

`Demo`. La diferencia está en `paint()`. Pero este no es más que un mecanismo para visualizar, dibujando una gráfica en la pantalla del computador (con puntos y x's), el valor de una variable.

`Analisis` es también un nodo autónomo que quizás no corresponda a ningún componente del sistema simulador, pero es parte de los dispositivos para hacer seguimiento a la simulación. Por eso se le incluye en este ejemplo. El lector debe notar, como detalle crucial, que la ejecución de los códigos `fact()` de `Demo` y `Analisis` se programan concurrentemente. Concurrencia es otro concepto clave en simulación, como veremos luego.

2.2.3. El método principal del simulador

Estamos listos ahora para conocer el código de la clase `Modelo`:

Quizás el detalle más importante a destacar en esa clase es el uso del modificador Java `static`, en la declaración (la firma decimos en la comunidad OO) de los atributos y métodos de la clase. El efecto inmediato de la palabra static es, en el caso de los atributos, es que los convierte en **variables de clase**. Esto significa que "existen" (Se reserva espacio para ellos y se les puede direccionar, pues su valor permanece, es estático) existan o no instancias de esa clase. No hay que crear un objeto para usarlos.

Esto es lo que nos permite en el simulador usarlos como variables de globales de la simulación. Lo único que tiene que hacer el modelista es recordar el nombre de su clase principal y podrá acceder a sus variables. Esto hemos hecho con `Modelo.variableDep` y `Modelo.variableInd`.

El lector habrá notado que el `static` también aparece al frente del `main()`. Es obligatorio usarlo en el caso del `main()`, pero se puede usar con cualquier método para convertirlo en un método de clase. Como con las variables de clase, un método de clase puede ser invocado sin crear un objeto de esa clase. Simplemente se le invoca usando el nombre de la clase. En Galatea usamos mucho ese recurso. De hecho, todas las instrucciones en el código de `main()` que comienzan con `Glider.` son invocaciones a métodos de clase (declarados con `static` en la clase `galatea.glider.Glider`). Veamos cada uno de esos:

Glider.setTitle("Un modelo computacional muy simple"); Le asigna un título al modelo que será incluido en los reportes de salida.

Glider.setTsim(50); Fija el tiempo de simulación, en este caso en 50.

Glider.setOutf(5) Fija el número de decimales en las salidas numéricas.

Glider.trace("Modelo.trc"); Designa el archivo `Modelo.trc` para que se coloque en él la traza que se genera mientras el simulador corre. Es una fuente muy importante de información a la que dedicaremos más espacio posteriormente. El nombre del archivo es, desde luego, elección del modelista. El archivo será colocado en el directorio desde donde se ejecute el simulador[4]

[4]Cuidado con esto. Si usa un ambiente de desarrollo como el NetBeans, este colocará esa salida en uno de sus directorios.

Algorithm 4 Modelo.java

```
1  /*
2   * Modelo.java
3   *
4   * Created on April 22, 2004, 10:13 PM
5   */
6  package demos.grano;
7  import galatea.glider.*;
8  /**
9   * Este es un primer ejemplo del como implementar un modelo
        computacional
10  * para simulacion en Galatea.
11  *
12  * @author  Jacinto Davila
13  * @version beta 1.0
14  */
15 public class Modelo {
16      /** Variables globales a todos el sistema se declaran
            aqui */
17      public static double variableDep = 0d ; // esta es una
            variable dependiente.
18      public static int variableInd = 0 ; // variable
            independiente.
19      /** No hace falta, pues usaremos una sola instancia "
            estatica"*/
20      public Modelo() {
21      }
22      /**
23       * Este es el programa principal o locus de control de
             la simulacion
24       * @param args the command line arguments
25       */
26      public static void main(String args[]) {
27          /** La red de nodos se inicializa aqui */
28          Delta delta = new Delta();
29          Analisis analisis = new Analisis();
30          Glider.setTitle("Un␣modelo␣computacional␣muy␣simple"
                );
31          Glider.setTsim(50);
32          //Glider.setOutf(5);
33          // Traza de la simulacion en archivo
34         // Glider.trace("Modelo.trc");
35          // programa el primer evento
36          Glider.act(delta,0);
37          Glider.act(analisis,1);
38          // Procesamiento de la red
39          Glider.process();
40          // Estadisticas de los nodos en archivo
41          //Glider.stat("Modelo.sta");
42          Glider.stat();
43      }
44 }
```

Glider.act(delta,0); Programa la primera activación del nodo `delta` (el objeto que representa el nodo `delta` del sistema simulado. Aclaramos esto un poco más adelante) para que ocurra en el tiempo `0` de simulación. Las activaciones posteriores, como vimos, son programadas desde el propio nodo `delta`, con la invocación `it(1)`.

Glider.act(analisis,1); Programa la primera activación del nodo `analisis`. Luego del tiempo `1`, y puesto que `analisis` también invoca a `it(1)`, ambos nodos se activarán en cada instante de simulación. Esta es la primera aproximación a la concurrencia en simulación.

Glider.process(); Comienza la simulación.

Glider.stat("Modelo.sta"); Designa al archivo

Solamente nos resta comentar este fragmento de código:

```
1 Delta delta = new Delta();
2 Analisis analisis = new Analisis();
```

En estas líneas se crean los objetos que representan, en la simulación, los componentes del sistema simulado. Ya explicamos que el único componente "real" es `delta`, pero `analisis` también debe ser creado aquí aunque solo se trate de un objeto de visualización.

Note el lector que podríamos usar clases con elementos `static` para crear esos componentes. Pero quizás lo más trascendental de poder modelar al Nivel Java es precisamente la posibilidad de crear y destruir componentes del sistema simulado durante la simulación, usando el código del mismo modelo. Esto no es posible en el viejo Glider. Será siempre posible en el Nivel Glider de Galatea, pero sólo porque mantendremos el acceso directo al entorno OO de bajo nivel (Java seguirá siendo el lenguaje matriz).

Esa posibilidad de crear y destruir nodos es importante porque puede ser usada para reflejar el **cambio estructural** que suele ocurrir en los sistemas reales. Este concepto se ha convertido en todo un proyecto de investigación para la comunidad de simulación, pues resulta difícil de acomodar en las plataformas tradicionales.

2.2.4. Cómo simular con Galatea, primera aproximación

Hemos concluído la presentación del código de nuestro primero modelo de simulación Galatea. Para simular debemos 1) compilar los códigos Java a `.class` (con el `javac`, como se explica en el apéndice A) y ejecutar con la máquina virtual (por ejemplo con `java`, pero también podría ser con `appletviewer` o con un Navegador y una página web, si convertimos el modelo en un applet).

La figura 2.1 muestra la salida que produce nuestro modelo en un terminal de texto.

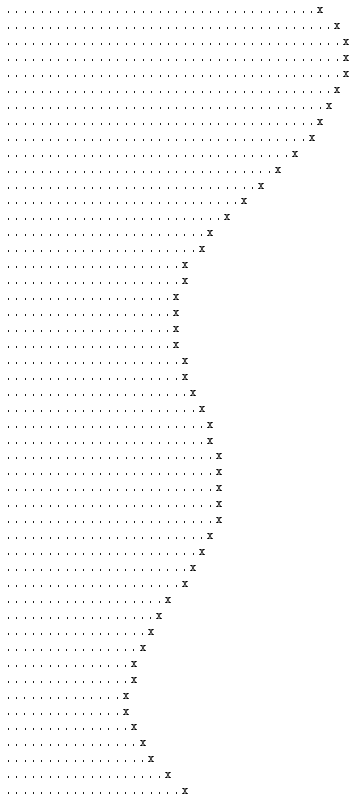


Figura 2.1: Salida del modelo en un terminal

No es un gráfico muy sofisticado, pero muestra el cambio de estado del sistema a lo largo del tiempo, con un mínimo de esfuerzo. Mucha más información útil se encontrará en los archivos de traza (`Modelos.trc`) y de estadísticas (`Modelo.sta`).

Dejamos hasta aquí, sin haber conocido demasiado, la simulación tradicional. En la siguiente parte del capítulo conoceremos un modelo de simulación multi-agente.

2.3. El primer modelo computacional multi-agente

En esta parte del tutorial usamos elementos fundamentales de lo que se conoce como la tecnología de agentes inteligentes. En beneficio de una primera lección sucinta, no entraremos en detalles conceptuales. Sin embargo, un poco de contexto es esencial.

modelado basado en agentes

La ingeniería de sistemas basados en agentes (*Agent-Based Modelling*, ABM, como se le suele llamar) es un desarrollo reciente en las ciencias computacionales que promete una revolución similar a la causada por la orientación por objetos. Los agentes a los que se refiere son productos de la Inteligencia Artificial, IA, [79], una disciplina tecnológica que ha sufrido un cambio de enfoque en los últimos años: los investigadores de la antigua (pero buena) Inteligencia Artificial comenzaron a reconocer que sus esfuerzos por modelar un dispositivo inteligente carecían de un compromiso sistemático (como ocurre siempre en ingeniería) con la posibilidad de que ese dispositivo "hiciera algo". Ese "hace algo inteligentemente" es la característica esencial de los agentes (inteligentes) tras la que se lanza el proyecto de IA aproximadamente desde la década de los noventa del siglo pasado.

El poder modelar un dispositivo que puede hacer algo (inteligente) y el enfoque modular que supone concebir un sistema como constituido por agentes (algo similar a concebirlo constituido por objetos) hacen de esta tecnología una herramienta muy efectiva para atacar problemas complejos [48], como la gestión del conocimiento y, en particular, el modelado y simulación de sistemas [13, 75].

La noción de agente tiene, no obstante, profundas raíces en otras disciplinas. Filósofos, psicólogos y economistas ha propuesto modelos y teorías para explicar qué es un agente, pretendiendo explicar al mismo tiempo a todos los seres humanos. Muchas de esas explicaciones refieren como es que funcionamos en una suerte de ciclo en el que observamos nuestro entorno, razonamos al respecto y actuamos, procurando atender a lo que hemos percibido, pero también a nuestras propias creencias e intenciones. Ese ciclo en cada individuo, lo convierte en causante de cambios en el entorno o ambiente compartido normalmente con otros agentes, pero que puede tener también su propia disciplina de cambio. Algunos de los cambios del ambiente son el efecto sinergético de la acción combinada de dos o más agentes.

Esas explicaciones generales han inspirado descripciones lógicas de lo que es un agente y una sociedad de agentes. En el resto del libro revisamos algunas de ellos, las mismas que usamos como especificaciones para el diseño de Galatea.

No hay, sin embargo, ningún compromiso de exclusividad con ninguna de ella y es muy posible que Galatea siga creciendo con la inclusión de nuevos elementos de la tecnología de agentes.

2.3.1. Un problema de sistemas multi-agentes

biocomplejidad

Biocomplejidad es el término seleccionado por la *National Science Foundation* de los EEUU para referencirse a "el fenómeno resultante de la interacción entre los componentes biológicos, físicos y sociales de los diversos sistemas ambientales de la tierra". Modelar sistemas con esas características es una tarea difícil que puede simplificarse con el marco conceptual y las plataformas de sistemas multi-agentes.

El ejemplo que mostramos en esta sección es una versión simplificada de un "modelo juguete" que ha sido construido como parte del proyecto "*Biocomplexity: Integrating Models of Natural and Human Dynamics in Forest Lanscapes Across Scales and Cultures*", NFS GRANT CNH BCS-0216722. Galatea está siendo usada para construir modelos cada vez más complejos de la dinámica apreciable en la Reserva Forestal de Caparo[5], ubicada en los llanos de Barinas, al Sur-Occidente de Venezuela.

Hemos simplificado el más elemental de esos modelos juguetes[6] y lo hemos convertido en un **juego de simulación**, en el que el **simulista** debe tomar el lugar del ambiente. Los agentes están implementados en una versión especial del paquete `galatea.glorias` [7].

El resultado es un modelo en el que el simulista es invitado a llevar un registro manual de la evolución de una Reserva Forestal (inventando sus propias reglas, con algunas sugerencias claro), mientras sobre ese ambiente actúa un conjunto de agentes artificiales que representan a los actores humanos de la Reserva.

Nuestra intención con el juego es familiarizar a los lectores con el desarrollo de un modelo multi-agente, sin que tengan que sufrir sino una pequeña indicación de la complejidad intrínseca de estos sistemas.

2.3.2. Conceptos básicos de modelado multi-agente

Siguiendo la práctica habitual en el modelado de sistemas, para crear el *modelo socio-económico-biológico* de la Reserva Forestal de CAPARO, construimos las descripciones a partir de un conjunto de conceptos básicos. Esos conceptos son: estado del sistema, dinámica, proceso y restricciones contextuales.

simulación multiagente

El estado del sistema es la lista de variables o atributos que describen al sistema en un instante de tiempo dado. Es exactamente equivalente a lo que en simulación se suelen llamar variables de estado. Pero también puede ser visto como una lista de atributos del sistema que cambian a lo largo del tiempo y que no necesariamente corresponde a magnitudes medibles en números

[5] Detalles en http://cesimo.ing.ula.ve/INVESTIGACION/PROYECTOS/BIOCOMPLEXITY/
[6] Programado originalmente por Niandry Moreno y Raquel Quintero.
[7] adaptado al caso por Niandry Moreno, Raquel Quintero y Jacinto Dávila.

reales. Los juicios de valor (cierto o falso) también se pueden incluir entre los atributos. En virtud de su naturaleza cambiante, nos referiremos a esos atributos como **fluentes** o **propiedades** del sistema que cambian al transcurrir el tiempo. Ya sabemos que, en Galatea, corresponderán a atributos de algún objeto.

Una dinámica es una disciplina, normalmente modelada como una función matemática del tiempo, describiendo como cambia una o varias de esas variables de estados (atributos o fluentes). Noten, sin embargo, que esas funciones del tiempo no necesariamente tienen que ser sujetas a otros tratamientos matemáticos. Para describir agentes, por ejemplo, puede ser preciso recurrir a funciones discretas (no derivables) parciales sobre el tiempo.

Un proceso es la conflagración de una o más de esas dinámicas en un arreglo complejo donde se afectan unas a otras. Es decir, el progreso de una dinámica al pasar el tiempo está restringido por el progreso de otra dinámica, en cuanto a que sus trayectorias (en el sentido de [92] están limitadas a ciertos conjuntos particulares (de trayectorias posibles).

Una restricción contextual es una limitación sobre los valores de las variables de estado o sobre las dinámicas que pueden estar involucradas en los procesos. Es decir, es una especificación de cuáles valores o dinámicas NO pueden ser parte del sistema modelado.

Le pedimos al lector que relea estas definiciones. Las usaremos a continuación en la medida en que conocemos el modelo juguete el cual está compuesto por:

La clase `Colono` que define a los agentes de tipo colono que ocupan una Reserva. Varias instancias de esta clase hacen de este un modelo multi-agente.

La clase `Delta` que implementa la función de transición del ambiente natural. En este caso, es solo un mecanismo de consulta e interacción con el simulista, quien estará "simulando", por su cuenta, la evolución del ambiente natural.

La clase `Modelo` que, como antes, contiene el programa principal y las variables globales del simulador.

La clase `Interfaz` que es uno de los elementos más importantes del simulador multi-agente: el conjunto de servicios que median entre los agentes y el ambiente cuando aquellos actúan y observan sobre este.

Con este modelo computacional queremos caracterizar el proceso en la Reserva Forestal que está siendo ocupada por Colonos con distintos planes de ocupación. La pregunta principal del proyecto es si existe una combinación de planes de ocupación que pudiese considerarse como intervención sustentable sobre la reserva, dado que no compromete la supervivencia de su biodiversidad.

En nuestro ejemplo, estaremos muy lejos de poder abordar esa pregunta, pero confiamos que el lector encuentre el juego interesante e iluminador en esa dirección.

2.3.3. El modelo del ambiente

Llamamos `Delta` a la clase que implementa el modelo del ambiente. En este modelo juego, `Delta` es solo una colección de rutinas para que el simulador interactúe con el simulista y sea este quien registre el estado del sistema.

El propósito de esto es meramente instruccional. El simulista tendrá la oportunidad de identificar los detalles operativos de la simulación y no tendrá problemas de automatizar ese registro en otra ocasión.

Sin embargo, para no dejar al simulista desválido frente a la complejidad de un ambiente natural, incluimos las siguientes recomendaciones y explicaciones:

> Una reserva forestal es un ambiente natural sumamente complejo. Cualquier descripción dejará fuera elementos importantes. Con eso en mente, nos atrevemos a decir que son 3 los procesos macro en la reserva forestal: 1) clima, 2) hidrología y relieve, con la dinámica de suelos y 3) vegetación que siempre será un resumen de la extraordinaria colección de dinámicas de crecimiento, reproducción y muerte de la capa vegetal.
>
> Noten que estamos dejando de lado, expresamente, el proceso de la población animal, salvo, claro, por aquello que modelaremos acerca de los humanos, usando agentes. Sin embargo, el efecto del proceso animal debe ser tomando en cuenta en el proceso vegetación, particularmente el consumo de vegetación por parte del ganado que puede ser devastador para la reserva.
>
> Para ayudar a nuestro simulista a llevar un registro de esos procesos y sus dinámicas, le sugerimos llevar anotaciones en papel del estado global de la reserva a lo largo del tiempo. Quizás sobre un mapa de la reserva, con la escala adecuada (al tiempo disponible para jugar), se pueda llevar ese registro mejor usando un código de símbolos como el que mostramos en la figura 2.2[8]
>
> Los símbolos de clima son suficientemente claros. Los de hidrología y relieve son un poco más difíciles de interpretar, pero como cambian con poca frecuencia no debería haber mucho problema. La dinámica de cambio más compleja será la de la vegetación.
>
> Lo que mostramos en la lámina es una idea de los tipos de cobertura vegetal. El más apreciado en una reserva de biodiversidad forestal es el que llamamos bosque primigenio, no tocado por la acción humana. Una vez que ha sido intervenido se convierte en bosque intervenido (y nunca más será primigenio). Pero los espacios naturales

[8] El simulista bien podría fotocopiar esa imagen y recortar los símbolos para marcar el mapa, como solían hacer los cartógrafos (ahora lo hacen las máquinas, se admite, pero no es igual de divertido).

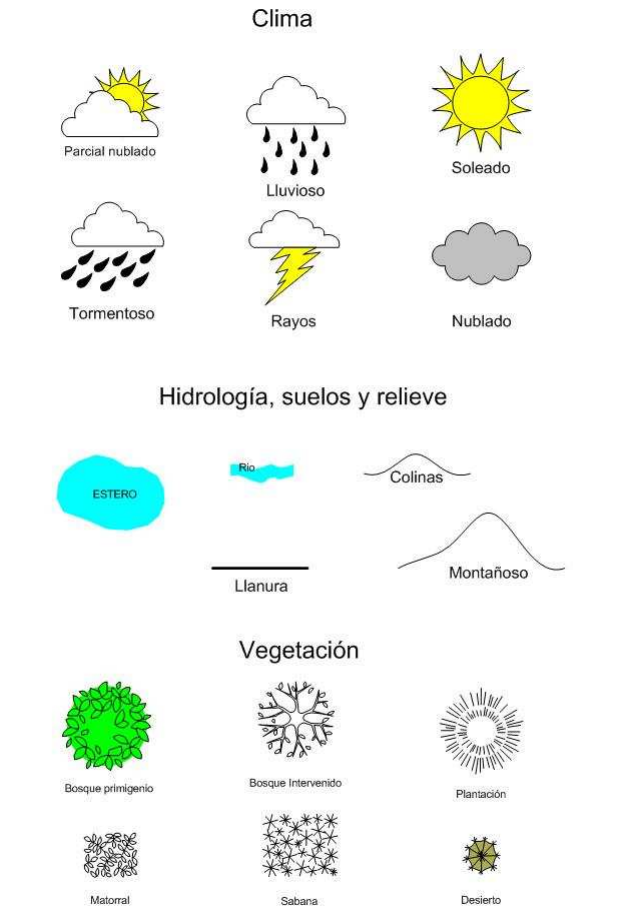


Figura 2.2: Símbolos ambientales

también pueden ser usados para plantaciones de especies explotables por su madera (como las coníferas que se ilustran con la figura). Los matorrales son, normalmente, bosques que han sido intervenidos y luego de ser abandonados (por los humanos y sus animales), han prosperado por su cuenta. El otro uso del espacio que es crucial, sobretodo para la ganadería, es la sabana. Finalmente, cuando el espacio es devastado sin posibilidad de recuperación se crea un desierto irrecuperable. Nuestro simulista querrá aprovechar estas indicaciones para modelar sus propias reglas de **cambio de uso de la tierra**.

El único detalle adicional a tener presente es la escala de tiempo en el que ocurren los cambios (por lo menos para la observación superficial). En el caso del clima, todos sabemos, es de minutos, horas, días y meses. En el caso de vegetación es de meses y años. En el caso de hidrología y suelos, va desde años (con las crecidas e inundaciones estacionales) hasta milenios.

Ahora podemos discutir el código de `Delta`:

Comentaremos sobre tres fragmentos de ese código en detalle:

```
1 for (int i=0;i<NUMCOLONOS;i++) {
2     agente[i]=new Colono();
3     agente[i].agentId= i+1;
4     GInterface.agentList.add(agente[i]); }
```

Esta pieza de código crea los agentes de esta simulación. Suele colocarse en el `main()` de `Modelo`, pero puede aparecer, como en este caso, en otro lugar, siempre que se invoque al comienzo de la simulación (en este caso, al crear el ambiente).

Note la invocación a `GInterface.agentList.add(agente[i])` con la que se registra a cada agente en una base de datos central para el simulador, gestionada por la clase GInterface del paquete `galatea.hla`. Así es como el simulador conoce a todos los agentes de la simulación.

El siguiente fragmento es puro código Java. Es la manera de leer desde el teclado y poder identificar las respuestas del usuario. El método `si`, incluido en el código simplemente lee el teclado y si el texto leído comienzo con el caracter '`S`', devuelve **true**. Si la respuesta no es afirmativa, el simulador termina su ejecución con la invocación a `System.exit(0)`.

```
1 InputStreamReader ir = new InputStreamReader(System.in);
2 BufferedReader in = new BufferedReader(ir);
3 if (!si(in))  System.out.println("Ambiente:␣Fin␣del␣juego!")
      ; System.exit(0);
```

El último fragmento es sumamente importante para el simulador multiagente:

```
1 for (int l=0;l<NUMCOLONOS;l++) {
2     // Actualiza el reloj de cada agente
```

Algorithm 5 contrib.biocaparotoy.Delta.java primera parte

```
1  package contrib.biocaparotoy;
2
3  import galatea.glider.*;
4  import galatea.hla.*;
5  import galatea.glorias.*;
6
7  public class Delta extends Node {
8      /**this variable indicates the quantity of settler agent at the forest
           reserve.*/
9      public int NUM_COLONOS=3;
10     /**This is an array of references to settler agents.*/
11     public Colono[] agente= new Colono[NUM_COLONOS];
12     /** Creates new Delta */
13     public Delta() {
14         super("Delta", 'A');
15         Glider.nodesl.add(this);
16         /* Las instrucciones han sido editadas, ver fuente original..
17         "Este es un juego de simulacion multiagente de una Reserva Forestal
18        "----------------------------------------");
19        "Le aconsejamos que tome nota de todos los detalles");
20        "");
21        "Comienza el juego!");
22        */
23         //Se crean las instancias de cada uno de los agentes colono
24         //Esto deberia ir junto con la declaracion de la red en el programa
25         //principal del modelo
26         for (int i=0;i<NUM_COLONOS;i++){
27             agente[i]=new Colono();
28             agente[i].agentId= i+1;
29             GInterface.agentList.add(agente[i]);
30         }
31     }
32     /** funcion de activacion del nodo */
33     public boolean fact(){
34         it(1);
35
36         try{
37             /*
38            "---------------------------- Es el tiempo "+Glider.getTime());
39            "Ambiente: Responda a cada una de estas preguntas con cuidado:");
40            "Ambiente: 1.- Donde esta cada uno de los agentes?");
41            "Ambiente: 2.- Como es el clima en cada lugar?");
42            "Ambiente: 3.- Como se estan comportando los torrentes
43                                        y el nivel freatico");
44            "Ambiente: 4.- Que cambios tienen lugar en la vegetacion");
45            "Ambiente: 5.- Que puede observar cada uno de los agentes");
46            "Ambiente: Continuar?(S/N)");
47             InputStreamReader ir = new InputStreamReader(System.in);
48             BufferedReader in = new BufferedReader(ir);
49             if (!si(in)) {
50                 System.out.println("Ambiente: Fin del juego!");
51                 System.exit(0);
52             }
53         } catch( Exception e) { };
54         for (int l=0;l<NUM_COLONOS;l++){
55             // Actualiza el reloj de cada agente
56             agente[l].clock=Glider.getTime();
57             // les transmite lo que deben ver
58             actualizarsensores(agente[l]);
59             // activa el razonado de cada agente
60             agente[l].cycle();
61         }
62         System.out.println("Simulador: Comienzo a procesar las influencias");
63         GInterface.gatherInfluences_test();
64         GInterface.process_test();
65         return  true ;
66     }
67     // ver figura siguiente
68    ..
```

Algorithm 6 contrib.biocaparotoy.Delta.java segunda parte

```
1     //.. ver figura anterior
2     public void actualizarsensores(Colono agente){
3          try {
4               InputStreamReader ir = new InputStreamReader(System.in)
                    ;
5               BufferedReader in = new BufferedReader(ir);
6               System.out.println("Ambiente: Hablemos del agente: "+
                    agente);
7               // La entrevista
8               System.out.println("Ambiente: Se establecio ya? (S/N)")
                    ;
9               if (!si(in)) {
10                   agente.inputs.add("No establecido");
11              }
12              // La entrevista 2
13              System.out.println("Ambiente: Ve una celda desocupada?
                    (S/N)");
14              if (si(in)) {
15                   agente.inputs.add("Celda desocupada");
16              }
17              // La entrevista 3
18              System.out.println("Ambiente: Ve una celda apta para
                    establecerse? (S/N)");
19              if (si(in)) {
20                   agente.inputs.add("Celda apta para establecerse");
21              }
22              // La entrevista 4
23              System.out.println("Ambiente: Ve una celda apta para
                    sembrar? (S/N)");
24              if (si(in)) {
25                   agente.inputs.add("Celda apta para sembrar");
26              }
27              // La entrevista 5
28              System.out.println("..: Se agotaron los recursos en su
                    parcela? (S/N)");
29              if (si(in)) {
30                   agente.inputs.add("Se agotaron los recursos");
31              }
32         } catch (Exception e) { };
33         System.out.println("Ambiente: Este agente observara "+
                agente.inputs);
34    }
35    public boolean si(BufferedReader in) {
36         int c, i;
37         int resp [] = new int[10];
38         i = 0;
39         try {
40              while ( (c=in.read()) != -1 && c!=10) { resp[i]=c ;}
41         } catch (Exception e) {};
42         return (char) resp[0]=='S' ;
43    }
44 }
```

```
3     agente[l].clock=Glider.getTime();
4     // les transmite lo que deben ver
5     actualizarsensores(agente[l]);
6     // activa el razonado de cada agente
7     agente[l].cycle(); }
```

Allí se le dice a cada agente que hora es, que cosas está observando a esta hora y se le pide que piense y decida que hacer (`cycle()`).

Los detalles del cómo funciona todo esto aparecen en el resto del libro. Por lo pronto, le pedimos al lector que recuerde que este es un modelo de tiempo discreto. Cada vez que el ambiente cambia, el mismo ambiente indica los cambios a los agentes y espera su respuesta.

Implementaciones en las que el ambiente y los ambientes corren "simultáneamente" también son posibles con Galatea. Pero requieren más esfuerzo de implementación, como mostraremos más adelante.

2.3.4. El modelo de cada agente

En Galatea, un agente es un objeto con una sofisticada estructura interna. Considere el código en el listado 7.

De nuevo concentraremos la atención, esta vez en 2 piezas del código:

El método `init()` en, código 7,es una muestra, un tanto críptica, de lo que tenemos que hacer cuando queremos fijar las metas del agente desde el principio. Un agente tiene metas, es decir, las reglas que determinan lo que querrá hacer.

```
1  public void init() outputs = new LOutputs();
2  //finding a place(0)
3  addPermanentGoal("buscarsitio",0,new String[]"No␣establecido
       ");
4  //settling down(1)
5  addPermanentGoal("establecerse",1,new String[]"Celda␣
       desocupada", "Celda␣apta␣para␣establecerse");
6  //cleaning the land and seeding agriculture of subsistence
       (2)
7  addPermanentGoal("limpiarsembrar",2,new String[]"Celda␣
       desocupada", "Celda␣apta␣para␣sembrar");
8  //deforesting and selling wood to illegal traders(3)
9  addPermanentGoal("talarvender",3,new String[]"Celda␣
       desocupada", "Celda␣con␣madera␣comercial");
10 //moving(4) addPermanentGoal("mudarse",4,new String[]"Se
       agotaron los recursos");
11 //expanding settler's farms(5)
12 addPermanentGoal("expandirse",5,new String[]"Celda␣
       desocupada", "Celda␣apta␣para␣expandir");
```

En este método, luego de crear la lista que guardará las salidas que este agente envía a su ambiente, se le asignan (al agente) sus metas permanentes (es decir, sus metas durante la simulación). Mostraremos, más adelante, una forma alternativa de programar las reglas a los agentes en Galatea.

Algorithm 7 contrib.biocaparotoy.Colono.java

```
1  package contrib.biocaparotoy;
2  import galatea.glorias.*;
3  import galatea.hla.*;
4  public class Colono extends Ag{
5      //Atributos que definen el estado interno del agente.
6      int x=-1;
7      int y=-1;
8      int recursos_economicos;
9      int num_celdas=0;//numero de celdas que posee
10     int MAX_NUM_CELDAS=9;
11     int[][] propiedad=new int[MAX_NUM_CELDAS][2];
12     int tr; //tiempo en la region invadida.
13     public Colono() {
14         super(6,"colono");
15         this.population++;
16         this.agentId=population;
17         init(); }
18     public void init(){
19         outputs = new LOutputs();
20         //finding a place(0)
21         addPermanentGoal("buscarsitio",0,new String[]{"No␣
               establecido"});
22         //settling down(1)
23         addPermanentGoal("establecerse",1,
24                 new String[]{"Celda␣desocupada","Celda␣apta␣para␣
                       establecerse"});
25         //cleaning the land and seeding agriculture of subsistence
               (2)
26         addPermanentGoal("limpiarsembrar",2,
27                 new String[]{"Celda␣desocupada","Celda␣apta␣para␣
                       sembrar"});
28         //deforesting and selling wood to illegal traders(3)
29         addPermanentGoal("talarvender",3,
30                 new String[]{"Celda␣desocupada","Celda␣con␣madera␣
                       comercial"});
31         //moving(4)
32         addPermanentGoal("mudarse",4,
33                 new String[]{"Se␣agotaron␣los␣recursos"});
34         //expanding settler's farms(5)
35         addPermanentGoal("expandirse",5,
36                 new String[]{"Celda␣desocupada","Celda␣apta␣para␣
                       expandir"}); }
37     ...
38 }
```

El formato de invocación de `addPermanentGoal` es, básicamente, una regla de **si** condiciones **entonces** acción. Por ejemplo, para decirle al agente que **si** *no se ha establecido*, **entonces** debe *buscar un sitio* para hacerlo, escribimos:

```
addPermanentGoal("buscarsitio",0, new String[]"No establecido");
```

La cadena "No establecido" será una entrada desde el ambiente para el agente (como se puede verificar en la sección anterior). "buscarsitio" es el nombre de una acción que el agente intentará realizar en el ambiente. Cada acción corresponde con un método o procedimiento de cambio del ambiente que se define en la llamada Interfaz que se explica la sección siguiente. La conceptualización correspondiente se encuentra en la teoría de simulación multiagente [13, 14, 17] y en la implementación que llamamos Galatea [15, 39, 45, 60, 76, 87, 88, 3]. Por lo pronto, estos agentes tienen limitadas sus reglas a lo que que se conoce como lógica proposicional, sin variables. El motor de inferencia correspondiente (para lógica proposicional, repetimos) es parte de la implementación en la clase `galatea.glorias.Ag`. Una implementación más expresiva existe también como parte de la clase `galatea.glorias.PrologAg` y la explicaremos más adelante.

2.3.5. La interfaz Galatea: primera aproximación

La clase Interfaz es sumamente difícil de explicar en este punto, sin el soporte conceptual de las influencias. Baste decir que se trata de los métodos que ejecuta el ambiente como respuesta a las salidas de los agentes. Los agentes le dicen al ambiente que quieren hacer, con sus salidas, y el ambiente responde ejecutando estos métodos.

Es por ello que, en el código que sigue, el lector podrá ver métodos con los mismos nombres que usamos para crear los objetos `Output` en el paquete `galatea.glorias.Ag`. Los que se muestran a continuación son cargados (en tiempo de simulación) y ejecutados por el simulador desde el objeto `GInterface` del paquete `galatea.hla`.

Los métodos en este ejemplos son sólo muestras. No hacen salvo indicar al simulista que la acción se ha realizado (con lo que el simulista deberá anotar algún cambio en su registro del ambiente). Pero, desde luego, el modelista puede colocar aquí, cualquier código Java que implemente los cambios automáticos apropiados para el sistema que pretende simular.

Noten, por favor, la lista de argumentos de estos métodos. Siempre aparecen un `double` y un `Agent` como primeros argumentos. Esta es una convención Galatea y la usamos para transferir el tiempo actual (en el `double`) y una referencia al objeto agente que ejecuta la acción (en `Agent`). Los métodos pueden tener argumentos adicionales, pero para ello es preciso usar la clase alternativa `PrologAg` para implementar los agentes, como se ilustra más adelante en este libro. Toda esta explicación, desde luego, tendría que ser suplementada con ejercicios de prueba. Pueden usar el código que se anexa y consultar con sus autores.

Algorithm 8 contrib.biocaparotoy.Interfaz.java

```
1  package contrib.biocaparotoy;
2
3  import galatea.glider.*;
4  import galatea.hla.*;
5  import galatea.glorias.*;
6
7  public class Interfaz {
8      public void buscarsitio(double t, Agent a) {
9          System.out.println("Interface:␣El␣agent␣"
10                 +a.agentId+"␣busco␣un␣sitio␣y␣lo␣encontro.")
                       ;
11     }
12     public void establecerse(double t, Agent a){
13         System.out.println("Interface:␣El␣agent␣"
14                 +a.agentId+"␣se␣ha␣establecido.");
15     }
16     public void expandirse(double t, Agent a) {
17         System.out.println("Interface:␣El␣agent␣"
18                 +a.agentId+"␣expandio␣su␣parcela.");
19     }
20     public void limpiarsembrar(double t, Agent a) {
21         System.out.println("Interface:␣El␣agent␣"
22                 +a.agentId+"␣limpio␣el␣terreno␣y␣sembro.");
23     }
24     public void mudarse(double t, Agent a) {
25         System.out.println("Interface:␣El␣agent␣"
26                 +a.agentId+"␣ha␣abandonado␣su␣terreno.");
27     }
28     public void talarvender(double t, Agent a) {
29         System.out.println("Interface:␣El␣agent␣"
30                 +a.agentId+"␣talo␣y␣vendio␣la␣madera.");
31     }
32     public void haceAlgo(double m, Agent a) {
33         System.out.println("Interface:␣El␣agent␣"+a.agentId+
               "␣hizo␣algo.");
34     }
35 }
```

2.3.6. El programa principal del simulador multi-agente

El código de la clase principal Modelo tiene pocos cambios respecto al ejemplo anterior, pero hay uno muy importante:

```
1  package demos.biocaparo.toy;
2    import galatea.*; import galatea.glider.*;
3    import galatea.hla.*;
4    import galatea.gloriosa.*;
5    /** *@authors Niandry Moreno, Jacinto Dávila *@version
         Juguete #1. */
6    public class Modelo {
7      public static Delta ambiente = new Delta();
8      /**Simulator. Main process. */
9      public static void main(String args[])  {
10        GInterface.init("demos.biocaparo.toy_1.Interfaz");
11        Glider.setTsim(30);
12        System.out.println("Inicio de simulacion");
13        // Traza de la simulación en archivo
14        Glider.trace("DemoCaparo.trc");
15        Glider.act(ambiente,0);
16        //Procesa la lista de eventos futuros.
17        Glider.process();
18        // Estadisticas de los nodos en archivo
19        Glider.stat("DemoCaparo.sta");
20        System.out.println("La simulacion termino");  } }
```

El cambio importante es la identificación de la clase donde se almacenan los métodos de interfaz:

```
GInterface.init("contrib.biocaparotoy.Interfaz");
```

Con este cambio, el sistema está listo para simular.

2.3.7. Cómo simular con Galatea. Un juego multi-agente

Como antes, simplemente hace falta compilar todos los .java y, luego, invocar al simulador:

```
java contrib.biocaparotoy.Modelo
```

Disfrute su juego!.

2.4. Modelando Procesos Humanos con Galatea

La descripción explícita de los agentes puede ser una herramienta muy útil para abordar la complejidad al nivel de detalle apropiado en ciertos sistemas. Sin embargo, no es la única forma de dar cuenta de conducta humana sistemática. La simulación tradicional DEVS[91, 92] permite, sin más, modelar un proceso consistente de una serie parcialmente ordenada y posiblemente recurrente de actividades caracterizadas como servicios. Un servicio es, en términos simples, una

proceso

servicio

Figura 2.3: Servicio en Galatea

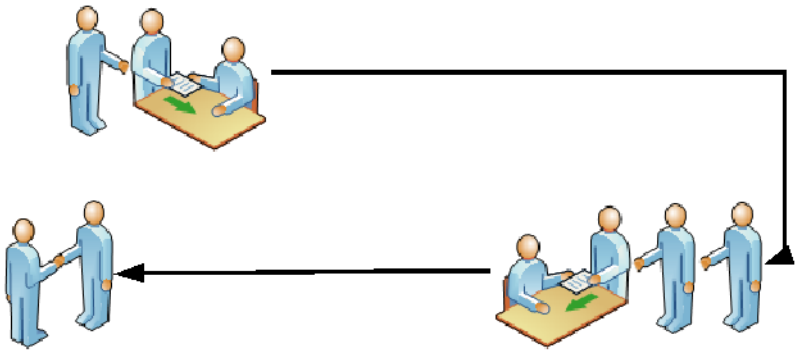


Figura 2.4: Proceso

cola de solicitantes que son servidos en lotes definibles durante ciertos tiempos determinables.

Esa conceptualización está imbuida en la semántica de Galatea, de manera que un servicio puede ser fácilmente representado en un modelo de simulación como se indica en la figura 2.3 y un proceso, de la misma manera, como un agregado de servicios, como en figura 2.4.

Permítannos ilustrar las posibilidad con un ejemplo de un sistema real. A continuación, la figura 2.5 describe un proceso institucional que se sigue en cierta institución pública Venezolana, para otorgar subvenciones (ayudas económicas) a investigadores y proyectos de investigación.

El paquete `contrib.fundacite`, en el repositorio Galatea, contiene la codificación de un modelo elemental, pero completamente funcional (simulable), de ese proceso real. Con ayuda del código en `contrib.fundacite.gGui`, es posible visualizar el comportamiento de ese sistema siguiendo el proceso en cada servicio de varias maneras. Podemos, por ejemplo, reportar el nivel de ocupación de cada servicio, con las estadísticas de sus colas (ver figura 2.6). Pero podemos también conocer la "inclinación hacia la estabilidad" de cada servicio, graficando el comportamiento de sus colas a lo largo del tiempo (en figura 2.7).

análisis de estabilidad

Esto que llamamos "inclinación hacia la estabilidad" es sumamente útil para

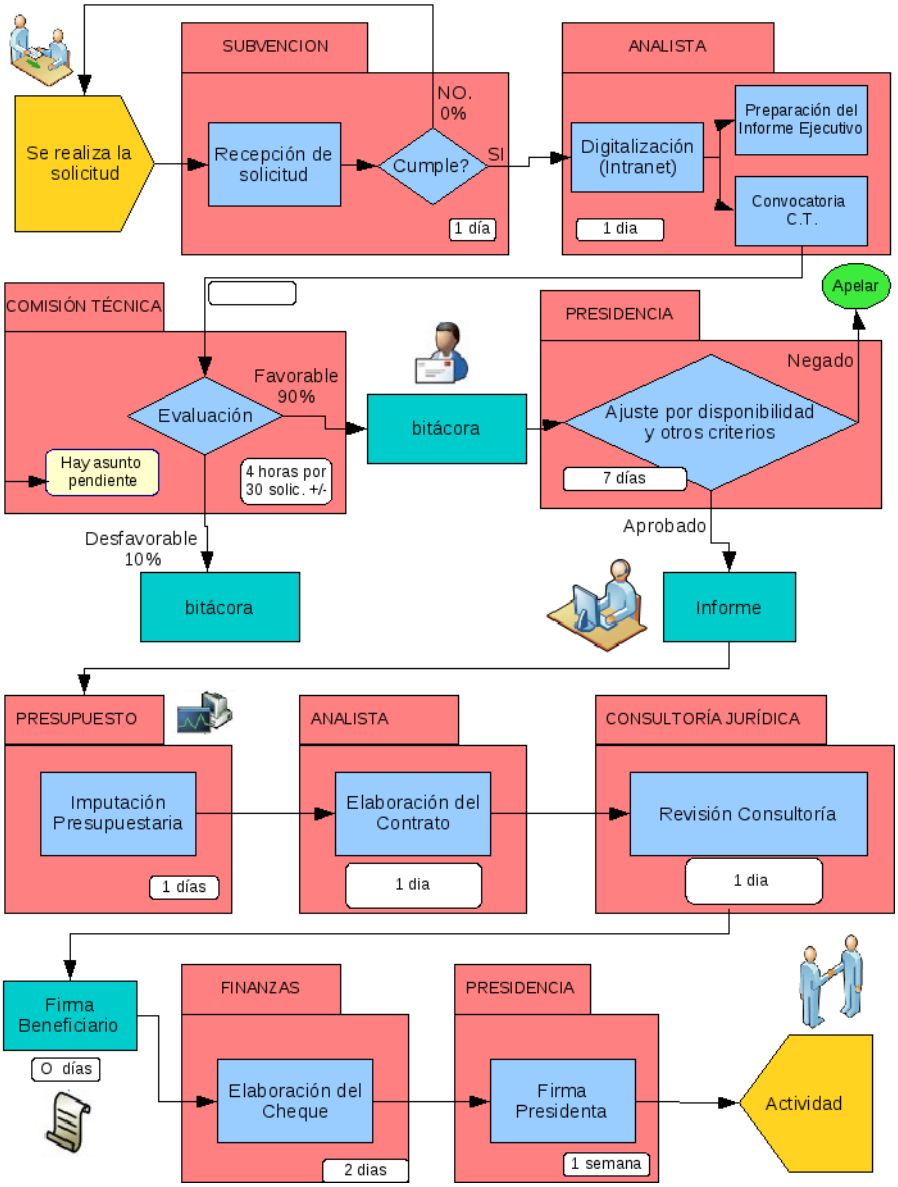


Figura 2.5: Un Proceso Organizacional

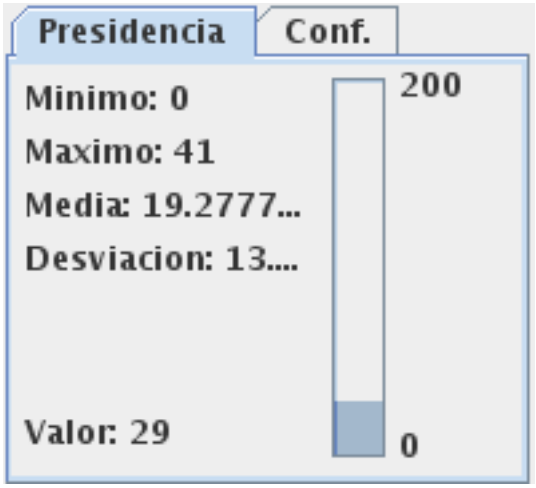


Figura 2.6: Ocupación de un servicio

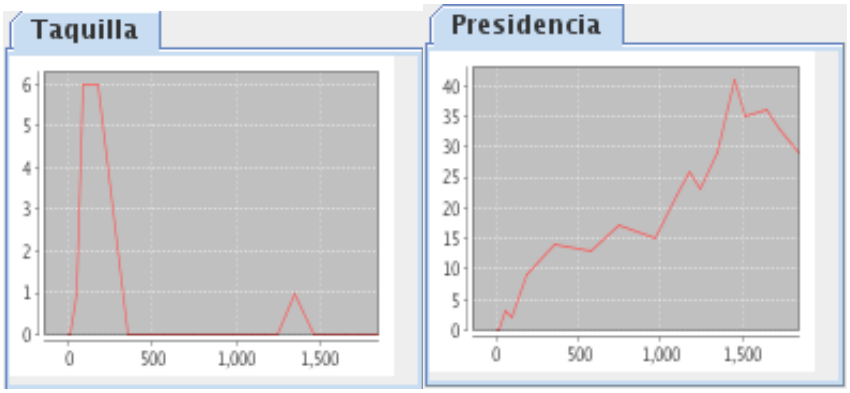


Figura 2.7: Inclinación hacia la estabilidad

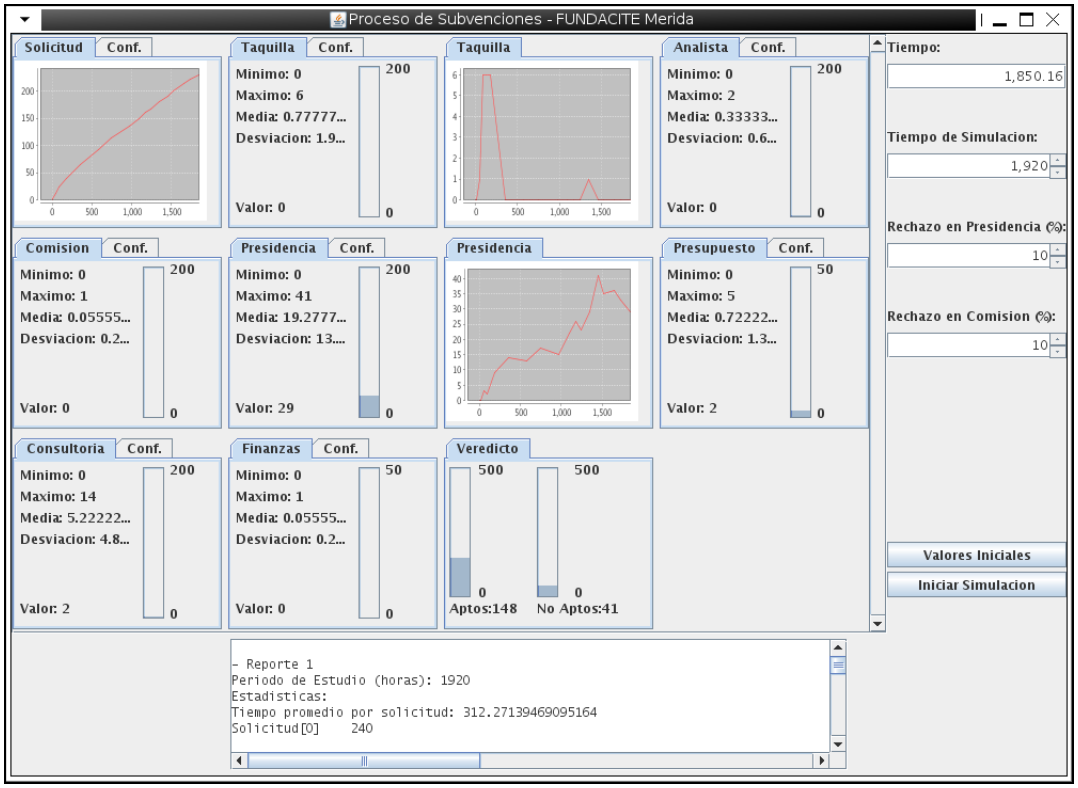


Figura 2.8: Interfaz al Usuario del Modelo del Proceso

un tomador de decisiones. Le permite saber si la configuración actual del sistema modelado se inclina hacia una conducta estable o deriva sin remedio hacia una situación insostenible en la que las colas de los servicios se llenarán y estos colapsarán. Todo esto es, desde luego, una mera aproximación en el dominio discreto a los análisis de estabilidad que se hacen con rigor matemático en el dominio continuo. Sin embargo, es igualmente útil y no mucho menos preciso.

2.5. Asuntos pendientes

No hemos hablado todavía de varios temas muy importantes para la simulación y realizables con Galatea. Galatea es una familia de lenguajes. Es más fácil describir reglas para los agentes en un lenguaje más cercano al humano, tales como

```
si esta_lloviendo_a_las(T) entonces
   saque_paraguas_a_las(T).
```

Exploraremos esa posibilidad en las siguientes secciones.

Otro tema especial es concurrencia. Java es multihebrado. Galatea también, pero no solamente gracias a la herencia Java. Tenemos previsiones para que una multiplicidad de agentes hebras se ejecuten al mismo tiempo que el simulador principal (y cualquier otra colección de hebras que disponga el modelista) en una simulación coherente. Modelar para aprovechar Galatea de esa manera requiere un dominio un poco más profundo los conceptos de sistemas multi-agentes a los que se le dedican los siguientes capítulos. Confiamos, no obstante, que el lector estará motivado, con los ejemplos anteriores, para continuar la exploración de las posibilidades que ofrece la simulación de sistemas multi-agente.

2.5.1. El porqué Galatea es una familia de lenguajes

Hemos dicho que Galatea es una familia de lenguajes y en esta sección vamos explicar porqué y a ilustrar cómo. La razón para la diversidad lingüística tiene profundas raíces en la comunidad de simulación. La experiencia global apunta a una necesidad básica de comunicación entre conocedores con diversas lenguajes de origen. No són solamente los lenguajes naturales, sino también los lenguajes de especialidad que separan, por ejemplo, a los matemáticos de los computistas, o a los prácticantes de las ciencias fácticas de los teóricos abstraccionistas.

Desde el comienzo, el proyecto Galatea se planteó combinar varios lenguajes, con la esperanza de ofrecer medios para romper barreras interdisciplinarias. Teníamos que comenzar por librar la separación entre los computistas y los usuarios de las computadoras. Por eso hablamos de los dos niveles de programación: Nivel Java y Nivel Galatea. El Nivel Java es el nivel de desarrollo de la plataforma y la elección de Java como lenguaje matríz tiene que ver con contar con las herramientas necesarias para intervenir la máquina. Algunos dirán que Java no es suficientemente de "bajo nivel" y tiene razón. Pero lo es para nuestros propósitos. En todo caso, Java no es el único lenguaje base. Prolog, a pesar de

ser un lenguaje de mucho más alto nivel en la escala Máquina-Humano (de la que estamos hablando), es usado en Galatea como plataforma para implementar los agentes.

Así, en el nivel Galatea ofrece(remos) a los usuarios de la plataforma de simulación otra familia de lenguajes. Al que llamamos propiamente lenguaje Galatea es una variante del lenguaje de simulación Glider [22, 23, 26, 24, 44, 43, 66, 70, 84, 85, 86] que, en su momento, se convirtió en una solución importante para los modelistas de sistemas. En el nivel Galatea, hemos enriquecido ese lenguaje acompañándolo con una serie de lenguajes de programación lógica que sirve el propósito específico de modelar agentes para simulación. Esos lenguajes son Openlog[11] y Actilog[12] y son ahora parte de la implementación Galatea, como se verá en el siguiente ejemplo.

2.5.2. Combinando lenguajes de programación para el cambio estructural: Java+Prolog

Vimos en la sección 2.4, cómo es posible usar la tecnología de simulación DEVS tradicional par dar cuenta de la conducta de un sistema de agentes. La perspectiva de agentes, no obstante, agrega un valor importante a esos modelos de colas y taquilla. Agregan la posibilidad del cambio estructural.

Cambio estructural es un concepto propuesto por la comunidad de economistas. Existe una revista cerrada dedicada al tema: *Structural Change and Economics Dynamics* que declara que su objetivo son los "trabajos sobre la continuidad y las rupturas estructurales en los patrones económicos, tecnológicos, conductuales e institucionales [..]. La revista también publica investigaciones puramente teóricas sobre la dinámica estructural de los sistemas económicos, en particular en el campo del análisis multisectorial, la aplicación de las ecuaciones diferenciales o en diferencias y la teoría de las bifurcaciones y el caos para analizar dinámicas económicas" [9]. Al parecer, el concepto es un intento de los economistas por plantear un problema teórico fundamental: el agotamiento de los modelos que no alcanzan a describir toda la dinámica de un sistema o sistemas que parecen requerir la composición de varios modelos tradicionales (por ejemplo, ecuaciones diferenciales por tramos) para dar cuenta de su conducta.

cambio estructural

En la comunidad de simulación el reto de los modelos de cambio estructural, como se les llama, ha sido asumido con una intepretación todavía más atrevida: modelos que cambian como consecuencia de su propia dinámica o, como decimos en la comunidad, que cambian en tiempo de simulación [25].

En Galatea defendemos una estrategia básica, que no pretende ser exclusiva, para abordar los modelos de cambio estructural. Podemos encargar a los agentes de vigilar y causar el cambio. En lugar de, por ejemplo, codificar un macro modelo que incluya todas las posibles variantes particulares de un sistema, la clásica estrategia para lidiar con los cambios estructurales, con Galatea podemos codificar en los agentes las reglas para cambiar de modelo. Permítasenos ilustrar

[9] Ver en Elsevier `http://www.elsevier.com` y Structural Change and Economics Dynamics `http://www.sciencedirect.com/science/journal/0954349X`

la estrategia sobre un ejemplo muy elemental y popular en simulación: el modelo de un banco.

El modelo de la taquilla[10] se ha convertido en un importante recurso didáctico en simulación. Un banco se reduce a un arreglo de taquillas frente a las cuáles se alinean los clientes que van llegando según un cierta regla de arribo y que son servidos también de acuerdo a ciertas otras reglas de servicio. Los clientes servidos se dirigen, al cabo del tiempo de servicio, hacia la salida. Esta es, de nuevo, la estructura cola-servicio que mostramos en la sección 2.4 y que, con la ayuda de los recursos estocásticos del simulador, puede servir para capturar la compleja dinámica de un proceso bancario en este caso.

Lo que proponemos hacer a continuación es extender ese modelo básico de las taquillas del banco para que dé cuenta del cambio estructural que suponemos ocurre en el banco cuando la regla de arribo desborda las capacidades establecidas por las reglas de servicios en las taquillas: un gerente razonable seguramente "abrirá más taquillas" en un ejercicio de adaptación que nosotros proponemos como una forma (primitiva pero válida) de cambio estructural.

2.5.3. El ejemplo del Gerente Bancario

Uno puede codificar un modelo flexible de un banco en el que se abran y cierren taquillas de acuerdo a las demandas de los clientes (es decir, la cantidad de clientes en el banco). Lo que se harían, normalmente, para lograr tal modelo es declarar una estructura máxima con todas las taquillas posibles en el banco y se ofrecería una manera de habilitar y deshabilitar taquillas segúun sea necesario. Esto es equivalente a programar un modelo de funciones matemáticas por tramos, en el que cada función se "activaría" en el tramo correspondiente para dar cuenta de la conducta de las variables dependientes, dado los valores que las independientes adquieren en ese tramo.

Lo que proponemos acá es sutil pero profundamente diferente. Proponemos que el propio modelo contenga el mecanismo, no para habilitar y deshabilitar subestructuras de la macro-estructura, sino para crear y eliminar directamente a las estructuras específicas. Galatea ofrece esta posibilidad gracias, en principio, a la plataforma subyacente orientada a los objetos y que permite crear y eliminar objetos con gran flexibilidad y robustez, pero también gracias a las previsiones para modelar agentes que se encargan de hacer el trabajo.

El modelo del Gerente [11] ilustra todo esto. Está definido por el siguiente código (códigos 9, 10 y 11) en el nivel Galatea que da origen a las clases y objetos, en el nivel Java, que se describen más adelante.

La conducta es idéntica a las descrita en el viejo modelo de las taquillas, salvo que ahora se incorpora un agente con capacidad para crear y eliminar taquillas de acuerdo a ciertas condiciones que observa en el sistema. El código del modelo del Gerente es un ejemplo de un modelo de simulación con un agente. Estamos en el proceso de desarrollo de un sofisticado traductor (el compilador Galatea)

[10] ver `demos.taquilla` en el repositorio

[11] ver paquete `contrib.gerente` en el repositorio

Algorithm 9 El Gerente Bancario. Primera Parte (NETWORK)

```
TITLE Sistema simple de tres taquillas
NETWORK
   Entrada(I){
      IT = 10;
      SENDTO(Taquilla[MIN]);     }
   Taquilla (R) [3]{
      RELEASE
      SENDTO(Salida);
      STAY = 45;     }
   Salida (E){}
...
```

Algorithm 10 El Gerente Bancario. Segunda Parte (AGENTS)

```
...
AGENTS
  Gerente {
   abd(cola_larga).
   abd(crear_taquilla).
   abd(taq_vacias).
   abd(elim_taquilla).
   abd(revisa_cola).

   observable(cola_larga).
   observable(taq_vacias).

   user_built(timing(_)).

   if timing(T)  then revisa_cola(T).
   if cola_larga then crear_taquilla.
   if taq_vacias then elim_taquilla.

   // código prolog para predicados user_built
   timing(T) :- gensym('', C), atom_number(C, T).  }
...
```

Algorithm 11 El Gerente Bancario. Tercera Parte (INTERFACE, DECL e INIT)

```
...
INTERFACE
    public void revisa_cola(double t, Agent agente, jpl.Integer at) {
      int clientes_en_banco = 0;
      for (int i = 0; i <  Taquilla3.taquilla.length; i++) {
         clientes_en_banco =+ Taquilla3.taquilla[i].getEl().ll();}
      if (clientes_en_banco > 10) {
         agente.inputs.add("cola_larga");        }     }

    public void queja(double t, Agent agente) {
        System.err.println("#"+agente.agentType+
            agente.agentId + ": Cola larga!!"); }

    public void crear_taquilla(double t, Agent agente) {
        if (Taquilla.mult > Taquilla.maxMult) {
           System.out.println("Banco lleno!!");
        } else {  add(Taquilla); } }

    public void taq_vacias(double t, Agent agente) {
    for (int i = 0, j = 0; i < Taquilla.mult; i++) {
      if (Taquilla3.taquilla[i].getEl().ll() +
          Taquilla3.taquilla[i].getIl().ll() == 0) {  j++;
      if (j > 1) {agente.inputs.add("taq_vacias");} } }}

    public void elim_taquilla(double t, Agent agente) {
       int i = 0;
       while ((i < Taquilla.mult - 1) &
            (Taquilla3.taquilla[i].getIl().ll() +
             Taquilla3.taquilla[i].getEl().ll() > 0)) { i++;}
        if (Taquilla3.taquilla[i].getIl().ll() +
            Taquilla3.taquilla[i].getEl().ll() == 0) {
            del(Taquilla);}}

DECL
  STATISTICS ALLNODES;
INIT
  TSIM = 300;
  ACT(Entrada,0); END.
```

Algorithm 12 gerente.kb

```
1 if_(timing(T), [revisa_cola(T)]).
2 if_(cola_larga, [crear_taquilla]).
3 if_(taq_vacias, [elim_taquilla]).
```

que podrá procesar esos códigos y convertirlos en código Java y Prolog para el sistema subyacente. Nos hemos propuesto que la convención para la traducción sea la siguiente:

1. Existe una clase principal del modelo guardada en su propio archivo `.java`. El traductor usará el nombre del archivo `.g` para nombrar esta clase. En este caso es `Taquilla3` (en el paquete `contrib.gerente`). El programador de la interfaz tendrá acceso a la clase principal usando ese identificador.

2. En la clase principal se declararán los objetos que representan a los nodos de la red, usando el mismo nodo que los define en la sección `NETWORK` del archivo `.g`, pero escrito con la primer letra en minúscula. En este caso, `Taquilla` da origen a `taquilla`, `Entrada` a `entrada` y `Salida` al objeto `salida` de ese tipo. `taquilla` será, inicialmente, un arreglo Java de 3 elementos debido a la capacidad declarada en el código con la instrucción `Taquilla (R) [3]`.

3. Todo método de la interfaz debe tener como primeros dos argumentos a `double t` y a `Agent agente`, de manera que el usuario pueda referirse al tiempo actual, con `t`, y al agente involucrado, con `agente` (otros nombres de variable son posibles. Lo importante es que sean del tipo correcto). Los restantes argumentos en esos métodos, si los hubiere, deben corresponder a tipos válidos en la biblioteca `jpl`.

4. Desde la interfaz el usuario tiene acceso a todos los métodos del nivel Java de las bibliotecas de Galatea.

Todo eso, desde luego, lo hará automáticamente el compilador Galatea. Pero siempre será posible que un modelista-programador pueda intervenir directamente en el nivel Java (procediendo con cuidado, desde luego).

A parte de todo lo dicho antes sobre cambio estructural, el modelo del agente es un excelente espacio para ilustrar porqué llamamos a Galatea una familia de lenguajes. Como ven, el código de la taquillas es el mismo código tradicional Glider, con algunas modificaciones que introduce Galatea para apoyarse en la sintáxis Java para manipular objetos (algoritmo `Taquilla3.java` en el repositorio). Pero también se incluye código en lenguajes de programación lógica para activar al agente gerente (algoritmo 12) y un código para que el simulador pueda interpretar las acciones propuestas por el agente sobre el modelo (algoritmo 13, noten que es el mismo código 11 salvo por el empaquetamiento en Java).

Algorithm 13 Interfaz.java

```
1  package contrib.gerente;
2  public class Interfaz {
3
4      public void revisa_cola(double t, Agent agente, jpl.
           Integer at) {
5        int clientes_en_banco = 0;
6        for (int i = 0; i <  Taquilla3.taquilla.length; i++) {
7           clientes_en_banco =+ Taquilla3.taquilla[i].getEl().
               ll();}
8        if (clientes_en_banco > 10) {
9           agente.inputs.add("cola_larga");        }     }
10
11     public void queja(double t, Agent agente) {
12         System.err.println("#"+agente.agentType+agente.
               agentId + ":␣Cola␣larga!!"); }
13
14     public void crear_taquilla(double t, Agent agente) {
15         if (Taquilla.mult > Taquilla.maxMult) {
16            System.out.println("Banco␣lleno!!");
17         } else {  add(Taquilla); } }
18
19     public void taq_vacias(double t, Agent agente) {
20     for (int i = 0, j = 0; i < Taquilla.mult; i++) {
21       if (Taquilla3.taquilla[i].getEl().ll() +
22           Taquilla3.taquilla[i].getIl().ll() == 0) {  j++;
23       if (j > 1) {agente.inputs.add("taq_vacias");} } }}
24
25     public void elim_taquilla(double t, Agent agente) {
26        int i = 0;
27        while ((i < Taquilla.mult - 1) &
28             (Taquilla3.taquilla[i].getIl().ll() +
29              Taquilla3.taquilla[i].getEl().ll() > 0)) { i
                    ++;}
30         if (Taquilla3.taquilla[i].getIl().ll() +
31             Taquilla3.taquilla[i].getEl().ll() == 0) {
32             del(Taquilla);}}
33 } }
```

Algorithm 14 gerente.main

```
/****************************************** libraries */
:- ['../../galatea/glorias/gloria/gloria.pl'].
% the agent's knowledge base
:- dynamic ghistory/1.
tracefile('gerente.dot').  % see flach/graphviz.pl
/************************************* control stuff */
% abducible predicates %
abd(cola_larga).
abd(crear_taquilla).
abd(taq_vacias).
abd(elim_taquilla).
abd(revisa_cola).
for_testing_only(nothing).
% we can use observables instead
observable(cola_larga).
observable(taq_vacias).
% built-in, user-defined predicates %
user_built(true).
user_built(G) :- xref_built_in(G), !.
% allowing any prolog builtin predicate user_built(timing(_)).
timing(T) :- gensym('', C), atom_number(C, T).
```

Se requiere, además, un código adicional (algoritmo 14), en el lenguaje Prolog, con el cual se incorporan a Galatea los agentes con un motor de inferencia desarrollado como parte del proyecto Gloria (`http://gloria.sourceforge.net`). Gracias a la interfaz Java-Prolog, provista por el subsistema JPL del Swi-Prolog, hemos podido incorporar a esos agentes basados en lógica dentro de los modelos de simulación del cambio estructural.

Estas extensiones permiten que un modelo tradicional de simulación, las taquillas de banco, incorpore elementos del cambio estructural. En este caso (ver los código de `contrib.gerente` en el repositorio Galatea), el banco que comienza funcionando con 3 taquilla, en ciertas condiciones de alta demanda (fijando el tiempo entre llegadas de personas en 1) y con suficiente tiempo (fijando el tiempo de simulación en 300), el banco termina con 11 taquillas, 7 nuevas abiertas durante la simulación, como muestra la figura 2.9.

```
Banco lleno!!
300.0         Scan (E)Salida[0]
300.0         Scan (R)Taquilla[10]
300.0         Scan (R)Taquilla[9]
300.0         Scan (R)Taquilla[8]
300.0         Scan (R)Taquilla[7]
300.0         Scan (R)Taquilla[6]
300.0         Scan (R)Taquilla[5]
300.0         Scan (R)Taquilla[4]
300.0         Scan (R)Taquilla[3]
300.0         Scan (R)Taquilla[2]
300.0         Scan (R)Taquilla[1]
300.0         Scan (R)Taquilla[0]
...
Sistema simple de tres taquillas con gerente
Time:              301.0
Time Stat:         301.0
Replication:         1
Date:              04/49/09 11:49:06
Elapsed time:      0h 1m 53.169s
```

Figura 2.9: Fragmento de traza del Banco con nueva estructura

Capítulo 3

Aprendizaje Computacional

3.1. Introducción

El Aprendizaje Computacional, AC, nuestra traducción del término *Machine Learning* (traducido también como Aprendizaje de Máquina en otros textos), es uno de los temas más difíciles y también uno de los más prometedores en el espacio tecnológico de la Inteligencia Artificial, IA. El tema es tratado por diversas comunidades bien establecidas ocupadas en temas que van desde la matemática (lógica y estadísticas), hasta ingeniería y biología. Entre las revistas más importantes de esta comunidad se cuentan *Machine Learning*[1], *Artificial Inteligence*[2] y *The Journal of Machine Learning*[3], pero solamente esta última ofrece acceso libre a su contenido.

Lo que parece animar este tema de estudio es la posibilidad de construir un dispositivo que, aún cuando se le haya atribuído una inteligencia muy primitiva en un principio, pueda aprender y mejorar su conducta gracias a ese aprendizaje. La codificación directa, en algún lenguaje de programación o representación del conocimiento, no cuenta como aprendizaje en el sentido que se pretende, a menos que tal codificación o representación sea producida por el mismo dispositivo. Tampoco cuenta en este sentido de aprendizaje *el-enterarse-de-algo*, una de las acepciones de la palabra *Learning* del idioma Inglés. Es decir, AC parece apuntar al más puro sentido de aprender que conocemos: el aprender del humano. Así que antes de abordar el tema, hacemos una revisión del concepto en esos términos amplios.

3.1.1. Una intuición sobre Aprendizaje

Las teorías de aprendizaje abundan. Una referencia obligada cuando se trata de aprendizaje humano es Piaget[73, 72], [41]y su escuela de la evolución

[1] Machine Learning `http://www.springerlink.com/content/100309/`

[2] Artificial Intelligence `http://www.elsevier.com/wps/find/journaldescription.cws_home/505601/description#description`

[3] The Journal of Machine Learning `http://jmlr.csail.mit.edu/`

psicogenética en los humanos. Piaget caracterizó, luego de estudios cuidadosos sobre niños, las etapas de desarrollo por las que transcurre un individuo desde la temprana infancia y hasta la adolescencia. En esos estudios, se postula una explicación del cómo el individuo va construyendo progresivamente estructuras mentales con las cuáles asimila, cada vez mejor, las relaciones que observa en su ambiente. Así, el individuo pasa de un estado de completa indefensión e ignorancia, a uno en el que comprende lo que pasa a su alrededor y puede articular razonablemente, y hasta explicar, su conducta en términos espaciales y temporales.

Los términos de esa contribución Piagetiana (y de sus seguidores) no son muy claros (quizás debido a un gran celo por la precisión). Sin embargo, dejan claro que el o la joven recorre mientras crece una serie de etapas en las que su sistema de creencias y la lógica subyacente se sofistican, dando lugar a la comprensión a la que se refiere. Que esto ocurra normalmente en la especie, por designio genético, significaría (para una postura racionalista) que nuestra biología está pre-programada para sostener la capacidad de aprender.

3.1.2. El papel de la memoria

Se suele menospreciar en caracterizaciones como esa el crítico papel que desempeña nuestra memoria en el proceso de aprendizaje. No se trata solamente del dispositivo para registrar la data sensorial. Antes, se trata de un almacén estructurado para dar sentido, así como un "sistema para olvidar" que permiten, entre otras cosas, que aquello que más nos importa sea más fácilmente accesible.

Memorizar no es, sin embargo, una capacidad exclusiva de los organismos biológicos. En un sentido casi trivial (pero no trivial), se memoriza un datum tan pronto se lo "anota" en algún medio físico que permite su "recuperación" posterior. Cualquier registro (escrito, impreso o grabado de algún modo) es memoria si existe una forma de leerlo. Así definida, se puede clasificar como memoria a una increíble variedad de medios y sistemas físicos y biológicos, orgánicos e inorgánicos: Desde el silicio, esa piedra con sus inusuales propiedades electromagnéticas, hasta los sistemas inmunológicos, con una también inusual red de caminos biológicos que recuerdan las amenazas al cuerpo y, así, lo inmunizan. Pasando, en ese enorme espectro, desde luego, por las redes de células neuronales.

3.1.3. Redes Neuronales

Las redes neuronales llevan el concepto de memoria a otro nivel. Ya no es solamente el cambio de estado que registra el datum. Con las redes es una configuración estructural de muchos estados (en los correspondientes dispositivos independientes) que se organizan para representar patrones complejos.

A las redes se les suele atribuir 2 características claves: robustez (pués la estructura es capaz de funcionar luego de una pérdida parcial de sus componentes) y otra asociada a la robustez, la flexibilidad (pués el sistema funciona recuperando patrones a partir de "pistas" parciales). Nos interesa acá rescatar otro

atributo de las Redes Neuronales que suele permanecer implícito: autonomía funcional o, para usar un término muy oscuro: **activabilidad**. Las redes neuronales son sistemas autónomos para efectos de activación y recuperación de la memoria. Una red puede leer un patrón para aprehenderlo o identificarlo, asociándolo con una respuesta, sin intervención de otros sistemas. No hace falta un lector separado para que la red funcione como memoria, ni para grabar, ni para recuperar. Esta característica abre la posibilidad de sistemas con memoria que pueden reaccionar oportuna e independientemente, usando esa memoria, a estímulos externos.

3.1.4. Redes Neuronales Artíficiales y el Aprendizaje Conectivista

La descripción anterior es la abstracción usada en la Ingeniería de la Inteligencia Artificial para crear redes neuronales artificiales. Básicamente se trata de una red de dispositivos muy simples: **sumadores** que admiten cierta cantidad de entradas, suman los valores en las entradas y toman una decisión puntual que reportan a través de una conexión de salida[50].

La red aprende cuando se le conecta (en corto circuito) con ciertas entradas y (Y) ciertas salidas. Una cantidad de mecanismos y algoritmos[46] han sido implementados para balancear esa relación entre entradas y salidas, normalmente ajustando los **pesos** que definen al modelo matemático de cada neurona. De esta forma, los pesos se convierten en el mecanismo de memoria pues, una vez fijados, pueden recuperar en la red neuronal las mismas salidas, dadas las mismas entradas (o unas no tan "similares").

La redes neuronales artificiales han tenido sus altos y bajos. Chomsky habló muy temprano y con mucho entusiasmo[5] (página 189) de esa "función de pesos", determinada empíricamente, que le permitiría al aprendiz de un lenguaje seleccionar la gramática apropiada para su lenguaje, a partir de una forma general universal. Se trataba (es decir, se trata) de una solución posible al problema de aprendizaje de una lenguaje a partir de los escasos datos y en el poco tiempo con el que logra hacerlo cualquier infante. Como veremos más adelante, ese es uno de los más grandes desafios a cualquier teoría del aprendizaje humano (o artificial).

Luego de los primeros resultados con un tipo de red neuronal (el perceptrón, [50] página 50), muchos creyeron que la abstracción era definitivamente insuficiente para describir (y realizar) procesos de aprendizaje. También se ha discutido sobre la imposibilidad de explicar lo que la red aprende en términos significativos para los humanos. Ciertamente la propia red no puede explicar "sus razones". Sin embargo, no se debe confundir esto con la imposibilidad de conectar a la red con una representación lógica del conocimiento.

3.1.5. Taxonomía de sistemas de aprendizaje: Supervisados o no supervisados

Las contribuciones en aprendizaje computacional (es decir, *Machine Learning*) han proliferado y madurado al punto de que es posible hablar ya de una taxonomía de técnicas de aprendizaje. La clasificación que se usa con mayor frecuencia para ubicar cada contribución, clasifica a los sistemas como 1) de aprendizaje no supervisado, 2) de aprendizaje supervisado y 3) de aprendizaje con refuerzos. Es común admitir que el problema de aprendizaje es planteado como uno de búsqueda de formas de clasificar un conjunto de instancias, cada una de las cuáles es una lista de pares característica-valor (o atributo-valor). "Si las instancias se presentan etiquetadas (es decir, se asocia cada una con alguna salida correcta particular) esa forma de aprendizaje se considera supervisada, a diferencia del aprendizaje no supervisado, en el cual no hay tales etiquetas"[53]. Los aprendizajes no supervisados, en ese caso, son programas que buscan asignar etiquetas útiles a las instancias, clasificándolas en el proceso. El aprendizaje con refuerzo es también una forma de aprendizaje supervisado, para algunos, pero una en la cuál la etiqueta es reemplazada por un valor que un tutor externo le otorga como calificación a cada esfuerzo de clasificación en un proceso iterativo (si el valor es uno entre positivo o negativo y las iteraciones se pueden acumular como una secuencia de ejemplos, el aprendizaje con refuerzo se reduce a aprendizaje supervisado).

Los agentes que presentaremos a continuación parecen tratar de escapar a esa taxonomía. Los agentes son entidades dotadas de (algún nivel de) autonomía. La condición de supervisados no parecería encajarles muy bien. Lo cierto, sin embargo, es que cada agente está plantado frente a un ambiente que le informa del resultado de sus acciones. Como se muestra a continuación, esa retroalimentación puede servir para que el propio agente (a través de su "crítico interno"), etiquete instancias como pertenecientes o no a ciertas clases. En particular, nuestro modelo aprovechará todo reporte de acción fallida, que llegue al agente desde el ambiente, para reportar que la meta que esa acción estaba tratando de alcanzar es una instancia que no pertenece a la clase de referencia o, en otras palabras, que es un ejemplo negativo.

Normalmente, en el aprendizaje supervisado se requiere de un conjunto de ejemplos positivos (o etiquetados como que sí pertenecen a una cierta clase de referencia) y un conjunto de ejemplos negativos (o etiquetados como que no pertenecen a la clase de referencia) y el trabajo del motor o programa de aprendizaje es encontrar la regla que discrimina entre ambos conjuntos. Es decir, la regla que describa a las instancias positivas y no a las negativas.

En la adaptación que se presenta a continuación, de un motor de aprendizaje para un agente, requerimos que este tenga una regla inicial, que se supone que describe las instancias positivas y le daremos los ejemplos negativos a partir de acciones fallidas como acabamos describir. La tarea del motor de aprendizaje será, entonces, corregir la regla que describe a los positivos para que no incluya a esos nuevos ejemplos negativos.

Estas adaptaciones de la definición de problema de aprendizaje todavía son,

sin embargo, insuficientes para permitir la inclusión de un subsistema de aprendizaje en un agente reactivo y racional como el que hemos estado describiendo. Una variante fundamental de la estrategia de aprendizaje parece ser necesaria.

3.2. Aprendizaje Superficial

Aprendizaje Superfiicial es el objetivo de esta sección. Acá reportamos una exploración de la conexión entre la programación lógica inductiva, ILP [67, 69, 18], y el modelo de agentes basado en lógica de Kowalski y Sadri[58, 54], que hemos usado para construir un modelo referencial de sistemas multiagentes. Como resultado de esa exploración, se presenta en esta sección el modelo modificado del agente Gloria, con capacidad para aprender mientras actúa.

El modelo de agente que hemos venido usando en este texto tiene una historia con muchos detalles. La Escuela de la Programación Lógica se cruzó con la noción sicologista de agente a finales del siglo pasado, de la mano de Kowalski[58] y como efecto colateral (inevitable, en retrospectiva) de un esfuerzo por construir una teoría de la lógica abductiva[54, 49, 35, 58]. Es el mismo concepto de **Abducción** propuesto por Charles Sanders Pierce, hace más de un siglo y que a mediados del siglo XX fuera propuesto tímidamente por Noam Chomsky[5], como la idea que podría conducir a la explicación de lo que llamó el gran problema de adquisición del conocimiento lingüístico (.ibid, pag 89-90). Permítasenos esta cita que reune las palabras de Pierce y de Chomsky al respecto (.ibid, pag 90):

> "La manera en que he estado describiendo la adquisición del conocimiento de un lenguaje, me lleva a pensar en una muy interesante y un tanto olvidada charla dictada por Charles Sanders Pierce hace más de 50 años[4], en la que desarrolló ciertas nociones similares pero acerca de la adquisición del conocimiento en general.
>
> Pierce argumentaba que los límites generales de la inteligencia humana eran mucho más estrechos de lo que podrían sugerir las suposiciones románticas acerca de lo ilimitada de la perfectibilidad humana (o, de hecho, de lo que sugieren sus propias concepciones "pragmáticas" sobre el curso del progreso cientifico en los más conocidos estudios filosóficos). Pierce creía que las limitaciones innatas sobre hipótesis admisibles son una precondición para la exitosa construcción de teorías, y que el "instinto adivinatorio" que produce las hipótesis, hace uso de los procedimientos inductivos sólo para "acción correctiva". Pierce sostenía en su exposición que la historia temprana de la ciencia muestra que muchas teorías correctas fueron descubiertas con extraordinaria facilidad y rapidez, a partir de una base de datos muy inadecuada y recién se descubrían ciertos problemas. Hablaba de "cuan pocos intentos tuvieron que hacer esos hombres de gran genio antes de acertar con las leyes de la naturaleza"

[4] C.S.Pierce, "the logic of abduction", nota de Chomsky

> y luego se preguntaba "cómo fue conducido el hombre a siquiera considerar cierta teoría correcta?. No puedes decir que ocurrió por suerte, puesto que las probabilidades son demasiado grandes en contra de que esa teoría, en los veinte o treinta mil años durante los cuales el hombre ha sido un animal pensante, surja en la cabeza de alguno de ellos".
>
> A fortiori, las probabilidad está todavía más en contra de que la verdadera teoría que explica cada lenguaje surja en la cabeza de todo niño de 4 años. Pierce agregaba "la mente del hombre tiene una adaptación natural para imaginar teorías correctas de ciertos tipos. . . Si el hombre no tuviera el don de una mente adaptada a esos requerimientos, no podría haber adquirido conocimiento alguno".
>
> En consecuencia, en nuestro caso actual, pareciera que el conocimiento de un lenguaje, su gramática, sólo puede ser adquirido por un organismo que está "preconfigurado" con una fuerte restricción sobre las gramáticas posibles. Esa restricción innata es una precondición, en el sentido Kantiano, para la experiencia lingüística, y parece ser el factor crítico al determinar el curso y resultado del aprendizaje lingüístico. El niño no puede saber al nacer cual lenguaje aprenderá, pero debe saber entonces su su gramática debe tener cierta forma predeterminada que excluye muchos lenguajes imaginables. Habiendo seleccionado una hipótesis permitida puede usar evidencia inductiva para la acción correctiva, confirmando o desconfirmando su elección. Una vez que la hipótesis está bien confirmada, el niño sabe el lenguaje definido por su hipótesis. Como consecuencia, su conocimiento va mucho más allá de la experiencia y, de hecho, le ayuda a caracterizar mucha de la data en su experiencia como defectuosa o confusa"[5].

Esta larga cita presenta la mejor especificación informal que hemos podido encontrar para lo que llamamos **Aprendizaje Superficial**. Nuestra intención, ambiciosa sin duda, es concretar una especificación técnica detallada del concepto y, luego (es decir, como siguiente paso probablemente en varias iteraciones) producir una realización computacional en forma de prototipo, como prueba práctica del concepto. Al final de este capítulo presentamos la primera realización computacional y la comparamos con un ejercicio de optimización.

En 1997, en un ejercicio similar [10](de prueba del concepto) produjimos un prototipo computacional de un modelo básico de un motor de razonamiento abdutivo, que sirve como el sistema de planificación de ujn agente orientado a metas[79]. ESe motor estuvo basado originalmente en un procedimiento de prueba de teoremas, el iff[35], y nos ha servido desde entonces para varios desarrollos, en particular la familia Galatea de lenguajes para modelar agentes para simulación [5] y una plataforma multiagente para computación científica. Vamos, a continuación, a repasar la exploración que hemos hecho para incluir

[5] Repositorio Galatea `http://galatea.sourceforge.net`

los agentes aprendices en Galatea. Quisimos explorar la relación entre la reducción de metas en sub-metas, entrelazada con la ejecución de acciones, por un lado, y la conducta adaptativa de un agente que puede aprender de sus fallas al actuar. Aún cuando el modelo de Agente de Kowalski [55]toma en cuenta lo que ocurre en el agente cuando falla una acción, no explica qué ocurre cuando la acción parece tener éxito y sin embargo la meta superior que se suponía alcanzaría no es alcanzada. Eso puede deberse a que las creencias, que el agente usa para reducir sus metas superiores a acciones, incluyen algunas creencias falsas que podrían ser revisadas en un proceso de aprendizaje.

En aquel trabajo exploratorio ([10], capítulo 6), sugerimos que habría una conexión entre los sistemas deductivos etiquetados [37] y el marco abductivo para agentes que se propone en el modelo de Kowalski. Dicha conexión se materializaría usando las metas para etiquetar a los planes que las procuran. Tal recurso serviría para rastrear la intención original detrás (arriba?) de cada acción al ser ejecutadas. De hecho, en la solución que hemos implementado, este mismo recurso es crucial para producir la tarea de aprendizaje a la que se someterá el agente, una vez que se comprueba el problema de creencias falsas en su base de conocimiento. Es decir, las fallas al actuar, precipitan un proceso de aprendizaje automáticamente.

La solución de aprendizaje que se propone en esta sección no está dirigida a computación de alto rendimiento, sino es más bien una forma de aprendizaje superficial . La estrategia se puede resumir en que el agente explora las relaciones más superficiales entre las metas, representadas por reglas en lógica, para detectar regularidades y patrones que pueden estar influyendo en sus fracasos al actuar. Pero el agente no lanza una operación masiva de aprendizaje, sino que explora el "vecindario" de su configuración actual. Algunos primeros experimentos [16], con un muy conocido sistema de aprendizaje automático [68], nos hicieron confiar en la posibilidad de integrar tales mecanismos en un entorno de simulación, objetivo que hemos logrado completamente en Galatea, tal como se explica hacia final de este capítulo.

aprendizaje superficial

Una aprendiz superficial podría funcionar como un muy apropiado asistente para análisis de datos asistido por el computador, dirigido por expertos humanos. Los dominios complejos, como la genómica o la socio-economía, en los que el análisis parece requerir de una capacidad para la búsqueda sistemática y muy rápida de patrones, pero también de las intuiciones que desarrollan los seres humanos, podrían estar mejor servidos por una estrategia de aprendizaje caracterizada por "humanos en el ciclo" con uno o más dispositivos automáticos. Es importante destacar que un aprendiz superficial no sólo debe ser capaz de interactuar con esos expertos, sino también de producir una representación del conocimiento aprendido que sea comprensible para los humanos. Algunos dirán que sea una representación lógica. Hemos visto cómo usar lógica para representar conocimiento en el agente y al propio agente. Veamos ahora todo eso junto con los medios de representación de un proceso de aprendizaje computacional.

3.3. Gloria: Un agente lógico, basado en lógica y aprendiz superficial

Gloria[6] es el nombre código que le hemos dado a nuestra implementación del modelo de agente basado en lógica que describe Kowalski en su nuevo libro [55]. El componente principal de Gloria es un programa Prolog que implementa el procedimiento de prueba de Fung y Kowalski (*if and only if proof procedure*, [35]), el cual se usa para procesar las metas del agente y así producir los planes de acción que agente ejecuta. Básicamente, convertimos a ese procedimiento de prueba en un motor de planificación para el agente y luego lo integramos con un motor de aprendizaje, como se verá más adelante.

Un Agente es alguien que hace algo por alguien más (esta es una de la acepciones comunes en Inglés y Español). La raíz en Latin de la palabra es *Agere*, el que hace. La diferencia ahora, parece ser, que el que hace no es alguien, sino algo. En la historia tecnológica contemporánea, la palabra Agente fue reciclada a principios de los años 1990, para enfatizar la necesidad de que los sistemas de Inteligencia Artificial, IA, terminaran haciendo algo. Fue Rodney Brooks, profesor del MIT, uno de los campeones de la idea de olvidar completamente las representaciones, en procura de conducta inteligente y real la cual, sin duda, no es sólo astuta, sino oportuna. Gloria es parte de un esfuerzo por responder a esa primera exigencia (construir sistemas que hagan algo, no sólo pensar), sin tener que claudicar las representaciones como propuso Brooks. Hay más detalles de ese esfuerzo en [10].

Recapitulando sobre lo que hemos venido diciendo, en la teoría de Kowalski, un agente es un objeto con un estado interno, pero que está permanentemente ocupado en interacciones con su ambiente. El agente observa el mundo (recolectando "perceptos", unidades de observación que pueden ser representadas como proposiciones describiendo su entorno y los eventos que en él ocurren) y actúa sobre él (postulando acciones al ambiente para que sea ejecutadas).

estado interno del agente

Para caracterizar ese estado interno del agente, se requieren esencialmente, dos tipos de estructuras de datos [7]:

Metas (u objetivos)

1. **Metas (u objetivos)** , que contienen descripciones de las intenciones generales del agente. Es decir, las declaraciones de las relaciones permanentes entre el agente y su entorno, a las que llamamos metas de mantenimiento o implicaciones y, otras, derivadas de las metas de mantenimiento, que declaran las tareas que serán reducidas a planes de acción, llamadas metas de logro.

Creencias

2. **Creencias** , que contienen las reglas de conducta del agente.

En la siguiente sección explicamos como usar una especificación con esas estructuras para convertir al agente en un dispositivo de planificación automática primero, y luego en un dispositivo que aprende.

[6] Repositorio de Gloria `http://gloria.sourceforge.net`

[7] Otras teorías proponen otras estructuras e.g. creencias, deseos e intenciones. Ver Agentes BDI[77, 61]

3.4. Una breve historia de la Planificación

La planificación es un problema con una voluminosa tradición académica[9] y la planificación automática (o planificación con el computador) tiene una sorprendente (y también larga) historia en la Inteligencia Artificial. Permítannos concentrarnos en una variante que hemos podido explorar a propósito del modelo de agente en lógica: la planificación por abducción. La abducción es una técnica con cada vez más arraigo en los sistemas basados en lógica. Ha sido usada en combinación con la lógica temporal de Allen [2] y varias veces con sistemas de programación lógica (por ejemplo: [29, 28], [81], [30], [64] y [74]). Se trata de, básicamente, el mismo principio de Pierce que mencionamos al principio del capítulo, a propósito de la cita de Chomsky, pero acá aplicado a una variedad de problemas (Ver en [4] una descripción general de las posibles aplicaciones).

abducción

Cuando se habla de planificación por abducción, normalmente se identifican como "abducibles" las acciones que se pueden realizar para alcanzar una meta dada. Por ejemplo, dada la meta g y la regla $g \leftarrow a_1 \wedge \ldots \wedge a_n$, el plan $a_1 \wedge \ldots \wedge a_n$ podría ser producto "abducido", siempre que cada a_i haya sido considerada como abducible. En este trabajo usamos esa y otras formas de reglas que también aprovechan la estrategia de reducir metas a sub-metas abducibles.

Muchos de los sistemas lógicos existentes usan el cálculo de eventos o algún otro formalismo basado en eventos, para realizar el razonamiento temporal requerido por cada problema de planificación. De acuerdo con Reiter [78], no hay alternativa sino la abductiva cuando se trata de razonar con una lógica temporal basada en eventos. Para facilitar la referencia, sin embargo, permítannos resumir los atributos de la estrategia abductiva y basada en eventos que resultan atractivos para aplicaciones de planificación:

1. La historia de los sistemas de planificación en IA (Ver [80] y la revisión en [10]) indica un cambio de sistemas y algoritmos para producir secuencias de pasos en planes que son completas, perfectamente ordenadas y bien definidas, hacia sistemas que realizar alguna forma de **planificación parcial** , en la cual los planes son sólo parcialmente descritos dejando al ejecutor amplia maniobra para escoger un orden particular para las acciones al momento de ejecutarlas[8]. Esta estrategia de compromiso mínimo (*least commitment strategy* [80]) puede ser obtenida naturalmente con un sistema abductivo y orientado a eventos que mantenga el problema de planificación **abierto** para actualizaciones durante la ejecución.

   planificación parcial

   planificación abierta

2. En términos conceptuales al menos, las "abstracciones a nivel meta",

[8] Existe una confusión desafortunada en la terminología que se usa en planificación en la comunidad de IA. A los planificadores parciales se le llama erróneamente **planificadores no-lineales**, para distinguirlos de los planificadores completos y lineales de las primeras generaciones. La etiqueta *no-lineal* es equivocada porque sugiere que el proceso de generar el plan es no lineal, cuando lo que se pretende es decir que el producto del proceso: el plan, es no-lineal, lo que, a su vez significa que el orden de los puntos de tiempo o intervalos en el plan no está explícita y completamente definido. Un objeción similar se puede hacer, desde luego, acerca de la etiqueta *lineal*. En este texto usamos la terminología de [80] que es la más común en la comunidad de representación del conocimiento.

meta-descripciones

como las que Reiter supone son indispensables con la abducción, son, además, muy deseables. Las meta-descripciones (descripciones que hablan de otras descripciones) se puede amalgamar con descripciones a nivel de objetos para obtener especificaciones muy expresivas. Nociones como "las actividades mentales del agentes" pueden así ser capturadas en la representación, como se muestra en [10]. Se ha dicho [82], incluso, que la planificación es una actividades abductiva a nivel meta y, por tanto, así debe ser representada.

planificación por regresión

planificación progresiva

3. Un marco operacional basado en eventos se presta para planificación tradicional por regresión , en la que la siguiente acción a ser realizada por el agente se determina de último, luego de identificar todas las acciones razonando hacia atrás en la línea temporal desde el estado objetivo. Pero también se presta para **planificación progresiva** , en la que la siguiente acción a ser realizada por el agente es calculada primero (cuando una identificación de todas las otras acciones no es requerida). El sistema puede, entonces, ser cortado a la medida de problema de planificación particular. Nosotros hemos insistido que en el caso de la planificación reactiva (o planificación para un agente reactivo), un planificador progresivo es mucho más apropiado (.ibid, [11] y [12]).

algoritmo reentrante o corutina

4. Finalmente, debido a que la "línea del tiempo" no tiene que estar explícitamente representada, como ocurre en el cálculo de situaciones, un planificador basado en eventos es más fácil de implementar como un **algoritmo reentrante o corutina** (*any-time algorithm*). Este tipo de algoritmo puede ser interrumpido en cualquier etapa de procesamiento (para que su proceso sea entrelazado con otros procesos) y, si se le permite correr por más tiempo (se le asignan más recursos), producirá mejores (y más refinados) planes.

Tradicionalmente se ha supuesto que la abdución implica un test o prueba de consistencia sobre las explicaciones generadas "con respecto a nivel objeto de la axiomatización del dominio"[78]. Esto suele suponer, además, que se requiere un probador de teoremas en el nivel meta (como si los hubiera de otro tipo) que 1) identifique las explicaciones abductivas[9], 2) que construya una teoría que combine la axiomatización original con el conjunto de explicaciones abducidas y 3) verifique la consistencia de la nueva teoría. Si esas explicaciones están escritas como sentencias cualquiera de la lógica de primer orden, LPO, ese test sería equivalente a una prueba de consistencia sobre un conjunto de oraciones en LPO que no es, en general, ni siquiera **semi-decidible**[80]. Por lo tanto, tal

[9] Esta identificación sólo es posible en el nivel meta, es decir, usando un meta-lenguaje que se refiera acerca de la axiomatización al nivel objeto. Por esto se habla de probador en el "meta-nivel". Sin embargo, uno podría decir que los otros dos pasos también requieren de tales capacidades en el meta-nivel, para referirse a la teoría que se está construyendo y sobre la que se está chequeando la consistencia. Probablemente a eso se refiera Reiter con su "test de consistencia en el meta-nivel" [78]. Para el lector que quiera aterrizar esta discusión sobre un ejemplo, le invitamos a considerar las meta-referencias que se muestran en el algoritmo 16 escritas así: `abd(do). abd(dob). abd(c). abd(b).`

prueba es **no-computable** en el caso general. Sin embargo, uno puede imponer restricciones sobre la forma de las teorías y las explicaciones abducidas para evitar el caso general. Por ejemplo, suponiendo que la teoría original es consistente, uno puede concentrarse en el chequeo de consistencia de las explicaciones abducidas únicamente. Este tipo de sub-prueba de consistencia se sabe que es computable para ciertas formas de abducibles (ver [63] para una prueba de este en el contexto del cálculo de eventos).

Las pruebas de consistencia se realizan con el fin de garantizar la integridad de las explicaciones. Hay, sin embargo, otras manera de lograr lo mismo. El diseñador de la axiomatización puede establecer restricciones de integridad, oraciones que describan las condiciones con las que la información almacenada en la teoría (la base de conocimiento) debe ser consistente. A esto se le conoce [29] como la **interpretación de consistencia** de las restricciones de integridad, que requiere que la teoría ampliada conformada por la teoría inicial, las explicaciones y las restricciones de integridad, sea consistente. En un primer momento, las restricciones de integridad no agregan nada sino complejidad a problema de probar la consistencia (porque habrá más oraciones lógicas que verificar). Sin embargo, si uno asume que la axiomatización es internamente consistente, entonces la prueba de consistencia involucra sólo a las explicaciones y a las restricciones de integridad.

interpretación de consistencia

Por otro lado, las restricciones de integridad puede ser interpretadas como metas a ser "logradas" por el sistema. Esta es la **interpretación tipo teorema** (de las restricciones de integridad) [29], que demanda que las restricciones de integridad sean derivables a partir de la nueva teoría (la axiomatización más las explicaciones). La interpretación tipo teorema es, en general, más "fuerte" (es decir, más restrictiva) que la interpretación de consistencia, aunque las dos coinciden para teorías que tienen la forma de completitud si y sólo sí en programas lógicos jerárquicos [34]. El punto interesante es que tipo teorema es siempre semi-decidible y puede ser usada 1) para impedir planes incorrectos (como los que alerta Pelavin en [71]), que surgen en el contexto de la planificación con acciones concurrentes y 2) para concentrar todo la computación en pruebas de reducción de metas, eliminando la necesidad de un procedimiento separado para verificar la consistencia.

interpretación tipo teorema

En este trabajo usamos la interpretación de las restricciones de integridad para decir que equivalen a metas del agente que deben ser reducidas a sub-metas (por medio de pruebas de implicación lógica) que luego el agente puede intentar ejecutar como acciones. Si tiene éxito, ello significa que el agente es de hecho logrando las metas luego de haberlas planificado correctamente. Es por todo esto que usamos programas lógicos abductivos y restricciones de integridad, interpretadas como metas, como los recursos básicos de un agente planificador.

3.4.1. Qué es un Programa Lógica Abductivo?

En [34] Fung y Kowalski definen un **programa lógico abductivo** como

programa lógico abductivo

sigue[10]:

Un programa lógico abductivo es una tripleta $< T, IC, Ab >$ donde:

1. T es un conjunto de **definiciones** con la forma **si-y-solo-si**:

$$p(X_1, \ldots, X_k) \leftrightarrow D_1 \vee \ldots \vee D_n \quad k, n \geq 0 \tag{3.1}$$

donde, a su vez, p se define como un símbolo de predicado distinto a la igualdad ($=$) y de cualquier símbolo de predicado usado en Ab, las variables $X_1, \ldots, X_k$ son todas distintas, y cada D_i is una conjunción de literales. Cuando $n = 0$, la disjunción es equivalente a **falso.** Observe, sin embargo, que uno de los disjuntos D_i podría ser una conjunción colapsada en un *verdad*, en cuyo caso la disjunción sería equivalente a **verdad**. Existe una definición para cada símbolo de predicado p. En general, dada una meta $p(y_1, \ldots, y_k)$, con $y_i = X_i\theta$, diremos, para cada variable involucrada que:

$$(D_1 \vee \ldots \vee D_n)\theta \tag{3.2}$$

es un conjunto de planes alternativos para alcanzar esa meta y, por tanto, que cada D_i es un plan (para lograr esa metal). Si el agente es en procura de más de una meta, los planes correspondientes pueden, desde luego, combinarse de muchas maneras.

2. IC (el conjunto de restricciones de integridad) es un conjunto consistente con implicaciones que tienen la forma:

$$A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n \quad m, n \geq 0 \tag{3.3}$$

donde cada A_i y B_i es un átomo. Cuando $m = 0$, la disjunción es equivalente a **falso**. Cuando $n = 0$, la conjunción es equivalente a **verdad**.

3. Ab es el conjunto de todos los símbolos de predicado abducibles, llamados los **abducibles**, todos diferentes de $=$ y de cualquier predicado definido en T.

El átomo $p(X_1, \ldots, X_k)$, del predicado definido en T se dice que es la *cabeza* de la definición. Las variables $X_1, \ldots, X_k$ están implícitamente cuantificadas universalmente, con alcance igual a toda la definición. Cualquier variables que aparezca en uno disjunto D_i de una definición y que no es una de las $X_1, \ldots, X_k$ está implícitamente cuantificada existencialmente, con alcance igual al disjunto. Fung [36] agrega también la restricción de que la definiciones deben ser de "rango restringido", es decir, toda variable que aparezca en un disjunto D_i de una definición debe aparecer también en un literal no-negativo dentro del mismo D_i o en la cabeza de la definición.

[10] Las nociones básicas como átomo, literal y forma si-y-sólo-si las usamos como es común en la comunidad de la programación lógica y cómo se les define en [56] y en [47].

3.4.2. Qué es una consulta?

Una **consulta** es una fórmula con la forma siguiente:

$$B_1 \wedge \ldots \wedge B_n \quad m, n \geq 0 \tag{3.4}$$

donde cada B_i es un **literal**, i.e. un átomo o la negación de un átomo. Todas las variables en la consulta son consideradas como cuantificadas existencialmente (No se permite otra forma de cuantificación).

En el contexto de planificación, una consulta es una conjunción de metas cuya "alcanzabilidad" conjunta está siendo tratada por el procedimiento de prueba. La información para probar que las metas son alcanzables se extrae normalmente de la base de conocimiento. El procedimiento abducible, sin embargo, permite otra fuente de **información de soporte** al razonamiento del agente, particularmente adecuada al problema de **planificación reactiva** puesto que puede ser actualizada en cualquier momento. Esa nueva fuente de información está constituída, en principio, por el conjunto de abducibles o suposiciones adoptadas hasta el momento, que pueden ser usadas para justificar o prevenir otras suposiciones.

3.4.3. Cuál es la semántica de un Programa Lógico Abductivo?

Para definir la semántica de un programa lógico, uno necesita establecer que es lo que el procedimiento de prueba computa a partir del programa dada una consulta. Es decir, debemos decir en qué consiste una respuesta. Dado un programa lógica abductivo $< T, IC, Ab >$, una **respuesta** a una consulta Q es un par (Δ, θ) donde Δ es un conjunto finito de átomos abducibles básicos (sin variables) y θ es una sustitución de términos básicos para las variables en Q, que satisfacen: **respuesta**

$$T \cup Comp(\Delta) \models Q\theta \tag{3.5}$$

y

$$T \cup Comp(\Delta) \models Cond\theta \tag{3.6}$$

para cada implicación $Cond$ en IC. La expresión $Comp(\Delta)$ se refiere a la completitud de Clark [6] de Δ. Completar una teoría C en el sentido Clarkiano, significa técnicamente que se "refuerzan" todas las definiciones con si en C y se les convierte en definiciones si-y-solo-si, negando cualquier átomo no definido en C, y agregando la correspondiente teoría de igualdad de Clark que define las condiciones necesarias y suficientes para establecer la igualdad sintáctica entre los términos. Una consecuencia importante de la completitud deΔ es, en consecuencia, que los predicados abducibles que no ocurren en Δ se declaran como equivalentes a **falso**. La completitud de Clark es una forma común de **suposición del mundo cerrado** en la programación lógica. Fué diseñada para proveer de semántica (llamada **semántica de la negación por falla**) a los programas lógicos normales (programas lógicos con literales negativos en el cuerpo de las cláusulas).

Lo que se obtiene con la completitud es transformar un programa lógico con negación por falla en una teoría lógica de primer orden con negación clásica. Esto resuelve el problema semántico porque la teoría resultante puede ser entendida en términos de teorías de modelos, como es tradicional en lógica de primer orden. Esto ocurre sin problema en la mayoría de los casos. Sin embargo, existen algunos casos patológicos en los que la completitud causa problemas. Un ejemplo bien conocido es el programa $p \leftarrow \neg p$, que queda transformado por la completitud en $p \leftrightarrow \neg p$, una teoría obviamente inconsistente.

La semántica de la completitud tampoco permite "la posibilidad de que el valor de verdad de una consulta [..] sea **indefinido** debido a que el (probador) interpretador no alcanza a terminar"[59]. Para formalizar su nuevo procedimiento iff, Fung y Kowalski [34] usaron una semántica descrita por Kunen en [59] y usada también por Denecker y De Schreye en su formalización del procedimiento SLDNFA [19], [64, 20]. La semántica de Kunen es una forma refinada de la semántica de tres valores de Fitting [32] para programas lógicos y ambas están basadas en la idea anterior de Kleene, (que sugiriera primero en 1938 [51], [52]), y que consistía de usar una lógica de tres valores para explicar computaciones que no terminan. En su tesis doctoral [36], Fung probó que el procedimiento **iff** es sano y completo con respecto a la semántica de tres valores de Kunen (**verdad**, **falso** e **indefinido**). Así que el $\models$ en las fórmulas anteriores debe ser leído como verdad en todos los modelos de tres valores de ese tipo.

3.4.4. Planificación como Abducción

Construyendo sobre las conceptualizaciones anteriores, podemos postular que un agente es un planificador abductivo a partir de programas lógicos abductivos, que representan las creencias del agente, que sirven para reducir las metas del agente, llamadas también restricciones de integridad, a un conjunto de predicados abducibles, que representan las acciones que el agente debe ejecutar para alcanzar esas metas. Vamos a formalizar (axiomatizar) a continuación una especificación del cómo el agente ejecuta esas acciones y cómo esas ejecuciones están mezcladas con más planificación. Permítannos, antes, declarar aquí qué significa que el agente falle al ejecutar un plan (y, por tanto, falle en alcanzar una meta de esa manera). Puede ocurrir una de las siguientes:

1. Un D_i falla, es decir, al menos una acción en ese plan falló, para cada posible valor de sus variables. Esto significa que ese D_i nunca tuvo éxito.

2. Todos los D_i fallan, para cada posible combinación posible de valores de las variables.

3. Un D_i ha sido intentado un cierto número de veces, n y ha fallado en cada una (esto puede ser considerado como intentar $D_i\theta_j$, con $0 < j < n$, donde cada θ_j es una sustitución diferente de la variables en la meta).

4. Todos los D_i han fallado, al menos, un cierto número de veces, n.

Observe que las primeras dos son casos especiales de las respectivas dos últimas, con $n \rightarrow$ ínf. Puesto que infinito es un valor tan inconveniente para un agente (especialmente uno con recursos limitados), uno podría decir que este agente tiene problemas con la correspondiente ejecución de sus planes, cuando quiera que n es suficientemente grande. Un **agente perfeccionista** tratará de revisar las definiciones que conducen a planes fallidos si se da el caso 3 y $n = 1$. Un **agente paciente** (o perezoso) se permitirá escoger la opción 4 y valores más grandes para n. No obstante, para que tenga sentido toda esa discusión sobre revisión de las teorías del agente, tenemos que abordar el problema de definir qué significa que un agente aprenda.

agente perfeccionista

agente paciente

3.5. Una Breve Historia del Aprendizaje Computacional

El aprendizaje en IA parece tener una historia incluso más larga que la de planificación. Así que, en lugar de intentar una revisión bibliográfica completa, permítannos comenzar con un planteamiento formal del problema de revisar una teoría, propuesto por Luc DeRaedt [18]:

Dados

1. Un conjunto de ejemplos positivos y negativos E (podemos usar $E+$ y $E-$ respectivamente para referirles por separado).
2. Un lenguaje L (L_h es el lenguaje básico, sin variables).
3. Una teoría inicial T
4. Una relación de cobertura c (básicamente un procedimiento para probar cobertura).
5. posiblemente, una teoría de soporte B (con la forma de un conjunto de cláusulas definidas)

Encuentre un conjunto de cláusulas $T' \subset L$ tal que T' (posiblemente en conjunción con B) cubre todo los ejemplos positivos y ninguno de los negativos y es tan cercana a T como resulte posible.

Desde la perspectiva del agente planificador, y como primera aproximación, uno podría decir que:

1. $E-$ es el conjunto de metas que el agente falla en alcanzar. Puede ser una sola.
2. $E+$ es el conjunto de instancias de metas que han sido exitosamente alcanzadas.
3. L está constituído por todas las cláusulas cuya cabeza tiene el mismo predicado de una meta fallida y sus cuerpos corresponden a las D_i en la definición de ese predicado. Note que una definición es, básicamente, un conjunto de cláusulas que definen un predicado.

4. La relación de cobertura podría ser provista por el mismo procedimiento de prueba que se use para derivar sub-metas a partir de las metas para producir el plan, salvo que al momento de chequear la cobertura no se tiene que generar un plan, sino sólo "verificar" uno. Esto es, de hecho, posible. Por simplicidad, sin embargo, en la implentación actual en Gloria/Galatea mantenemos dos procedimientos separados.

5. La teoría de soporte podría consistir de las otras creencias y definiciones en la base de conocimiento del agente, no asociadas a las metas que fallan.

DeRaedt ofrece también una revisión de los algoritmos que se conocen para revisar teorías en este sentido. Uno puede usar un algoritmo particular y específico prácticamente para cada teoría de aprendizaje. Sin embargo, hay dos conceptos fundamentales que aparecen en todas las teorías de aprendizaje basado en lógica: los operadores de generalización y de especialización.

Siguiendo a DeRaedt, el operador de generalización de una teoría$\gamma_{\rho,g}$, que se apoya en el operador de refinamiento generalizador para una cláusula ρ_g se define así:

$$\gamma_{\rho,g}(T) = \{T - \{c\} \cup \{c'\} | c' \in \rho_g(c) \ \wedge \ c \in T\} \bigcup \{T \cup \{c\} | c \in L_h\} \qquad (3.7)$$

En forma análoga, el operador de especialización de una teoría $\gamma_{\rho,s}$, que se apoya en el operador de especialización de una cláusula ρ_s se define así:

$$\gamma_{\rho,g}(T) = \{T - \{c\} \cup \{c'\} | c' \in \rho_g(c) \ \wedge \ c \in T\} \bigcup \{T \cup \{c\} | c \in L_h\} \qquad (3.8)$$

Contamos ya en Galatea (ver archivo `flach.pl` en el paquete `galatea.glorias.gloria` en el repositorio) con un prototipo que implementa ambos operadores, basado en la implementación elemental del algoritmo MIS de aprendizaje de Shapiro, discutida por Peter Flach en [33]. Una primera aproximación para acomodar el proceso de aprendizaje en la arquitectura del agente planificador se discute en la siguiente sección.

3.6. Una especificación elemental de un Agente Aprendiz

A continuación presentamos la especificación original de **GLORIA**[11], que se muestra en la tabla 3.1, y que servirá como el punto de partida para la nueva especificación del aprendiz. Esta especificación se puede interpretar así:

CYCLE Las estructuras mentales del agente están cíclicamente sometidas a los efectos de dos procesos básicos que interactúan entre ellos. El predicado

[11] Por cierto, Gloria es un acrónimo: ***G**eneral-purpose, **L**ogic-based, **O**pen, **R**eactive and **I**ntelligent **A**gent.*

$$
\begin{array}{lr}
cycle(KB, Goals, T) & . \\
\quad \leftarrow\ demo(KB, Goals, Goals', R) & . \\
\quad \wedge\ R\ \leq n & . \\
\quad \wedge\ act(KB, Goals', Goals'', T+R) & . \\
\quad \wedge\ cycle(KB, Goals'', T+R+1) & [\mathbf{CYCLE}] \\
 & . \\
act(KB, Goals, Goals', T_a) & . \\
\quad \leftarrow\ Goals \equiv\ PreferredPlan\ \vee\ AltGoals & . \\
\quad \wedge\ executables(PreferredPlan, T_a, TheseActions) & . \\
\quad \wedge\ try(TheseActions, T_a, Feedback) & . \\
\quad \wedge\ assimilate(Feedback, Goals, Goals') & [ACT] \\
 & . \\
executables(Intentions, T_a, NextActs) & . \\
\quad \leftarrow\ \forall A, T(\ do(A,T)\ is\_in\ Intentions & . \\
\qquad \wedge\ consistent((T = T_a) \wedge Intentions) & . \\
\qquad \leftrightarrow\ do(A, T_a)\ is\_in\ NextActs\ ) & [EXEC] \\
 & . \\
assimilate(Inputs, InGoals, OutGoals) & . \\
\quad \leftarrow\ \forall A, T, T'(\ action(A, T, succeed)\ is\_in\ Inputs & . \\
\qquad \wedge\ do(A, T')\ is\_in\ InGoals & . \\
\qquad \rightarrow\ do(A, T)\ is\_in\ NGoal\ ) & . \\
\quad \wedge\ \forall A, T, T'(\ action(A, T, fails)\ is\_in\ Inputs & . \\
\qquad \wedge\ do(A, T')\ is\_in\ InGoals & . \\
\qquad \rightarrow\ (\mathbf{false}\ \leftarrow\ do(A, T))\ is\_in\ NGoal\ ) & . \\
\quad \wedge\ \forall P, T(\ obs(P, T)\ is\_in\ Inputs & . \\
\qquad \rightarrow\ obs(P, T)\ is\_in\ NGoal\ ) & . \\
\quad \wedge\ \forall Atom(\ Atom\ is\_in\ NGoal & . \\
\qquad \rightarrow\ Atom\ is\_in\ Inputs & . \\
\quad \wedge\ OutGoals\ \equiv\ NGoal\ \wedge\ InGoals & [ASSIM] \\
 & . \\
A\ is\_in\ B\ \leftarrow\ B\ \equiv A\ \wedge\ Rest & . \\
 & . \\
try(Output, T, Feedback) \leftarrow\ tested\ by\ the\ environment.. & [\mathbf{TRY}]
\end{array}
$$

Cuadro 3.1: El predicado *cycle* de Gloria

demo (que se explica a continuación) que reduce metas a submetas y activa nueva metas, y *act* que "consulta al ambiente". El agente le envía un envía un mensaje con las acciones a ejecutar y el ambiente responde con un mensaje que contiene las observaciones que corresponden a ese momento (incluyendo la retroalimentación sobre el resultado de la acción). Observe que los mensajes desde el agente pueden incluir acciones "observacionales" que le indican a los sensores que perciban de cierta manera. Así que este modelo de agente también contempla un agente que puede enfocar sus sensores. Desde luego, lo que el agente termina observando depende de las propiedades del ambiente en ese momento de interacción. Es también importante aclarar que esta versión del ciclo es una reducción de la propuesta original de Kowalski que, no obstante, no pierde nada de generalidad. Sigue siendo la secuencia de observación-pensamiento-acción que caracteriza al foco de control del agente.

ACT Este proceso transforma las estructuras mentales del agente en cierto momento si, en su "plan preferido"el agente selecciona aquellas acciones que pueden ser ejecutadas en ese momento, las envía (simultáneamente) al ambiente y analiza y asimila la retroalimentación obtenida desde el ambiente. El plan preferido por el agente es aquel sobre el cual el agente está concentrando su atención, es decir, el primero en el ordenamiento (heurístico) de todas sus metas.

EXEC Estas son todas las acciones en el plan preferido por el agente cuya tiempo de ejecución puede ser el momento actual. Note que no se trata de un simple test de igualdad. Puede tratarse de una instanciación del valor de la variable que indica el tiempo de acción, pues puede ocurrir que las acciones en el plan no tengan un tiempo específico predefinido para la ejecución. Así que el test de consistencia verificará las restricciones sobre esas variables para establecer si pueden ser ejecutadas sin problemas en el tiempo T_a. Esto, desde luego, requiere un esfuerzo de cómputo considerable que algunos "ejecutores" podrían eximir confiando en el ambiente como la mejor vía para verificar si la acción de hecho podía realizarse en ese momento.

ASSIM . Este tipo de asimilación se refiere al procesamiento de una retroalimentación más "informativa" desde el ambiente que indique, no solamente si las acciones fallan o tienen éxito, sino que incorpore nuevas observaciones para el agente.

Una primera aproximación a una gloria aprendiz podría ser como se muestra en la tabla 3.2. Lo importante es tener presente que el predicado *cycle* ha cambiado y que ahora requerimos una definición apropiada del predicado *learn*. Los otros predicados también se adaptan un tanto, como se explica a continuación.

Un detalle sumamente importante en esta especificación es que el predicado learn, tal como ocurrió antes con demo, representa un procedimiento acotado.

$$
\begin{array}{ll}
cycle(KB, Goals, T) & . \\
\quad \leftarrow\ demo(KB, Goals, Goals', R) & . \\
\quad \wedge\ R\ \leq n & . \\
\quad \wedge\ act(KB, Goals', Goals'', T+R) & . \\
\quad \wedge\ L\ \leq m & . \\
\quad \wedge\ learn(KB, Goals', KB', Goals'', L) & . \\
\quad \wedge\ cycle(KB', Goals'', T+R+L+1) & [\mathbf{CYCLE}] \\
 & .
\end{array}
$$

Cuadro 3.2: El predicado *cycle* de una Gloria Aprendiz

El valor m es el límite máximo para L, el número de pasos de cómputo dedicados a aprender en cada paso del ciclo. Haciendo a m suficientemente grande, podríamos permitir que cualquier algoritmo de aprendizaje se incorporara al agente, sacrificando probablemente, su capacidad reactiva y su apertura al ambiente. Por esta razón, en la implementación recurrirmos a la versión acotada del MIS que implementó Flach [33] y la adaptamos para que sirve justamente ese propósito, acotando además, de otras formas, el espacio de búsqueda del aprendiz.

Para ver como funciona, debemos antes definir cómo conectar el proceso reentrante de planificación del agente con cada problema de aprendizaje.

3.7. De la Planificación al Aprendizaje

Imaginemos una situación en la que act trata de ejecutar un plan producido por demo y encuentra, en la retroalimentación, que una o más acciones en ese plan ha fallado. Más aún, imaginemos que act incorpora una especice de "crítico" (criticón) que establece que esas fallan requieren revisar la definición correspondiente (a la meta involucrada) en la base de conocimiento (o creencias) del agente, KB.

El crítico divide la KB en la definición a ser revisada, Def y el resto de la KB original, que llamams KBr. El crítico podría también seleccionar todas (o una buena cantidad de) las instancias exitosas de la meta que falla, para crear $E+$, el conjunto de ejemplos positivos. Así mismo, el crítico usaría la instancia fallida de la meta para crear un conjunto de un sólo elemento $E-$, con el ejemplo negativo.

La teoría inicial a ser revisada, T, sería el conjunto de cláusulas con la definición Def (este es un conjunto perfectamente definido, a partir de la KB original, que serviría también para definir el lenguaje L). El subsistema de aprendizaje podría entonces usar como teoría de soporte B a KBr, haciendo $B = KBr$. Con esto tendríamos el problema de aprendizaje correspondiente a una meta fallida y todos los elementos de cómputo llegado el momento de revisar la base de conocimientos del agentes. Así ha sido implementado en Galatea.

3.8. Del Aprendizaje a la Planificación

Suponiendo que la tarea de aprendizaje se completa en un tiempo finito (y no muy grande), el subsistema de aprendizaje (definido por el predicado *learn*) devolverá una nueva versión del Def de la sección anterior. Llamemos a este Def'. El agente podrá continuar con el ciclo, haciendo $KB' = KBr + Def'$ y preservando las mismas metas pendientes por evaluar como estaban antes del aprendizaje. Note, sin embargo, que preservar las mismas metas y su historia previa de logros, asume que el agente está aprendiendo simplemente un mejor refinado conjunto de reglas que modelan su ambiente. Si el ambiente cambia sus reglas, el agente tendría que olvidar toda la historia previa (al menos para propósitos de aprendizaje) y podría verse forzado a una revisión total de sus planes pendientes por ejecutar (para deshacerse de aquellas metas y acciones que fueron deducidas con las viejas cláusulas). Nos preguntamos, incluso, si cierta medida de estabilidad del ambiente no será indispensable para permitir que estos ejercicios de aprendizaje superficial sean factibles. Esto requiere aún más investigación.

3.9. Combinando el aprendizaje con otras actividades mentales

El agente aprendiz elemental descrito en las secciones anteriores ya representa una forma de entrelazar el aprendizaje, el actuar y el razonamiento para planificar. Esta solución, insistimos, sólo puede ser apropiada para un agente real si el proceso o subsistema de aprendizaje completa sus tareas en un tiempo razonable (un número finito y no muy grande de pasos de razonamiento, por ejemplo). Es en este sentido básico que decimos que esta nueva arquitectura es un **agente aprendiz superficial** : un agente que será capaz de adaptar su conocimiento para seguir funcionando correctamente en su ambiente, siempre que las tareas de aprendizaje sean suficientemente acotadas (presumiblemente superficiales) para ser tratables en los términos computacionales que se han explicado antes.

agente aprendiz superficial

Una solución más flexible, pero también más compleja, podría ser un proceso de aprendizaje re-entrante, tal como se hace con el proceso de planificación que implementa el predicado demo. Pero para esto, el agente requirirá de un algoritmo de aprendizaje progresivo, que pueda suspenderse cuando quiera que se le acaba el tiempo al agente, produciendo resultados parciales. Esos resultados parciales son ciertamente posibles para la programación lógica inductiva. Con un motor inductivo de-arriba-hacia-abajo (*top-down*), los subprocesos de generalización y especialización están entrelazados. Si ocurriera una terminación prematura, el algoritmo debería arrojar una solución intermedia que, por ejemplo, sólo cubra algunos ejemplos positivos o permita algunos negativos. Una solución imperfecta que, no obstante, debería ser mejor que la regla inicial, en ciertas condiciones del espacio y del método de búsqueda. Qué se puede hacer con las metas previas o con tareas de aprendizaje simultáneas es algo que

Algorithm 15 El agente que aprende a construir un arco

```
1 arch(A, T) :- col(C1, C2, T1), leq(T1, T), beam(B, T).
2 col(C1, C2, T) :- c(C1, T), c(C2, T), do(C1,C2,T).
3 beam(B, T) :- b(B,T), dob(B, T).
4
5 abd(do). abd(dob). abd(c). abd(b). abd(leq).
6
7 observable(c(,)). observable(b(,)).
```

requiere todavía de más investigación.

En la siguiente sección, mostramos como la arquitectura elemental que se ha definido e implementado en Galatea, puede ser usada para simular ciertos tipos de agentes que se adaptan a las condiciones de su ambiente.

3.10. Aprendiendo, paso a paso, en un mundo de bloques

Gracias a que todos los componentes se pueden definir con precisión, los mundos de bloques son un dominio muy popular para probar las teorías y dispositivos en IA. A continuación les mostramos un ejemplo de planificación en un mundo de bloques, que es una variación del problema del arco usado en el sistema MARVIN y reportado por Muggleton y Buntine en el mismo artículo el que se presenta al sistema de aprendizaje CIGOL[?]

El programa abductivo, en notación Prolog, que usa el planificador se puede ver en el código 15

Se trata, repetimos, de cláusulas Prolog cuyos significados pretendidos se pueden referir así en Español:

Se construye una arco A en el instante T si las columnas C1 y C2 se construyen en el tiempo T1 y T1 es antes o al mismo tiempo que T y la viga B se coloca en su lugar en el instante T.

Las columnas C1 y C2 se construyen en el tiempo T si la C1 es una columna en el tiempo T y C2 es una columna en el tiempo T y el agente coloca ambas (C1 y C2) en el sitio apropiado en el instante T.

La viga B se construye en T si hay un objeto viga B en T y el agente coloca esa viga en T.

Por simplicidad, sin embargo, nos permitimos usar una versión más sencilla en 16. El tiempo es, desde luego, fundamental en planificación. Sin embargo, con este ejemplo de juguete podamos usar un programa igualmente expresivo en el que se deja el tiempo implícito en el orden de la representación.

Como advertimos antes, el predicado `abd` es un metapredicado que indica que `do`, `dob`, `c` y `b` son todos predicados abducibles en este programa lógico abducible. El predicado `observable` es otro metapredicado para indicar al razonador que esos predicados (`c` y `b`) no deben ser abducidos sino recibidos como observaciones desde el agente, cuando aplique. El resto del código constituye

Algorithm 16 El agente que aprende a construir un arco, sin tiempo explícito

```
arch(A) :- col(C1, C2), beam(B).
col(C1,C2) :- c(C1), c(C2), do(C1), do(C2).
beam(B) :- b(B), dob(B).
abd(do). abd(dob). abd(c). abd(b).
observables(c(_)). observables(b(_)).
```

la base de conocimiento inicial, KB, del agente. Digamos ahora que, de alguna manera (hay varias) este agente adquiere la meta: *se construye un arch a*, que puede ser codificada así: `arch(A).`

En Galatea, que integra al agente Gloria, contamos con varias maneras de invocar al motor de inferencia. Por simplicidad y como un auxilio a los lectores que quieran replicar este ejercicio con un sistema Prolog, digamos que invocamos el predicado `demoarch` que hemos definido así:

```
demoarch(R, G) :-
  demo_gloria(R, [[(c(5), c(8), b(10), true),
                  (p :: (arch(A), true), true),
                   true, [], []] ], G ).
```

El primer atributo del predicado auxiliar demo_gloria es una entero R que cuenta el número de pasos de razonamiento que se permitirán al sistema. El segundo atributo es mucho más complejo. Es una lista de metas, cada una de las cuáles es también un posible plan. En nuestro ejemplo, el único plan contiene un conjunto de átomos que representan lo que el agente observa `(c(5), c(8), b(10), true)` y la meta pendiente por procesar `(arch(A), true)`. Esta es la salida que se obtiene con el Prolog:

```
?- demoarch(50,G).
G = [[ (c(5), c(8), b(10), do(5), do(5), dob(10), true),...
true


?- demo_arch(50, G), write_frontier(G).
Abducibles set[1] = {
-  c(5, tf),
-  c(8, tf),
-  b(10, tf),
-  do(5, 5, tf),
-  leq(tf, tf),
-  do_b(10, tf),  };
Achievement Goals[1] = {  };
Maintenance Goals[1] = {  };
HF[1] = { [] }

Abducibles set[2] = {
-  c(5, tf),
-  c(8, tf),
-  b(10, tf),
-  do(5, 5, tf),
```

```
-  leq(tf, tf),  };
Achievement Goals[2] = {
-  t:: (b(_G9436, tf), true),
-  p:: (do_b(_G9436, tf), true),  };
Maintenance Goals[2] = {  };
HF[2] = { [_G9436 eq 10] }

Abducibles set[3] = {
-  c(5, tf),
-  c(8, tf),
-  b(10, tf),  };
Achievement Goals[3] = {
-  t:: (c(_G2705, tf), true),
-  p:: (do(5, _G2705, tf), leq(tf, tf), beam(_G712, tf), true),  };
Maintenance Goals[3] = {  };
 HF[3] = { [_G2705 eq 5] }

Abducibles set[4] = {
-  c(5, tf),
-  c(8, tf),
-  b(10, tf),  };
Achievement Goals[4] = {
-  t:: (c(_G2704, _G2706), true),
-  p:: (c(_G2705, _G2706), do(_G2704, _G2705, _G2706), ...
Maintenance Goals[4] = {  };
HF[4] = { [_G2704 eq 5, _G2706 eq tf] }

G = [[ (c(5, tf), c(8, tf), b(10, tf), do(5, 5, tf), leq(tf, tf),...
```

La meta original ha sido reducida a cuatro planes para alcanzarla. Cada uno de los planes contiene su propio conjunto de abducibles, con las observaciones (que son dadas como entradas) y, lo más importante, las acciones abducidas, que el agente debe ejecutar para alcanzar la meta. El plan también refiere cuáles son las metas pendientes, aún por resolver (las de logro, las metas incondicionales y las de mantenimiento que coinciden con las llamadas restricciones de integridad).

Por simplicidad, concentrémonos sólo en el primer plan. Dice que el agente observa las columnas 5 y 8 y la viga 10 y que debe colocar la columna 5 en los dos lugares para soportar a la viga 10, que debe ir en su sitio también. Por todo lo que sabemos sobre el mundo de los bloques, este agente está en un ambiente donde no es posible colocar el mismo objeto en dos posiciones al mismo tiempo. Sin embargo, este agente no lo sabe. Sus reglas no contempla esa posibilidad. Asi que está condenados a tratar la secuencia de acciones `do(5), do(5)` y fallar en la construcción del arco.

Supongamos ahora que este agente es uno de esos perfeccionistas (que definimos antes matemáticamente) y una vez que se entera de la falla (por la retroalimentación de su ambiente) trata de revisar su conocimiento para corregirlo. Para permitir eso, hemos programado una muy sencilla tarea de aprendizaje, usando la versión modificada del programa prolog implementado por Peter Flach [33], basado en el sistema Model Inference System, MIS, de E. Shapiro [83].

Comenzamos codificando la teoría de soporte apropiada:

```
1 obj(1). obj(2). obj(3). obj(4). obj(5). obj(6). obj(7). obj(8).
2 bg((neq(X,Y) :- true)) :- obj(X), obj(Y), not(X=Y).
```

```
bg((c(1) :- true)). bg((c(2) :- true)). bg((c(4) :- true)).
bg((c(6) :- true)). bg((c(7) :- true)). bg((c(5) :- true)).
bg((c(8) :- true)). bg((do(1) :- true)).
bg((do(2) :- true)). bg((do(4) :- true)).
bg((do(6) :- true)). bg((do(7) :- true)).
bg((do(5) :- true)).
```

y las siguientes definiciones de tipos de datos:

```
literal(neq(X,Y), [obj(X), obj(Y)]).
literal(c(X), [obj(X)]).
literal(do(X), [obj(X)]).
literal(col(X,Y), [obj(X), obj(Y)]).
term(obj(X), [obj(X)]) :- obj(X).
```

El conjunto de ejemplos puede ser visto en la siguiente cláusula Prolog que hemos usado para invocar al aprendiz. Acá usamos sólo algunas instancias de la meta `col/2` y la última que se convierte, como dijimos, en el ejemplo negativo. Noten que hemos incluído el `do(5)` como un hecho para efectos del conocimiento de soporte. Es decir, se asume que la acción se realizó, pero falló en alcanzar la meta planteada.

```
learn(Clauses) :-
    inducespec([(col(X,Y):-c(X),c(Y),do(X),do(Y))],
                [ +col(1,2), +col(2,4), +col(6,7),
                  -col(5,5) ], Clauses).
```

Esta cláusula auxiliar también nos permite mostrar que el aprendiz no está comenzando de cero. Está usando la cláusula fallida como la teoría inicial, que se somete a revisión. De hecho, la cláusula fallida es usada, junto con la teoría de soporte, para demostrar que la teoría actual cubre, incorrectamente, el ejemplo negativo. Al final, el programa produce esta salida:

```
?- learn(C).
a((col(G234, G235):- c(G234), c(G235), do(G234), do(G235)), [obj(G234), obj(G235)])
...specialised into:
  col(G234, G235):- neq(G234, G234), c(G234), c(G235), do(G234), do(G235)
3..Found clause: col(G627, G628):-neq(G627, G628)
C = [ (col(G627, G628):-neq(G627, G628)),
 (col(G234, G235):- neq(G234, G234), c(G234), c(G235), do(G234), do(G235))]
true
```

En el caso con tiempo explicito, el resultado es similar:

```
?- learn(C).
3..Found clause: col(_G267, _G268, _G269):-true
a((col(_G267, _G268, _G269):-true), [obj(_G267), obj(_G268), time(_G269)])
    ...specialised into: col(_G267, _G268, _G269):-neq(_G267, _G268)
C = [ (col(_G267, _G268, _G269):-neq(_G267, _G268))]
true
```

En ambos casos, lo que se muestra es que, básicamente, el sistema está aprendiendo una versión refinada de la cláusula original pero que no implica a la meta fallida. Con este propósito, el programa ha capturado una distinción clave: los objetos que se usen para construir las dos columnas no pueden ser uno y el mismo.

Notará el lector, sin embargo, que la nueva cláusula que se ofrece como solución no tiene todos los elementos (los abducibles) necesarios para generar un plan. Este fue un problema de la implementación inicial que ha sido resuelto completamente en el sistema integrado en Galatea, tal como se muestra en el siguiente ejemplo final de aprendizaje multi-agente

3.11. Agentes que aprenden en una simulacion

Esa implementación elemental del MIS de Shapiro, hecha por Flach, ha servido para complementar el motor de inferencia de los agentes de Galatea y permitir esta forma controlada que llamamos aprendizaje superficial, en tiempo de simulación.

Completamos ahora nuestra contribución a la modeloteca GALATEA (en este texto) con un modelo en el que los agentes se cambian ellos mismos en un intento por adaptarse en y a su ambiente simulado. Nuestra implementación de la teoría de aprendizaje descrita en la sección 3.5 anterior es experimental todavía, pero ha sido suficiente para incorporarle a los agentes de Galatea un motor de aprendizaje superficial que les permite corregir su propio código. Al igual que los anteriores, el ejemplo es muy elemental, con la intención de exhibir las funcionalidades antes que el sistema modelado. Se trata de una variante del modelo de biocomplejidad, en el que los agentes colonos pujan por ser exitosos en una forma reducida de economía que se renta de la tierra.

3.11.1. El modelo de los agentes que aprenden a ser exitosos en la reserva forestal

Retomando las explicaciones de la sección 2.3.1, inspirados por la Reserva Forestal de Caparo, se trata de un sistema económico con un tipo de agente que se representa explícitamente: el colono (los demás se mantienen implícitos por simplicidad) que puede instalarse en el área apropiándose de un pedazo de tierra (*settle*), cultivar esa tierra (*plant*) y criar ganado (*cattle*). Esas son sus capacidades "teóricas". Sin embargo, sus creencias originales (su base de conocimiento inicial) sólo dan cuenta de su motivación por el éxito (con una sola regla de activación) y con una estrategia muy básica para lograrlo: instalarse (*settle*) y cultivar (*plant*). El colono original no sabe (no cree) que la ganadería es también una estrategia para el éxito. Mucho menos sabe que es la mejor estrategía en virtud de la configuración de su ambiente.

colono

Las condiciones económicas son provistas por el ambiente simulado en el que actúan los agentes. Las acciones de los agentes aumentan sus posesiones, bien en la forma de la tierra o en capital financiero ahorrado. La tierra que el agente pasa a poseer (gracias a *settle*) es descontada de la tierra disponible, así que se trata de un recurso escaso. El capital, por otro lado, no se entiende como recurso escaso sobre el supuesto de que todas esas acciones agregan algún valor, aunque en diversa medida. Es decir, ninguna acción produce pérdida sino relativa a las otras. Los "premios" proporcionados por cada acción son: el colono que se insta-

la por vez primera (settle), recibe una asignación de tierra y luego incrementos progresivos. El colono que cultiva (*plant*) recibe una cantidad de dinero proporcional a la tierra que posee. El colono que cría ganado (*cattle*) recibe también una cantidad proporcional a su terreno, pero mayor que la que recibe el cultivar. Las condiciones de simulación propician que al menos algunos agentes se vean expuestos al fracaso, con lo cual se dispara un proceso de aprendizaje en cada uno de ellos, que los lleva a corregir su conducta incorporando la más rentable acción de ganadería (*cattle*).

Este modelo simple de la economía forestal capitalista es regulado por los siguiente parámetros:

1. Cantidad de colonos
2. Factor de conversión de la tierra al capital al cultivar.
3. Factor de conversión de la tierra al capital con ganadería.
4. Umbral de acceso al éxito.
5. Tierra asignada al instalarse.
6. Tierra total disponible.
7. Tiempo de simulación.

Las salidas a evaluar son

1. Cantidad de colonos que aprenden a ser exitosos.
2. Capital total acumulado.

Hemos codificado un modelo de simulación con esos parámetros y variables de estado (ver el paquete `contrib.learners` en el repositorio Galatea) y lo hemos usado para correr algunos experimentos de simulación que se explican a continuación.

3.11.2. Experimento de simulación con agentes aprendices

El simulador de los agentes aprendices es muy parecido a los que se describen en el capítulo 2. Un programa principal, `Modelo`, una clases para describir al tipo `Colono`, una clase con el modelo del ambiente, `Geografía`, y la `Interfaz` que contiene los métodos que convierten las intenciones de los agentes (más que intenciones son influencias[31, 14, 17]) en cambios sobre el ambiente. Además de eso, las metas de mantenimiento y creencias del agente son codificadas en el componente Prolog (archivos .main y .kb) y todo es integrado automáticamente gracias a los servicios del sistema JPL integrado al software de SWI-Prolog (ver paquete `galatea.glorias` en el repositorio Galatea).

Al correr el modelo, el simulador despliega la traza Glider tradicional, pero también un vaciado de los cambios que ocurren en el estado interno de los

agentes y en la interfaz entre Java y Prolog. Así puede uno enterarse de que el agente está observando, razonando, modificando su base de conocimiento para registrar sus observaciones o, el caso más interesante, cambiando sus reglas de razonamiento luego de detectar una falla en alguna acción.

Por ejemplo, el siguiente es un registro de cuando el agente detecta una acción fallida:

```
# GInterface: executing plant
# GInterface: trying to execute public void
  contrib.learners.Interfaz.plant(double,contrib.learners.Colono,jpl.Integer)
# Interfaz: failed(plant(7),succeed(7))
```

Esa acción dispara inmediatamente un proceso de aprendizaje que culmina con una nueva regla de conducta que reemplaza una regla anterior, como se ilustra acá (el lector debe cuidar de no confundirse en la salida del simulador. El programa no necesariamente despliega esto en orden):

```
# Gloria: guardando settle(7):-true in agent ex_colono1
# Gloria: guardando plant(7):-true in agent ex_colono1
# Gloria: rules of agent ex_colono1
# Gloria: revising -> to succeed(_G965) do settle(_G965), plant(_G965).
# Learner ex_colono1: example succeed(7) is covered by
  [ (succeed(_G965):-settle(_G965), plant(_G965))]
# Learner: ex_colono1's␣example␣-succeed(7)␣processed
#␣Gloria:␣newly␣learnt␣rule␣->␣ex_colono1␣:
␣␣to␣succeed(_G965)␣do␣cattle(_G965),␣settle(_G965),␣plant(_G965).
#␣Gloria:␣guardando␣settle(7):-true␣in␣agent␣ex_colono1
#␣Gloria:␣guardando␣plant(7):-true␣in␣agent␣ex_colono1
```

Esta salida muestra como el agente, luego de fallar en la acción *plant* (que debería conducir a alcanzar la meta succeed), dispara una tarea de aprendizaje en la cual la regla:

para tener éxito, establézcase y plante

es reemplazada por

para tener éxito, introduzca ganado, establézcase y plante.

La acción de introducir ganado (*cattle*) tiene como efecto pecular un mayor rendimiento económico que permite que el agente supere un umbral predefinido y a partir del cual será exitoso, según se define.

Para transformar una regla en la otra, el agente recurre al pequeño motor de aprendizaje imbuído en Gloria. El código fue producido originalmente por Peter Flach en [?] y lo hemos modificado para ajustar una versión muy controlada y restringida de los operadores de generalización y especialización que se muestran en esas figuras (ver los archivos Prolog del paquete `galatea.glorias.gloria` en el repositorio Galatea).

El lector podrá confirmar que el algoritmo de generalización 17 no es realmente muy útil, pues simplemente comienza de cero la búsqueda. Esto difícilmente aplique como parte de la definición de aprendizaje superficial de la que hemos hablado. No quisimos, sin embargo, aumentar la complejidad del experimento con un algoritmo más sofisticado, por ahora. El método de especialización 18, por otro lado, hace justo el trabajo necesario, tomando la regla cuestionada como punto de partida de la búsqueda. Haciendo un ejercicio de especialización

Algorithm 17 Operador de Generalización en Gloria Aprendiz

```
% Generalisation by adding a covering clause
generalise(Ag, N, Cls,Done,Example,Clauses, NNN):-
    search_clause(Ag, N, Done,Example,Cl),
    writef("\n# Learner: Found the clause \n%  - %w\n\n", [Cl]),
    NN is N + 1,      % gv_node_trace(N,Cl,NN),
    process_examples(Ag, NN, [Cl|Cls],[],[+Example|Done],Clauses, NNN).

search_clause(Ag, N, Exs,Example,Clause):-
     literal(Head,Vars),
     try((Head=Example)),
     ss(AVars, Vars),
search_clause(Ag, N, 5,a((Head:-true),Vars),Exs,Example,Clause).
% base case
search_clause_d(Ag, NN, D,a(Clause,Vars),Exs,Example,Clause):-
       covers_ex(Ag, Clause,Example,Exs),!.
search_clause_d(Ag, N, D,Current,Exs,Example,Clause):-
       D>0,D1 is D-1, NN is N + 1,
     specialise_clause(Current,Spec),
     search_clause_d(Ag, NN, D1,Spec,Exs,Example,Clause).
```

mínimo bajo el concepto de absorción θ, definido en [18], es posible especializar una cláusula agregandole un literal o aplicándole una substitución, tal como prescribe el código que se muestra.

En las condiciones extremadamente controladas (cláusulas muy pequeñas, no recursivas, por ejemplo), estos códigos elementales son suficientes para generar la conducta de aprendiz de que hemos venido modelando y que conduce a los siguientes resultados.

3.11.3. Resultados de los experimentos de simulación

¿Cómo saber que esta forma de aprendizaje corresponde a un mecanismo de adaptación para mejorar la conducta del agente?. La definición matemática de aprendizaje que se establece en la sección 3.5 no es adecuada para una comparación a este nivel, pues no contempla una comparación entre mejores y peores conductas. Así que, para responder a esta pregunta, proponemos ahora una correspondencia entre el modelo de simulación de esa economía reducida de una reserva forestal y un problema elemental de optimización.

Supongamos, por simplicidad, que tenemos sólo dos agentes. El "problema global" en la reserva forestal podría plantearse como un problema de optimización como el que se muestra en la figura 3.11.3, donde C_i es el capital acumulado por el colono i, T_i es la tierra a su cargo, T es toda la tierra disponible, K_p es el factor de rendimiento de la tierra en capital al cultivar (*plant*), K_c es el factor de rendimiento de la ganadería (*cattle*), U_i es el umbral en el que un factor de rendimiento es sustituído por el otro y, desde luego, $K_p < K_c$. Una simplificación adicional es posible, incluso deseable, sobre el umbral de transición de rendimiento. Puede uno suponer que es el mismo para todos los agentes

Algorithm 18 Operador de Especialización en Gloria Aprendiz

```
specialise(Ag, N, Cls,Done,Example,Clauses, NNN):-
   false_clause(Ag, Cls,Done,Example,C),
   specialise_clause(C,a(S, _)),
   %       C = a(To_be_removed, _),
   remove_one(To_be_removed, Cls, Cls1),
   NN is N + 1,
   process_examples(Ag, NN, [S|Cls1],[],[-Example|Done],Clauses, NNN).

% specialise_clause(C,S) <- S is a minimal specialisation
%                           of C under theta-subsumption
specialise_clause(Current,Spec):-
   add_literal(Current,Spec).
specialise_clause(Current,Spec):-
   apply_subs(Current,Spec).

% less restrictive than the original predicate
add_literal(a((H:-true),Vars),a((H:-L),Vars)):-!,
   literal(L,LVars),
   not(would_unify(L, H)), % preventing recursive clauses
 ss(LVars, Vars). % using a new predicate

add_literal(a((H:-B),Vars),a((H:-L,B),Vars)):-
   literal(L,LVars),
   not_too_big((L,B)), % Problem-specific condition
   not(repeated(L,B)), % another problem-specific condition
   not(would_unify(L, H)), % preventing recursive clauses
   ss(LVars, Vars).

% modified to support strict specialisation
apply_subs(a(Clause,Vars),a(Spec,SVars)):-
  copy_term(a(Clause,Vars),a(Spec,Vs)),
  apply_subs1(Vs,SVars),
  % writef("applying a subs %w to %w \n", [Vs,Clause]),
  not(just_the_same(Clause, Spec)). % strict specialisation
```

maximizar $C_1 + C_2$
sujeto a que
$C_1 = K_p * T_1$ si $C_1 < U_1$
$C_1 = K_c * T_1$ si $C_1 \geqslant U_1$
$C_2 = K_p * T_2$ si $C_2 < U_2$
$C_2 = K_c * T_2$ si $C_2 \geqq U_2$
$T_1 + T_2 \leqq T$
$T_1 \geqq T_1^0$
$T_2 \geqq T_2^0$

Figura 3.1: El problema de optimización con dos agentes

($U_1 = U_2 = U$), asumiendo que el sistema le dará las mismas oportunidades a cualquier colono de cambiar su economía de siembra a ganado. No hay que olvidar, sin embargo, que incluso esto puede ser atrevido en ese sistema, puesto que depende básicamente de los precios de la ganadería que, como se sabe, varían en el tiempo y dependen de la red de relaciones comerciales que mantenga cada colono.

Como podrán anticipar los conocedores, esta forma del problema es suficientemente cercana a la estructura clásica de un problema de programación lineal multiobjetivos trivial. Gracias a esta cercanía uno puede decir, por ejemplo, que si los valores de las variables dependientes C_i se mantienen todos del mismo lado del umbral U, podemos decir que las soluciones óptimas corresponden a los puntos sobre la recta $T_1 + T_2 = T$, siempre claro que se cumplan también las restricciones de valor inicial (lo cual nunca es un problema en este contexto, pues suponemos que los agentes no pierden capital y este siempre se puede convertir en tierra). Más aún, esto siempre ocurrirá si toda la tierra disponible, T, es mayor a cierto valor dependiente del umbral U, ($T > 2U/K_c$en el caso de dos agentes), puesto que todos los colonos podrán alcanzar el correspondiente valor máximo de capital.

Las dificultades podrían surgir cuando $T \leqq 2U/K_c$pues entonces el sistema sólo alcanzará el máximo capital acumulable si la tierra se otorga a quienes la van a "aprovechar" mejor. Suponiendo un mecanismo de búsqueda completo y no sesgado (como el método simplex), además de las suposiciones previas (el mismo U para todos), esas dificultades no surgen. Resta por comprobar si la "búsqueda" que supone el proceso de simulación es completa y no sesgada, como para abrogarse estas mismas ventajas.

En términos prácticos, podemos comparar la salida producida por un optimizador con la producida por el simulador. Aprovechando la implementación del algoritmo simplex incorporada al Swi-Prolog, hemos codificado una aproximación al problema como se muestra en 3.11.3.

Este esqueleto lineal del problema nos permite fijar los valores de K, T, mientras se establece que los valores iniciales de las variables de optimización están fijos también así $T_1^0 = 10$, $T_2^0 = 10$ y se fija, independientemente (por fuera) del modelo lineal, el valor de U. Es importante notar que `K1` y `K2` tomarán valores que correspondan a K_c o K_p luego de la decisión respecto a U. De esta manera, podemos referir los resultados de la simulación en una tabla comparativa (ver tabla 3.11.3) con los resultados del modelo teórico de optimización.

La tabla 3.11.3 muestra las salidas producidas por la aproximación teórica lineal (T_{1teo}, T_{2teo} y $Capital_{teo}$) junto a los valores de las mismas variables generados por el simulador con los agentes aprendices (T_{1sim}, T_{2sim} y $Capital_{sim}$). La aproximación teórica en algunas casos se asocia a dos salidas posibles (e.g. T_{1teo}= 9 ó 1, en la fila 1), puesto que el valor depende de cual de los agentes se asume que transciende el umbral U primero y, en consecuencia, aprovecha el valor de K_c para escalar su aporte a la función objetivo. Es decir, allí se puede apreciar que el modelo teórico es apenas una aproximación que no da cuenta de una dinámica que introducen las restricciones condicionales como $C_1 = K_p * T_1$ si $C_1 < U_1$en la figura 3.11.3. El analista debe hacer un ajuste

T	U	T_1^0	T_2^0	K_c	$2U/K_c$	T_{1teo}	T_{2teo}	$Capital_{teo}$	T_{1sim}	T_{2sim}	$Capital_{sim}$	
10	50	1	1	10	10	9 ó 1	1 ó 9	91	5	5	100	1
10	50	1	5	10	10	1 ó 5	9 ó 5	91 ó 55	3	7	73	2
10	50	5	1	10	10	9 ó 5	1 ó 5	91 ó 55	7	3	73	3
10	50	9	1	10	10	9	1	19 ó 91	9	1	91	4
10	50	1	9	10	10	1	9	91 ó 19	1	9	91	5
100	50	1	1	2	50	1 ó 99	99 ó 1	199	49	49	196	6
100	50	1	5	2	50	1 ó 95	99 ó 5	199 ó 195	46	53	198	7
100	50	5	1	2	50	5 ó 99	95 ó 1	195 ó 199	53	46	198	8
100	50	9	1	2	50	10 ó 99	90 ó 1	190 ó 199	57	43	200	9
100	50	1	9	2	50	1 ó 90	99 ó 10	199 ó 190	43	57	200	10
100	50	10	10	2	50	10 ó 90	90 ó 10	190	49	49	196	11
100	50	20	20	2	50	20 ó 80	80 ó 20	180	50	50	200	12
100	50	50	50	2	50	50	50	150	50	50	200	13

Cuadro **3.3**: Comparación del modelo de simulación y una aproximación teórica óptima

Algorithm 19 La aproximación en programación lineal al optimizador con dos agentes

```
1  :- use_module(library(simplex)).
2
3  restricciones(T10, T20, T) -->
4          constraint([t1, t2] =< T),
5          constraint([t1] >= T10),
6          constraint([t2] >= T20).
7
8  solucion(T10, T20, K1, K2, T, C1, C2, Val1, Val2, Cap) :-
9          gen_state(S0),
10         restricciones(T10, T20, T, S0, S1),
11         maximize([K1*t1, K2*t2], S1, S),
12         variable_value(S, t1, Val1),
13         variable_value(S, t2, Val2),
14         C1 is K1*Val1,
15         C2 is K2*Val2,
16         Cap is K1*Val1+K2*Val2.
17
18 % ?- solucion(1, 1, 1, 10, 10, C1, C2, T1, T2, Capital).
```

de los parámetros de entrada para compensar por esa incapacidad del modelo de programación lineal. El modelo de simulación, por otro lado, incorpora esa dinámica y, dados suficientes tiempo de simulación y un "paso" apropiado en la búsqueda de los valores apropiados de las variables independientes (T_{1sim} y T_{2sim}), calcula valores mejores (filas 1, 9, 11, 12 y 13), iguales (4 y 5) o posiblemente mejor (dependiendo de la referencia teórica que se use), y en todo caso aproximados a los del modelo teórico. Los resultados de la simulación, además, parecen menos sesgados (como se ve en la fila 1) entre los agentes, lo que sugiere que el proceso de búsqueda por simulación podría obtiene mejores respuestas desde la perspectiva local de los agentes, en algunos casos.

Es importante aclarar, sin embargo, que para alcanzar esos resultados el simulador requiere un tiempo de simulación suficientemente grande. Con tiempos menores, la búsqueda se detiene en valores subóptimos. El tiempo de simulación, además, puede crecer tanto que no pueda ser manejado por variables enteras (como las que usamos en el código Galatea de este modelo), por lo que se requiere atender a otro parámetro importante: el paso. Por paso nos referimos a los valores con los que se incrementa (en este caso, sólo se incrementa, pero se podría hacer de otra manera) el valor de cada variable independiente durante la simulación. El código de `Interfaz.java` muestra como cada vez que el agente ejecuta la acción *plant*, el terreno que tiene asignado crece en cierta medida. Esa medida es el paso al que nos referimos y para valores no tan grandes de T y U, puede ser tan pequeño como 1. Pero para $T = 100$, debimos aumentarlo hasta 3, de manera que los tiempos de simulación fuesen manejables.

La tabla 3.11.3 ofrece, desde luego, una perspectiva muy limitada. Pero que-

maximizar $\sum_i C_i$
sujeto a que
$\forall_i(C_i = K_p * T_i$ si $C_i < U_i$ y $C_i = K_c * T_i$ si $C_i \geqslant U_i)$
$\sum_i T_i \leqq T$
$\forall_i(T_i \geqq T_i^0)$

Figura 3.2: El problema genérico de optimización que resuelven los agentes aprendices

da claro que la simulación con ese modelo de agentes sirve al ejercicio de optimización (es decir, para mejorar la conducta de todo el sistema) por medio de la capacidad que tienen los agentes de corregir su conducta aprendiendo nuevas reglas durante la simulación. Nos atrevemos a especular que estamos frente a un nuevo y más expresivo marco para problemas de optimización que se pueden representar con la forma genérica que se muestra en la figura

Capítulo 4

Conclusiones

Este libro se construye en la intercepción de tres líneas de investigación sobre agentes inteligentes: la lógica, la inteligencia artificial y la simulación. El capítulo 1, ha pretendido conectar al lector con una práctica asociada a la teoría de agentes y toda la obra del Prof. Robert Kowalski. La lección más importante de ese capítulo es una extensión simple de la gran lección de la Programación Lógica que el mismo Kowalski sentara como una de las bases de la computación moderna: la lógica puede servir como lenguaje de modelado de agentes, tal como ha servido como lenguaje de programación de computadores. Distanciándonos un tanto del rigor tradicional que se usa en la matemática introductoria de la programación lógica, nos hemos concentrado en ilustrar, con ejemplos seleccionados, las posibilidades expresivas y la flexibilidad del lenguaje. El capítulo 2 tiende un puente hacia la simulación, ese otro paradigma computacional que depende crucialmente de la posibilidad de representar conocimiento para el computador. Sin modelos no hay simulación. La lección en ese capítulo es que es posible incorporar la conceptualización de los agentes, de diversas maneras, como parte de modelos de simulación no triviales. Galatea es nuestra contribución colectiva en esa dirección y, hemos tratado de mostrar, ofrece una serie de servicios de investigación que aún debemos explorar. El último capítulo 3 es una incursión atrevida en el complejo mundo del aprendizaje de máquina o aprendizaje computacional, como decidimos llamarlo. Es atrevida porque para poder integrar en el contexto de la simulación y de los sistemas multi-agentes, hemos debido adoptar un noción muy débil: el aprendizaje superficial, que sólo se puede considerar efectivo si el agente se enfrenta al problema de aprendizaje con un conocimiento a corregir que no esté tan distante de la forma correcta. Con esa suposición de trabajo, es posible implementar un motor de aprendizaje con recursos acotados, es decir, que pueda detenerse aún antes de completar la revisión de espacio de búsqueda. De hecho, ese fue el diseño original de Shapiro para el MIS que, gracias a la gentileza del prof. Peter Flach al compartir esa pieza de código, es ahora parte de los agentes en Galatea.

El libro cumplirá sus propósito si sirve para guiar a sus lectores en la exploración de posibilidades para el uso de la lógica en el modelado de agentes

inteligentes y en la realización de experimentos de simulación asociados a esa exploración.

Apéndice A

De la mano de Galatea para aprender Java

Este apéndice comienza con una revisión, estilo tutorial, de los conceptos básicos subyacentes al desarrollo de software orientado a los objetos (OO) en el lenguaje de programación Java.

No se trata, simplemente, de pagarle servicio al lenguaje Java que estamos usando en casi todo este documento para modelar sistemas que simularemos en la plataforma Galatea. Lo que queremos hacer va un poco más allá. Los primeros esfuerzos en simulación con el computador estuvieron íntimamente ligados con esfuerzos en computación dirigidos a desarrollar lenguajes de programación más expresivos. La programación orientada a objetos surgió de esos esfuerzos y como respuesta a la urgencia de los simulistas por contar con herramientas lingüístico-computacionales para describir los sistemas que pretendían modelar y simular. Java es la culminación de una revolución que comenzó con el primer lenguaje de simulación, SIMULA [89]: la orientación a los objetos, OO.

Nuestra intención con Galatea es repatriar la expresividad de la orientación por objetos, junto con la enorme cantidad de mejoras de Java, para su uso, ya no en simulación tradicional, sino en simulación orientada a los AGENTES, OA. Este tutorial Java ha sido dispuesto en dos partes: 1) Un recorrido por lo más elemental del lenguaje, incluyendo la instalación más económica que hemos podido imaginar, de manera que el simulista pueda comenzar a trabajar con los pocos recursos que pueda reunir; y 2) Un ejercicio guiado de desarrollo que termina con una aplicación Java, con el simulador incluido y con una interfaz gráfica. Las interfaces gráficas (a pesar de que contamos con un primer prototipo de un IDE [3]) es uno de los aspectos más débiles de Galatea. Así que queremos enfatizar, desde el principio, que toda ayuda será bien recibida.

A.1. Java desde Cero

Sin suponer mayor conocimiento previo, procedemos con los detalles de instalación.

A.1.1. Cómo instalar `java`

El software básico de Java[1] está disponible en:

http://java.sun.com/javase/downloads/ (Abril 2009)

Al instalar el J2SE, un par de variables de ambiente serán creadas o actualizadas. En "path" se agregará el directorio donde fue montado el software y en "classpath" el directorio donde están los ejecutables `.class` originales del sistema Java. Ambas variables deben se actualizadas para Galatea como se indica a continuación.

A.1.2. Cómo instalar `galatea`

Galatea cuenta con una collección de clases de propósito general (no exclusivo para simulación), algunas de las cuáles aprovecharemos en este tutorial. Por ellos explicamos de inmediato como instalarla. Todo el software de Galatea viene empaquetado en el archivo galatea.jar. Para instalarlo, haga lo siguiente:

Configure el ambiente de ejecución así:
Unix, GNU/Linux y MacOS:

```
setenv $GALATEAHOME$ $HOME/galatea
setenv CLASSPATH $CLASSPATH:$GALATEAHOME:.
```

Windows:

```
set CLASSPATH = $CLASSPATH:galatea.jar
```

donde `/galatea` en Unix y `Galatea`, en Windows, son nombres para el directorio donde el usuario alojará sus programas. No tienen que llamarse así, desde luego. Coloque el archivo `galatea.jar` en este directorio (noten que aparece listado en el `CLASSPATH`).

A.1.3. Hola mundo empaquetado

Para verificar la instalación, puede usar el siguiente código, ubicándolo en el archivo `Ghola.java` en el directorio donde ha instalado Galatea que llamamos directorio de trabajo en lo sucesivo:

```
1 /** Ghola.java
2  ** Created on April 17, 2004, 12:57 PM */
3 package contrib.Ghola;
```

[1] En minúscula cuando nos refiramos al programa. En Mayúscula al lenguaje o a toda la plataforma.

```
4   import galatea.*;
5   /** * * @author El simulista desconocido * @version 1.0 */
6   public class Ghola {
7   /* @param argumentos, los argumentos de linea de comandos
        */
8     public static void main(String args[])  {
9        List unalista = new List(); unalista.add("Simulista")
             ;
10       System.out.println("Hola Mundo!, esta lista "+
             unalista);
11       System.out.println("contiene la palabra: "+unalista.
             getDat()); }}
```

Este es un codigo simple de una clase, `GHola`, con un solo método, main, que contiene una lista de instrucciones que la máquina virtual ejecutará cuando se le invoque. El programa debe antes ser "compilado" con la instrucción siguiente (suponiendo que estamos trabajando en ventana con el shell del sistema operativo, desde el directorio de trabajo):

```
javac Ghola.java
```

Si todo va bien, el sistema compilará en silencio creando un archivo `Ghola.class` en el subdirectorio `contrib` del subdirectorio `Ghola` del directorio de trabajo (verifiquen!).

Para correr[2] el programa use la instrucción:

```
java demos.Ghola.Ghola
```

Noten ese nombre compuesto por palabras y puntos. Ese es el nombre completo de su programa que incluye el paquete al que pertenece [3]. Si el directorio de Galatea está incluido en los caminos de ejecución (la variable PATH que mencionamos antes), podrá invocar su programa de la misma manera, no importa el directorio desde el que lo haga.

La salida del programa al correr será algo así:

```
Hola, esta lista List[1;1;/] -> Simulista
contiene la palabra: Simulista
```

Ud. ha corrido su primer programa en Java y en Galatea.

A.1.4. Inducción rápida a la orientación por objetos

Considere los siguientes dos problemas:

> EJEMPLO 1.0
>
> El problema de manejar la nomina de **empleado**s de una institución. Todos los empleados son personas con *nombre*s e *identificación*, quienes tienen asignado un *salario*. Algunos empleados son **gerente**s

[2] correr es nuestra forma de decir: ejecutar o activar el programa.

[3] Esa notacion corresponde también al directorio donde está el ejecutable. Revise *./demos/Ghola/Ghola* si en Unix y *.\demos\Ghola\Ghola* en Windows

quienes, además de los ATRIBUTOS anteriores, tienen la responsabilidad de dirigir un departamento y reciben un *bono* específico por ello.

El problema es mantener un registro de empleados (los últimos, digamos, 12 pagos a cada uno) y ser capaces de ***actualizar*** salarios y bonos y de ***generar*** cheques de pago y reportes globales.

Para resolver el problema con la orientación por objetos, decimos, por ejemplo, que cada tipo de empleado es una **CLASE**. Cada empleado será representado por un objeto de esa clase. La descripción de la CLASE incluye también, los ***METODOS*** para realizar las operaciones que transforman el ESTADO (los *atributos*) de cada objeto, de acuerdo a las circunstancias de uso. Observen que necesitaremos otros "objetos" asociados a los empleados, tales como cheques y reportes.

EJEMPLO 2.0

Queremos controlar la **dedicación** de ciertas empleados a ciertas tareas. Todo **empleado** tiene una *identificación* asociada. Además, cada empleado puede ser responsable de un numero no determinado de *tareas*, cada una de las cuales consume parte de su *horario* semanal y de su *dedicación* anual al trabajo. Hay algunas `restricciones` sobre la dedicación anual al trabajo y sobre el numero de horas asignadas a los diversos tipos de tareas.

En este caso, el objeto principal es el **reporte** de carga laboral de cada **empleado**. El empleado puede ser modelado como un objeto sencillo, uno de cuyos atributos es la *carga* en cuestión. Necesitamos también, algunos objetos que controlen la comunicación entre el usuario del sistema y los objetos anteriores y que permitan elaborar el informe de carga progresivamente.

Lo que acabamos de hacer con esos dos ejemplos es un primer ejercicio de diseño orientado a los objetos, OO. En sendos problemas, hemos identificado los tipos de objetos involucrados (a los que identificamos con **negritas** en el texto) y los atributos (*marcados así*) y sus métodos (***marcados así***). También hemos asomado la posibilidad de incluir otro elemento importante en OO, las `restricciones`.

En la secciones siguiente trataremos de aclarar cada uno de esos términos[4] y el cómo se reflejan en un código Java.

A.1.5. Qué es un objeto (de software)

En OO, un objeto es una *cosa virtual*. En algunos casos (como en los problemas anteriores) es la representación de un objeto físico o "social". Los objetos de software son MODELOS de objetos reales, en ese contexto.

Desde el punto de vista de las estructuras de datos que manipula el computador, un objeto es "una cápsula que envía y recibe mensajes". Es una estructura

[4]Salvo el de restricciones que queda lejos del alcance de un tutorial elemental

de datos junto con las operaciones que la consultan y transforman. Estos son los métodos, los cuáles corresponden a las rutinas o procedimientos en los lenguajes de programación imperativa.

En lo sucesivo y siempre en OO, un objeto es simplemente una *instancia de una clase*. La clase es la especificación del objeto. Una especie de descripción general de cómo son todos los objetos de **este tipo**.

A.1.6. Qué es una clase

Ya lo dijimos la final de la sección anterior. Una clase es una especificación para un tipo de objetos. Por ejemplo, para nuestro primer problema podriamos usar:

```
1 class Empleado  {
2    String nombre;
3    String ID;
4    float salario;  }
```

`Empleado` es una clase de objetos, cada uno de los cuáles tiene un `nombre` (cadena de carácteres, `String`), una identificación (otro `String`) y un `salario` (un número real, `float` en Java).

A.1.7. Qué es una subclase

Una de las ventajas más interesantes de la OO es la posibilidad de **reusar** el software ajeno extendiéndolo y adaptándolo a nuestro problema inmediato. Por ejemplo, si alguien más hubiere creado la clase `Empleado` como en la sección anterior, para crear la clase `Gerente` (que entendemos como un tipo especial de empleado) podríamos usar:

```
1 class Gerente extends Empleado {
2    String departamento;
3    float bono;
4    Gerente(String n, String i, float f) {
5       Empleado(n,i,f);
6       departamento = "Gerencia";
7       bono = 1.20f*f;
8       salario =+ 20 } }
```

A.1.8. Qué es una superclase

La posibilidad de crear subclases implica la existencia de una jerarquía de tipos de objetos. Hacia abajo en la jerarquía, los objetos son más especializados. Hacia arriba son más generales. Así la superclase (clase inmediata superior) de Empleado podría ser la siguiente si consideramos que, en general, un Empleado es una Persona:

```
1 class Persona  {
2    String nombre; String id;
3    public void Persona( String n, String id ) {
4       this.nombre = n; this.id = id;}
5    public void identifiquese( )  {
6       System.out.println("Mi␣nombre␣es␣" + this.id );
           }}
```

Con lo cual, tendríamos que modificar la declaración de la clase Empleado, así

```
1 class Empleado extends Persona {
2    float salario ;
3    public void Empleado(String n,String id,float
        primerSueldo) {
4      Persona( n, id ); this.salario = primerSueldo ; }
5    public void identifiquese() {
6     super.identifiquese();
7     if (this.salario < 1000000 ) {
8        System.out.println("...␣y␣me␣pagan␣como␣a␣un␣
            profesor);␣}}}
```

En estas dos últimas especificaciones de las clase `Persona` y `Empleado` ha aparecido el otro elemento fundamental de un objeto de software: los métodos. Un método es una rutina (como los procedimientos en Pascal o las funciones en C) que implementan el algoritmo sobre las estructuras de datos y atributos del objeto (y de otros objetos asociados).

En OO, los métodos se heredan hacia abajo en la jerarquía de clases. Por eso, los métodos de nuestra clase `Persona`, son también métodos de la clase `Empleado`, con la posibilidad adicional, en Java, de poder invocar desde una sub-clase, los métodos de su superclase, como se observa en nuestro ejemplo (`super.identifiquese()`[5]).

Un útimo detalle muy importante es la presentación de un tipo especial de método: **el constructor**, que es invocado antes de que el objeto exista, justamente al momento de crearlo (con algo como *new Constructor(Parametros);*) y cuya objetivo suele ser el de iniciar (*inicializar* decimos en computación siguiendo el anglicismo) las estructuras de datos del objeto. Noten el uso del constructor `Empleado( String n, String id, Float primerSueldo )` en el último ejemplo. Los métodos constructores no se heredan. Las previsiones para crear los objetos deben ser hechas en cada clase. Si no se declara un constructor, el compilador Java introduce uno por omisión que no tiene parámetros.

[5]Algunas versiones de la máquina virtual permiten usar super para invocar al constructor de la superclase, siempre que aparezca como la primer instrucción del constructor de la clase hijo. Esto, sin embargo, no siempre funciona.

A.1.9. Qué es poliformismo

Con lo poco que hemos explicado, es posible introducir otro de los conceptos revolucionarios de la OO: el polimorfismo. En corto, esto significa que un objeto puede tener varias formas (cambiar sus atributos) o comportarse de varias maneras (cambiar sus métodos).

Por ejemplo, suponga que Ud declara la variable `fulano` de tipo `Persona` y crea una instancia de `Persona` en `fulano`:

```
Persona fulano = new Persona();
```

Y luego, ejecuta la instrucción

```
fulano.identifiquese();
```

sobre ese `fulano`. La conducta que se exhibe corresponde a la del método `identifiquese` en la clase `Persona`.

Sin embargo, si Ud declara:

```
Empleado fulano = new Empleado();
```

Y luego, ejecuta la instrucción

```
fulano.identifiquese();
```

sobre ese `fulano`. La conducta que se exhibe corresponde a la del método `identifiquese` en la clase `Empleado`, pues en este clase se **sobreescribió el método**. Es decir, se inscribió un método con el mismo nombre y lista de argumentos que el de la super-clase que, queda entonces, oculto.

Noten que este comportamiento diverso ocurre, a pesar que un `Empleado` es también una `Persona`. Esta es una de las dimensiones del polimorfismo.

El polimorfismo, no obstante, tiene un carácter mucho más dinámico. Por ejemplo, esta instrucción es perfectamente válidad en Java, con los códigos anteriores:

```
Empleado fulano = new Gerente("Fulano de Tal","F", 1000000f);
```

La variable `fulano` de tipo `Empleado`, se refiere realmente a un `Gerente` (que es una subclase de `Empleado`).

En este caso, fulano es un Empleado. Si Ud trata de utilizar algún atributo específico de la subclase, no podrá compilar. Trate con:

```
1 Empleado fulano = new Gerente("Fulano de Tal","F", 1000000f);
2 System.out.println("mi bono es "+fulano.bono);
```

Aquí `bono`, a pesar de ser un atributo de la clase `Gerente`, es desconocido si el objeto es de tipo `Empleado`. El compilador lo denuncia. Al revés, en principio la instanciación también es posible:

```
Gerente fulano = (Gerente) new Empleado("Fulano de Tal", "F", 1000000f) ;
```

El compilador lo admite como válido al compilar. Sin embargo, la máquina virtual[6] lanzará una excepción (el error es `ClassCastException`) en tiempo de ejecución. Todo esto es parte del sistema de seguridad de tipos de Java. No hay espacio en el tutorial para explicar todas sus implicaciones.

[6] Algunas máquinas virtuales) no reportan error de ejecución.

A.1.10. Qué es un agente

En la jerga de la Inteligencia Artificial, un agente es un dispositivo (hardware y software) que percibe su entorno, razona sobre una representación de ese entorno y de su conducta y actúa para cambiar ese mismo entorno (o para impedir que cambie). El término, desde luego, está asociado a la etimología tradicional (del latin *agere*: hacer) y se le usa para indicar a aquel (o aquello) que **hace algo** (posiblemente en beneficio de alguien más). La adopción del término en la literatura ingenieril de la Inteligencia Artificial tiene que ver con la aparición de una corriente o escuela de trabajo que enfatiza que los dispositivos inteligentes deben demostrar que lo son "haciendo algo". En el extremo opuesto (los que no hacen nada) se pretendió dejar a los primeros razonadores simbólicos que concentraban todo el esfuerzo en emular el "pensar" de los humanos, con un mínimo compromiso por la acción.

Como ilustramos en este libro, en Galatea nos reencontramos con esa noción del razonador simbólico que procesa una representación de cierto dominio de conocimiento (asociado a un sistema) y usa el producto de ese procesamiento para guiar la acción de un agente. El objetivo de proyecto Galatea, en cuanto a agentes se refiere, es proveer a modelistas y simulistas de un lenguaje con suficiente expresividad para describir a los humanos que participan de un sistema que se quiere simular. Por esta razón, nuestros agentes son inteligentes.

Un agente inteligente tiene su propio "proyecto" de vida, con metas, creencias y capacidades que le permiten prestar, con efectividad, servicios para los que se requiere inteligencia

A.1.11. Qué relación hay entre un objeto y un agente

Algunos autores reduce la caraterización de agente a una entidad que percibe y actúa [79]. Usando la matemática como lenguaje de especificación, dicen ellos que un agente es una función con dominio en una "historia perceptual" y rango en las acciones:

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

Esta caracterización matemática es ciertamente muy útil conceptualmente (con algunos complementos, como mostramos en los capítulos centrales de este libro), pero lo es poco para el desarrollador de software. Es quizás más provechoso decir que un agente es un objeto, en el sentido que hemos venido discutiendo, con una sofisticada estructura interna (estructuras de datos, atributos y métodos) y con una relación intensa de intercambio con su entorno. En Galatea, esta visión de agente se refleja en la implementación de la clase `galatea.hla.Agent` y en el paquete `galatea.gloria`, como explicamos en el libro.

A.1.12. Por qué objetos

La programación orientada a los objetos se ha impuesto tan arrolladoramente que pocos harán ahora la pregunta que intitula esta sección. Sin embargo, queremos hacer énfasis en lo apropiado de la OO, y ahora la orientación a los agentes,

OA, como herramientas para abordar la complejidad. Los profesores Oscar García y Ricardo Gutierrez, de Wright University [40], explican que una medida del esfuerzo de desarrollo en programación imperativa tal como:

$$Esfuerzo\ de\ programacion\ =\ constante*(lineas\ de\ codigo\ total)*1{,}5 \quad \text{(A.1)}$$

Se convierte en la OO:

$$E.\ de\ pr.\ en\ OO\ = constante\ *\Sigma_{objetos}(l.\ de\ c.\ de\ cada\ objeto)*1{,}5 \quad \text{(A.2)}$$

Debemos admitir, sin embargo, el esfuerzo considerable que debe realizar un programador para aprender los lenguajes OO, buena parte del cual debe emplearlo en aprender lo que otros programadores han hecho. De su habilidad para re-utilizar, dependera su éxito casi definitivamente. En particular, un lenguaje como Java que conserva parte de la sintáxis de nivel intermedio del lenguaje C puede resultar muy desafiante. Por esta razón, este texto incluye el siguiente recorrido Tutorial sobre Java.

A.2. Java en 24 horas

A.2.1. Java de bolsillo

El material incluido en esta sección está basado en un curso dictado por Andrew Coupe, Sun Microsystems Ltd, en 1994. Ha sido actualizado con los detalles relevantes para Java 2, la nueva generación de Java y hemos cambiado su ejemplo para adecuarlo a un problema más cercano a los simulistas.

Coupe resumió las características de este lenguaje OO, así: "Java es un lenguaje pequeño[7], simple, seguro, orientado a objetos, interpretado o optimizado dinámicamente, que genera byte-codes, neutral a las arquitecturas, con recolección de basura, multihebrado, con un riguroso mecanismo para controlar tipos de datos, diseñado para escribir programas dinámicos y distribuidos". Más allá del ejercicio promocional, tradicional en un entorno industrial tan competido como este, esa última definición debe servir para ilustrar la cantidad de tecnologías que se encuentran en Java. Esa combinación junto con una política (más) solidaria de mercadeo y distribución, han convertido a Java, a nuestro juicio, en una plataforma de desarrollo muy efectiva y muy robusta.

Para el usuario final, el programador de aplicaciones, quizás la virtud que más le puede interesar de Java es la facilidad para desarrollos de software distribuidos: "Escríbelo aquí, córrelo dondequiera" es uno de los panfletos publicitarios de Java. Significa que uno puede producir el software acabado en una máquina con arquitectura de hardware completamente diferente de aquella en la que se ejecutará el software. Una máquina virtual se encarga de unificar las "interfases" entre programas Java y las máquinas reales.

[7] ya no lo es

Estas fueron la metas del diseño de Java. Las listamos aquí con una invitación al lector para que verifique su cumplimiento:

- Java tiene que ser simple, OO y familiar, para lograr que los programadores produzcan en poco tiempo. JAVA es muy parecido a C++
- Java debe ser seguro y robusto. Debe reducir la ocurrencia de errores en el software. No se permite el goto. labels, break y continue, si se permiten. Pero sin las armas blancas de doble filo: No a la aritmética de punteros (apuntadores)
- Java debe ser portátil, independiente de la plataforma de hardware.
- JAVA tiene que ser un lenguaje útil (usable).
- Java implanta un mecanismo para recolección de basura. No a las filtraciones de memoria.
- Java solo permite herencia simple entre clase (una clase solo es subclase de una más). Es más fácil de entender y, aún, permite simular la herencia múltiple con `interface`.

Java trata de eliminar las fuentes más comúnes de error del programador C. No permite la aritmética de apuntadores. Esto no significa que no hay apuntadores, como algunas veces se ha dicho. En OO, una variable con la referencia a un objeto se puede considerar un apuntador al objeto. En ese sentido, todas las variables Java (de objetos no primitivos, como veremos) son apuntadores. Lo que no se permite es la manipulación algebraica de esos apuntadores que si se permite en C o C++.

El programador no tiene que preocuparse por gestionar la memoria. Esto significa que no hay errores de liberación de memoria. El trabajo de reservar memoria para los objetos se reduce a la invocacion **new** que crea las instancias, pero sin que el programador tenga que calcular espacio de memoria (como se puede hacer con algunas invocaciones *malloc* en C). Para liberar la memoria, tampoco se requiere la intervención del programador. La plataforma incorpora un "recolector de basura" (garbage collector) que se encarga de recuperar a memoria de aquellos objetos que han dejado de ser utilizados (ninguna variable apunta hacia ellos). El mecanismo puede ser ineficiente sin la intervención del usuario, pero el mismo recolector puede ser invocado a voluntad por el programador (Ver objeto gc).

Java es seguro por tipo. Esto quiere decir que se establecen tipos de datos muy rígidos para controlar que la memoria no pueda ser empleada para propósitos no anticipados por el programador. El mapa de memoria se define al ejecutar.

Hay un chequeo estricto de tipos en tiempo de compilación que incluye chequeo de cotas, una forma de verificación antivirus, cuando las clases se cargan a través de la red. El cargador de clases les asigna a los *applets* de diferentes máquinas, diferentes espacios de nombres.

applets

Los **applets** , estas aplicaciones Java que se descargan desde la red, operan

en lo que se conoce como una “caja de arena" (como las que usamos para las mascotas) que impide que el código desconocido pueda afectar la operación de la máquina que lo recibe. Esto es protección contra caballos de troya. Los applets son revisados al ejecutarlos. El cargador chequea los nombres locales primero. No hay acceso por la red. Los applets sólo puede acceder a su anfitrión. Los applets más allá de una firewall, sólo pueden acceder al espacio exterior.

Java pretende ser portátil (neutral al sistema operativo). Nada en JAVA debe ser dependiente de la implementación en la máquina donde se le ejecute. El formato de todos los tipos está predefinido. La definición de cada uno incluye un extenso conjunto de bibliotecas de clases, como veremos en un momento. Los fuente JAVA son compilados a un formato de *bytecode* independiente del sistema. Esto es lo que se mueve en la red. Los *bytecodes* son convertidos localmente en código de máquina. El esfuerzo de portabilidad es asunto de quienes implementan la máquina virtual en cada plataforma.

Java tiene tipos de datos llamados primitivos. Son enteros complemento dos con signo. byte (8 bits), short (16 bits), int (32 bits), long (64 bits); Punto flotantes IEEE754, sencillo (float) y doble (double); y Char 16 bit Unicode.

Java es interpretado o compilado justo a tiempo para la ejecución. Esto significa que código que entiende la máquina real es producido justo antes de su ejecución y cada vez que se le ejecuta. La implicación más importante de esto es que el código Java tarda más (que un código compilado a lenguaje máquina) en comenzar a ejecutarse. Esto ha sido la fuente de numerosas críticas a la plataforma por su supuesta ineficiencia. Un análisis detallado de este problema debe incorporar, además de esas consideraciones, el hecho de que los tiempos de descarga a través de la red siguen siendo un orden de magnitud superiores a la compilacion *just-in-time* y, además, que la plataforma Java permite enlazar bibliotecas (clases y paquetes) en tiempo de ejecución, incluso a través de la red.

La otra gran idea incorporada a Java desde su diseño es la plataforma para multiprogramación. Los programadores Java decimos que Java es multihebrado. El programador puede programar la ejecución simultánea[8] de “hilos de ejecución", hebras, en un mismo programa. La sincronización (basada en monitores) para exclusión mutua, un mecanismo para garantizar la integridad de las estructuras de datos en los programas con hebras, está interconstruida en el lenguaje (Ver `Threads`).

A.2.2. Los paquetes Java

Un paquete es un módulo funcional de software y la via de crear colecciones o bibliotecas de objetos en Java. Se le referencia con la instrucción `import` para incorporar sus clases a un programa nuevo. Lo que llamamos la Application Programing Interface, API, de Java es un conjunto de paquetes con la documentación apropiada para que sean empleados en desarrollos. La especificación

[8] Por lo menos pseudosimultánea. El paralelismo real depende, desde luego, de contar con más de un procesador real y una máquina virtual que paralelice.

JAVA incluye un conjunto de bibliotecas de clases que acompañan cualquier distribución de la plataforma: `java.applet, java.awt, java.io, java.lang, java.net, java.util`.

El paquete `java.applet` proporciona una clase para programas Java imbuidos en páginas web: Manipulación de applets. Métodos para manipular audioclips. Acceso al entorno del applet.

El paquete `java.awt` es la caja de herramientas para armar GUIs, mejorada (ya es casi obsoleta) con la introducción de la familia Swing. Tanto Awt como Swing implementan los componentes GUI usuales: `java.awt.Graphics` proporciona algunas primitivas de dibujo. Se apoya en el sistema de ventajas del sistema operativo local (`java.awt.peer`). `java.awt.image` proporciona manipuladores de imágenes.

El paquete `java.io` es la biblioteca de entrada y salida: Input/output `streams`. Descriptores de archivos. `Tokenizers`. Interfaz a las convenciones locales del sistema de archivos.

El paquete `java.lang` es la biblioteca del lenguaje JAVA: Clases de los tipos (`Class`, `Object`, `Float`, `String`, etc). La Clase `Thread` provee una interfaz independiente del sistema operativo para las funciones del sistema de multiprogramación.

El paquete `java.net` contiene las clases que representan a objetos de red: Direcciones internet. `URL`s y manipulares `MIME` Objetos `socket` para clientes y servidores.

El paquete `java.util` Un paquete para los "utilitarios": `Date`, `Vector`, `StringTokenizer`, `Hashtable`, `Stack`.

JAVA es una plataforma de desarrollo de software muy económica. Muchos proveedores de tecnología teleinformática son propietarios de licencias de desarrollo JAVA (IBM por ejemplo) aunque Sun Microsystems sigue siendo la casa matriz, ahora parte de Oracle, Inc.

Los Java Developer's Kits son los paquetes de distribución de la plataforma y ahora se denominan Software Developer Kits. Se distribuyen gratuitamente para todos los sistemas operativos comerciales. Incluye herramientas de documentación y BDKs. Hay muchos IDEs (NetBeans, por ejemplo). Browsers Explorer y Netscape lo incorporaron desde el principio como uno de sus lenguajes script (pero, cuidado!, que Java NO ES JAVASCRIPT).

Es muy difícil producir un resumen que satisfaga todas las necesidades. Por fortuna, la información sobre Java es muy abundante en Internet. Nuestra intención aquí es proveer información mínima y concentrar la atención del lector en algunos detalles útiles para simulación en las próximas secciones.

A.2.3. Java Libre

En el 2006, y luego de considerable presión por parte de miles de desarrolladores alrededor del mundo[9], la empresa Sun, propietaria de los derechos

[9] por ejemplo, aquellos que alimentan el repositorio de software libre sf.net

de autor de Java (y empleadora de su creador), decidió liberar el código de la plataforma.

En la comunidad de software libre se dice que una pieza de software ha sido liberada cuando se permite distribuirle de acuerdo a ciertos términos explicados en un texto de licencia. Sun, dueños de Java, decidieron adoptar una muy conocida licencia de software libre, la GPL, con una excepción también muy popular: la *excepción* `classpath`. En términos simples, eso significa que el código de Java es libre para ser usado, leído, copiado y mejorado, siempre que cualquier mejora a la plataforma sea también compartida en los mismos términos. Es decir, el código Java de Sun tiene ahora "izquierdo de copia".

Sin embargo, para no forzar a que cualquier aplicación hecha con esa plataforma fuese también libre, Sun incorporó la *excepción* `classpath`, inventada por la Free Software Foundation para su proyecto homónimo, con la cuál sólo las modificaciones de la plataforma base obligan al izquierdo de copia. Es decir, si uno usa la plataforma como una librería para construir sus programas, no está obligado al izquierdo de copia (tal como ocurre con otras implementaciones de Java, e.g. Harmony, del proyecto Apache). Ese derecho del autor[10] sólo se aplicará para los cambios y mejoras sobre la plataforma de Sun.

En el proyecto Galatea hemos decidido liberar el código exactamente en los mismos térmios: **GPL con la excepción *Classpath***. Esto significa que no obligaremos a nadie a liberar sus códigos si construyen invocando a Galatea. Sólo esperamos que las mejoras a la propia plataforma sean también libres y podamos así seguir construyendo una herramienta de simulación libre, eficiente y efectiva.

A.2.4. Un applet

Un applet[11] es una forma inmediata de agregar dinámismo a los documentos Web. Es una especie de extensión indolora del browser, para las necesidades de cada página Web. El código se descarga automáticamente de la red y se ejecuta cuando la página web se expone al usuario. NO se requiere soporte en los servidores Internet, salvo para alojar el programa y la página Web.

Sin más preámbulo, esto es un applet:

```
1 import java.awt.*;
2 import java.applet.*;
3 public class EjemploApplet extends Applet {
4   Dimension appsize;
5   public void init() { appsize = this.size(); }
6   double f(double x) {
7      return (Math.cos(x/5) + Math.sin(x/7) + 2) *
          appsize.height / 4; }
```

[10] sí, el izquierdo de copia es un derecho del autor

[11] El nombre es una variación de la palabra inglesa para aplicación "enchufable". Estas variaciones se han vuelto comunes para la comunidad Java con la aparición de los *Servlets*, los *Midlets* y los *Aglets*, todos componentes `enchufables`.

```
8   public void paint(Graphics g) {
9      for ( int x = 0; x < appsize.width ; x++ ) {
10       g.drawLine(x, (int) f(x), x+1, (int) f(x+1));
           }}}
```

Este último es un programa muy sencillo que dibuja una grá fica de la funcion `f` al momento de cargar el applet (que es cuando el sistema invoca el método `init()`. Noten que los applets no usan la función `main()` como punto de arranque los programas. Esta es reservada para las aplicaciones normales).

Para ejecutar el applet puede hacerse (luego de generar el `.class`):

```
appletviewer EjemploApplet
```

o invocarlo desde una página web como se mostrará más adelante.

A.2.5. Anatomía de un applet

Una applet es una aplicación Java construida sobre la colecciónn de clases en el paquete `java.applet` o `java.swing`.

Así se declara la clase que lo contiene:

```
import java.applet.*;
public class AppletMonteCarlo extends Applet {
}
```

Donde `AppletMonteCarlo` es el nombre que hemos escogido para nuestro ejemplo (y que el programador puede cambiar a voluntad, desde luego).

A diferencia de las aplicaciones "normales", que solo requieren codificar un método `main()` como punto de arranque, los applets deben configurar, por lo menos 2, puntos de arranque. Con `init()` se indica al sistema el código que debe ejecutarse cada vez que se descargue la página web que contiene el applet. Con `start()` se indica el código que habrá de ejecutarse cada vez que el applet se exponga a la vista del usuario del navegador (cada vez que la página web se exhiba o se "maximice" la ventana del navegador).

```
Import java.applet.*;
public class AppletMonteCarlo extends Applet {
  public void init() { }
  public void start() { } }
```

A.2.6. Parametrizando un applet

Los applets, imbuidos en páginas Web como están, tienen acceso a información el esas páginas. Por ejemplo, nuestro applet podría obtener un nombre que necesita, de una de las etiquetas de la página que lo aloja, con este código:

```
1 import java.applet.*;
2 public class AppletMonteCarlo extends Applet  String param;
3 public void init()  param = getParameter("imagen");
```

y con este texto incluido en la página, justo en el punto donde se inserta el applet. En los ejemplos que acompañan este texto, se muestra la página web (`AppletMonteCarlo.html`) que invoca a este applet, sin ese detalle del parámetro que se sugiere como ejercicio al lector.

```
param name=imagen value="imagen.gif"
```

A.2.7. Objetos `URL`

El appleto puede, además, usar las clase de apoyo para construir direcciones Web bien formadas y usarlas en la recuperación de información:

```
1 import java.applet.*; import java.net.*;
2 public class AppletMonteCarlo extends Applet  {
3    String param;
4   public void init() {
5      param = getParameter("imagen");
6      try {
7        imageURL = new URL(getDocumentBase(), param);
8      } catch (MalformedURLException e) {
9        System.out.println("URL␣mal␣escrito"); } return; } }
```

A.2.8. Gráficos: colores

El applet es una “aplicación gráfica". Es decir, toda la salida se realiza a través de un panel que es una ventana para dibujar. No es debe sorprender que exista una relación muy estrecha entre applets y herramientas de dibujo.

El siguiente código crea un arreglo para almacenar una paleta de 3 colores que se usará más adelante.

```
1 import java.applet.*;
2 int paints[] ;
3 /* prepara la caja de colores */
4 paints = new int[3];
5 paints[0]=Color.red.getRGB();
6 paints[1]= Color.green.getRGB();
7 paints[3]= Color.blue.getRGB();
```

Lo más interesante de esta relación entre applets y dibujos no es sólo lo que uno puede programar, sino lo que ya ha sido programado. Este código, por ejemplo, contiene una serie de objetos que nos permiten cargar una imagen a través de la red. La imagen puede estar en alguno de los formatos profesionales de internet (.gif, jpeg, entre otros). El sistema dispone de un objeto para almacenar la imagen (`Image`) y, además de todo un mecanismo para garantizar que la imagen es cargada integramente (`MediaTracker`).

```
1 Image picture;
2 MediaTracker tracker;
```

```
tracker = new MediaTracker(this);
picture = this.getImage(imageURL) ;
tracker.addImage(picture,0);
```

Java provee también de objetos para que el programador puede manipular gráficos (`Graphics`) que se dibujan en su panel.

```
Graphics gc;
gc = getGraphics;
```

A.2.9. Ejecutando el applet

El código a continuación, es una implementación del método start() que muestra como cargamos la imagen (con el `tracker`) y luego la dibujamos en el panel del applet (`gc.drawImage`), suponiendo, por supuesto que los objetos han sido declarados e inicializados en otros lugares del código.

```
import java.applet.*;
public class AppletMonteCarlo extends Applet  {
  String param; ...
  public void start()  {
    try  { tracker.waitForID(0); }
    catch (InterrumptedException e)  {
      System.out.print("No␣pude!"); }
  imagewidth = picture.getWidth(this);
  imageheigth = picture.getHeight(this);
  gc.drawImage(picture, 0, 0, this);  }
```

A.2.10. Capturando una imagen

Con muchas herramientas de visualización científica se hace un esfuerzo, quizás exagerado, por aislar al usuario de los detalles de manipulación de su data. En muchos casos, por el contrario, el usuario-programador necesita todas las facilidades para manipular esa data para sus propósitos particulares.

El código que se muestra a continuación es un ejemplo de cómo cargar la imagen que hemos obtenido a través de la red (en el objeto picture) en un arreglo de pixels. Un pixel corresponde a un punto en la pantalla del computador. Para el software, el pixel es un número que designa el color e intensidad que se "pinta" en cierta posición de la pantalla.

```
PixelGrabber pg; int pixels[] ;
pixels = new int[imagewidth*imageheight] ;
pg = new PixelGrabber(picture, 0, 0, imagewidth, imageheight, pixels, 0, imagewidth);
try { pg.grabPixels(); }
catch (InterruptedException e)  {
  System.err.println("No␣pude␣transformar␣la␣imagen"); }
```

Lo que hemos hecho ese código es capturar (agarrar, *Grabber*. Ver `PixelGrabber`) los píxeles de nuestra imagen en un arreglo desde donde los podemos manipular a voluntad (como se muestra en el ejemplo completo).

A.2.11. Eventos

Las aplicaciones de software moderna rara vez interactuan con el usuario a través de un comandos escritos con el teclado. Las formas más populares de interacción tienen que ver con el uso del ratón, de menús, de botones y de otra opciones gráficas.

Un problema con esos diversos modos de interacción es que requieren ser atendidos casi simultáneamente y en forma coherente con la aplicacion y con sus interfaz al usuario.

Para resolver ese problema de raíz, la plataforma Java incorporó, desde el principio, un sistema de manejo de eventos en su interfaz gráfica. Los eventos son, para variar, representados como objetos (de la clase `Events`) y dan cuenta de cualquier cosa que pase en cualquier elemento de interfaz (o en cualquier otro, de hecho). Un evento es un objeto **producido** por aquel objeto sobre el que ocurre algo (por ejemplo un botón que es presionado, *click*). El objeto que produce el evento deben tener asociado un manejador del evento. Este es otro objeto que **escucha** los eventos (un `Listener`), los **reconoce** (de acuerdo a caracteristicas predefinidas por el programador) y los **maneja**, ejecutando ciertos códigos que han sido definidos por el programador. En un dramático cambio de dirección, Java 2 transformó el mecanismo de manejo de eventos y es por ello que algunos viejos programas Java que manipulaban eventos, tienen dificultades para funcionar.

Lamentablemente, los detalles del manejo de eventos son difíles de resumir. Es mucho más productivo ver el código de algunos ejemplos. Este, por ejemplo, es el que usamos en el applet que hemos venido construyendo. Observen como cambia la declaración de la clase para "implementar" los objetos que escuchan eventos en cada dispositivo:

```
1  public class AppletMonteCarlo extends Applet implements
       KeyListener, MouseListener {
2    ...
3    public void keyPressed(KeyEvent e) {  System.out.println("
       keyPressed"); }
4    public void keyReleased(KeyEvent e) {
5       System.out.println("keyReleased");
6       /* pasa al nuevo color */ paintcol++; paintcol =
          paintcol
7       System.out.println("Nuevo␣color:␣" + paintcol ); }
8    public void keyTyped(KeyEvent e) { System.out.println("
       keyTyped"); }
9    public void mouseClicked(MouseEvent e) {
10     System.out.println("mouseClicked");
11     /* Captura la posicion del ratón */
12     int x = e.getX(); int y = e.getY();
13    /* Un click dispara la reconstrucción de la imagen */
14    if (x < imagewidth & y < imageheight)
15       /* llamar a la rutina de llenado: floodFill */
16       floodFill(x,y, pixels[y*imagewidth+x]);
```

```
17        /* .. y luego la redibuja */
18        repaint(); // desde el browser llama a paint()
19    public void mouseEntered(MouseEvent e) { System.out.
          println("mouseEntered");}
20    public void mouseExited(MouseEvent e) { System.out.println
          ("mouseExited"); }
21    public void mousePressed(MouseEvent e) { System.out.
          println("mousePressed");}
22    public void mouseReleased(MouseEvent e) { System.out.
          println("mouseReleased"); }
23    .... }
```

En este caso, los interesantes son los métodos `keyRelease()` y `mouseClicked()`. Los demás no hacen nada (salvo imprimir un mensaje en salida standard), pero **deben** ser implementados (Ver `interface`s en Java).

Sugerimos al lector que, sobre el código completo de este ejemplo ubique a los productores de eventos y a los escuchas.

A.2.12. `paint()`:Pintando el applet

Un detalle importante para cerrar la discusión sobre visualizació (sobretodo la animada) es quien pinta el dibujo del applet. Lo pinta el usuario cada vez que quiere, pero también lo pinta el propio navegador (browser) se exhibe es la página, en respuesta a su propia situación (abierta, cerrada, bajo otra ventanas, etc.).

Para proveer una interfaz uniforme al sistema y al programador, se ha dispuesto del método paint(). El programador escribe el código de paint, para definir como desea que "se pinte" sus applet. Pero el método es invocado por el navegador. Si el usuario quieren forzar una llamada a `paint()`, usa `repaint()`, como se muestra en el ejemplo.

Este es el código que usamos para `paint()`. Noten el uso de otro método `update()`.

```
1  public void paint(Graphics gc) { update(g); }
2  public void update(Graphics g) {
3    Image newpic;
4    newpic = createImage( new MemoryImageSource(imagewidth,
         imageheight, pixels, 0, imagewidth );
5    g.drawImage(newpic, 0, 0, this) ; }
```

A.2.13. El AppletMonteCarlo completo

.

Como dijimos al principio de esta parte, hemos querido ofrecer un ejemplo completo de una aplicación Java funcional, destacando el manejo gráfico en el que Galatea no es tan fuerte (por falta de tiempo de desarrollo). Eso es el AppletMonteCarlo. Pero su nombre también sugiere una herramienta conceptual sumamente popular en simulación.

El método de MonteCarlo es una aplicación de la generación de números aleatorios. En nombre rememora el principado Europeo, célebre por sus casinos y sus juegos de azar (entre otras cosas).

No vamos a diluirnos en los detalles, que se obscurecen fácilmente, de la estocástica. Lo que tenemos en el ejemplo es una aplicación simple de la rutina de MonteCarlo de generación de números aleatorios para estimar areas encerradas por un perímetro que dibuja una función.

Lo interesante del ejemplo es que la función no tiene que ser alimentada la ejemplo con su fórmula matemática o con una tabla de puntos. Una imagen (.gif o .jpeg) es el forma que el programa espera para conocer la función sobre la que calculará el área, al mismo tiempo que cambia los colores de puntos para ilustrar el funcionamiento del algoritmo. No se ofrece como un método efectivo para el cáculo de áreas, sino como un ejemplo sencillo de la clase de manipulaciones que se pueden realizar con Java. Noten, por favor, que en este ejemplo, el simulador de números aleatorios empleado no es el originario de Java, sino la clase `GRnd` de **Galatea**[12].

Ahora pueden ir hasta el repositorio a descargar la última versión.

[12]Es muy importante notar como inicializamos la "semilla" del generador Galatea en el método `init()` del `AppletMonteCarlo`.

Apéndice B

ULAnix Oraculum, una forma rápida de probar Galatea

Galatea es una compleja pieza de software. No sólo combina dos lenguajes muy diferentes con larga tradición en la computación moderna (Java y Prolog), sino que introduce nuevos lenguajes (Glider/Galatea, Actilog y OpenLog) e integra los medios para procesar códigos en todos esos lenguajes e interfaces gráficas para usuarios de sus modelos. En consecuencia, levantar la plataforma de desarrollo al punto que cualquiera pueda construir sobre ella no es sencillo. De hecho, tampoco es sencillo preparar una máquina para que un usuario cualquiera pueda probar los modelos acabados.

Por esta razones, nos propusimos desde el principio aprovechar otro desarrollo en el que hemos estado involucrados para entregar con este libro, un sistema listo para usar. Las distribuciones de software "vivas"[1] sirven este propósito perfectamente. ULAnix[65, 8] es un ejercicio local de distribución de software a la medida. En este caso, lo hemos ajustado a la medida de un modelista o simulista que quiera revisar el proyecto Galatea.

B.1. ULAnix Oraculum

B.1.1. Qué es

ULAnix es una distribución de software a la medida de sus usuarios. Es es un esfuerzo local que adopta una estrategia de distribución de software listo para usar que ha venido atrayendo interés, gracias a las libertades que ofrece el software libre. El software se "empaqueta" en algún medio, como CD, DVD o memoria flash USB, que puede ser usado para "arrancar" (boot) la máquina con todo el sistema listo para funcionar, sin necesidad de instalación o configuración. Los administradores o desarrolladores de software seleccionan los programas que cierto usuario o grupo de usuarios realmente necesitan, y los pre-instalan y

[1] `http://en.wikipedia.org/wiki/LiveDistro`

configuran en una "imagen" que es luego cargada y adaptada a cada hardware "en vuelo". En algunos casos, se constituye toda una familia de "imagenes" para propósitos cercanos pero todavía distinguibles. Por ejemplo, ULAnix Scientia es software científico a la medida de los investigadores que requieren herramientas estadísticas, matemáticas y de computación. Fue diseñada con los estudiantes de ciencias e ingeniería en mente.

ULAnix Oraculum

Así, **ULAnix Oraculum** es nuestra versión auxiliar de ULAnix para distribuir a Galatea lista para ser usada, ahorrándole al interesado o interesada horas de configuración de la plataforma subyacente. En todos los casos se trata de una **Distro Viva** *LiveDistro* desarrollada en nuestra Universidad de Los Andes a partir de la distribución Debian de GNU/Linux. El objetivo general del proyecto es proveer los mecanismos y procesos técnicos para el cultivo del Software Libre en el seno de la Universidad.

Distro Viva

A través del proceso madre ULAnux (La MetaDistro), es posible producir versiones pre-instaladas y pre-configuradas, para aplicaciones específicas, del Sistema Operativo Libre GNU/Linux en, por ejemplo, un "CD vivo" (*LiveCD*), en un "DVD vivo" (*LiveDVD*) o en una memoria portátil viva (*LiveUSB*). Con estas versiones "vivas" de un sistema operativo pre-instalado en un medio portátil, el usuario se exime del trabajo de configuración de su ambiente virtual de actividades, y de la dependencia de un computador particular que haya sido así preparado. Este Software Libre permite que cualquier usuario pueda disponer de los programas que requiere para hacer su trabajo, configurados y listos para funcionar en prácticamente cualquier hardware que el usuario se encuentre mientras se desplaza para trabajar. Esta independencia del hardware es, no solamente una alternativa para facilitar el trabajo de cada usuario, sino un mecanismo para reducir los costos de dotación de hardware y aprovechar al máximo las capacidades computacionales, especialmente en ambientes con recursos limitados.

Es, sin embargo, un enorme desafío (y un logro impresionante) que una misma "imagen" del software pueda hacer funcionar a cualquier máquina, adaptándose a cualquier combinación particular de dispositivos provenientes de miles de procesos distintos de fabricación y configuración. Esto no quiere excusar que ULAnix falle con alguna máquina. Pero ciertamente se debe tener en mente cuando se le haga funcionar en una y otra máquina diferente.

B.1.2. Cómo usarlo

Las Distros Vivas se acomodan en una unidad de memoria o medio de almacenamiento desde el cuál se pueda hacer "arrancar" el computador. Las computadores personales, especialmente las más modernas, pueden seleccionar, con o sin ayuda del usuario, el medio desde el cual leerán el software del sistema operativo que controlará a la máquina mientras el usuario trabaja.

Así que para usar ULAnix Oraculum, **simplemente hace falta colocar el medio** (el CD o DVD en la unidad lectora, el USB en uno de los puertos USB) **en posición y encender (o reencender) la máquina**, cuidando que esta use el software en esa unidad al momento de cargar el sistema operativo (esta

decisión puede ser programada a la máquina en su configuración base o *setup*).

Una forma alternativa, cada vez más popular, de usar una distro viva es con una máquina virtual. Estos son sistemas de software que simulan un ambiente de hardware para que algún sistema operativo cargue en ese "ambiente virtual" como si estuviera haciéndolo en una máquina real. Son cada vez más efectivos en simular el hardware. Así que es perfectamente posible ejecutar ULAnix "dentro" de máquinas virtuales como, por ejemplo, *Virtual Box*.

B.2. Netbeans

B.2.1. Qué es

Los ambientes de desarrollo de software (IDE, *Integrated Development Environment*) son programas que sirven como herramientas de trabajo para los desarrolladores de Software. Suelen combinar un editor de código, un gestor de archivos y otros programas para compilación, depuración y ejecución de los códigos que se estén produciendo. No son programas simples. Ciertamente, no están diseñados para quienes no entienden de programación, sino todo lo contrario y pueden ser difíciles de configurar y entonar. El proyecto Galatea no está atado a ningún IDE particular. De hecho, al principio del Apéndice A se explica cómo usar Galatea sin IDE. Pero si hacemos un esfuerzo para convocar a otros desarrolladores al proyecto, el IDE se vuelve esencial, al punto que hemos decidido embarcarnos en el desarrollo de un IDE para simulistas (el Galatea IDE o GIDE[3, 76] que aún no está listo).

Para la plataforma Java hay dos IDE muy populares: Netbeans y Eclipse. Galatea funciona con ambos. Sin embargo, en esta distribución de software hemos decidido usar NetBeans con la intención de que podamos aprovechar la plataforma de desarrollo de interfaces para el usuario y de programación visual que posee.

B.2.2. Cómo usarlo

Netbeans es un programa que se instala en el sistema de menus del ambiente gráfico de su computadora. En ULAnix Oraculum, por ejemplo, se lo conseguirá por la vía Aplicaciones → Programación → Netbeans, o la equivalente.

B.3. Galatea en ULAnix Oraculum

B.3.1. Cómo descargarla desde el repositorio

El repositorio del código es un componente esencial del proyecto Galatea. Nos permite coordinar el esfuerzo del grupo y ofrecer de inmediato el software a todos los interesados en cualquier parte del mundo conectado a Internet. Este gran servicio se lo debemos al proyecto Sourceforge.net, quienes mantienen una plataforma relativamente fácil de administrar y sumamente robusta. El núcleo

del servicio es el sistema Subversión, un programa para manejo de versiones, que nos permite agregar las contribuciones de cada programador sin riesgo alguno de confusión o pérdida de información. Siempre podemos revertir los cambios y consolidar las contribuciones parciales en las definitivas.

Una gran ventaja de Netbeans es que incorpora otras dos herramientas importantes: 1) Un módulo Subversion, que nos permite hacer todas las operaciones sobre repositorio desde el IDE y 2) Un módulo Ant, un programa para gestión de los llamados scripts de configuración: códigos de alto nivel escritos con sintáxis XML, que nos permiten programar procesos completos de compilación y armados de los programas que se entregan para distribuir y usar en cada computador.

Así, para configurar NetBeans con una copia fresca de Galatea descargada desde el repositorio, recomendamos a los programadores interesados seguir la chuleta.netbeans, preparada por Kay Tucci y disponible en el mismo repositorio [2] o las recomendaciones de los Blogs Galateando y Simulación Lógica y Sistemas MultiAgente [3].

B.3.2. Cómo usarla

No obstante, todo ese trabajo de configuración de subversión y ant está hecho en la instalación de ULAnix Oraculum. El programador puede repetirlo si lo desea, pero es posible usar Galatea de una vez. Los fuentes han sido descargados desde el repositorio y una colección de programas `build.xml` en cada paquete importante, le permiten al usuario ejecutar los códigos sin más.

B.3.3. Cómo ejecutar los modelos de simulación.

Esos códigos Ant en xml son pequeños programas que facilitan las tareas de los programadores y de los usuarios. Cada uno automatiza una serie de tareas identificadas por sus nombres o *targets*. Por ejemplo, para recompilar a Galatea, dentro de NetBeans, en el panel de Proyectos, la rama de los Fuentes (*Sources*), dentro del paquete `galatea`, el script `build.xml`, contiene los targets **build** y jar. Un click los pone a correr, en este caso, sin consecuencia para el usuario, pues sólo se trata de preparar la librería base. Para probar uno de los programas con los modelos acabados intente, por ejemplo, dentro del paquete `contrib.burocracia` ejecutar primero el target **jar** y luego el target **run**.

[2] `https://galatea.svn.sourceforge.net/svnroot/galatea/Galatea/trunk`
[3] `http://galateando.blogspot.com`,`http://slysma.wordpress.com`

Bibliografía

[1] Dublin core metadata initiative, 2006. Dublin, USA.

[2] James F. Allen. Temporal reasoning and planning. In J. F. Allen, H. Kautz, R. Pelavin, and J. Tenenberg, editors, *Reasoning About Plans*. Morgan Kauffmann Publishers, Inc., San Mateo, California, 1991. ISBN 1-55860-137-6.

[3] Marisela Cabral. Prototipo del módulo GUI para la plataforma de simulación galatea, 2001. Tutor: Uzcátegui, Mayerlin.

[4] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley, Menlo Park, CA, 1985.

[5] Noam Chomsky. *Language and Mind*. Harcourt Brace Jovanovich, Inc., Massachusetts Institute of Technology, enlarged edition, 1972. ISBN-10: 0155492578 Library of Congress Catalog Card Number: 70-187121.

[6] K.L. Clark. Negation as failure. In H. Gallaire and J. Minder, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[7] Irving Copi and Carl Cohen. *Introducción a la Lógica*. . LIMUSA, 1998.

[8] J. Dávila, J. Carrero, J. Molina, G. Díaz, and D. Hernández. Ulanix scientia: Software a la medida de científicos y tecnólogos. Jornadas de Popularización de la Ciencia, Mérida, 2008.

[9] J. Dávila and M. Reyes. *Systems Thinking and E-Participation: ICT in the Governance of Society*, chapter Articulated Planning. Information Science REFERENCE, 2009. ISBN: 978-1-60566-860-4.

[10] Jacinto A. Dávila. *Agents in Logic Programming*. PhD thesis, Imperial College of Science Technology and Medicine., London, UK, June 1997.

[11] Jacinto A. Dávila. Openlog: A logic programming language based on abduction. In *PPDP'99. International Conference on Principles and Practice of Declarative Programming*, Lecture Notes in Computer Science. 1702, Paris, France, 1999. Springer.

[12] Jacinto A. Dávila. Actilog: An agent activation language. In V Dahl and P. Wadler, editors, *Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science. 2562. Springer, 2003.

[13] Jacinto A. Dávila and Kay A. Tucci. Towards a logic-based, multi-agent simulation theory. In *International Conference on Modelling, Simulation and Neural Networks [MSNN-2000]*, pages 199–215, Mérida, Venezuela, October, 22-24 2000. AMSE & ULA.

[14] Jacinto A. Dávila and Kay A. Tucci. Towards a logic-based, multi-agent simulation theory. *AMSE Special Issue 2000. Association for the advancement of Modelling & Simulation techniques in Enterprises*, pages 37–51, 2002. Lion, France.

[15] Jacinto A. Dávila and Mayerlin Uzcátegui. Galatea: A multi-agent simulation platform. *AMSE Special Issue 2000. Association for the advancement of Modelling & Simulation techniques in Enterprises*, pages 52–67, 2002. Lion, France.

[16] Jacinto A. Dávila and Mayerlin Y. Uzcátegui. Agents that learn to behave in multi-agent simulations. In *Modelling, Simulation and Optimization*. http://iasted.org, 2005.

[17] Jacinto A. Dávila, Mayerlin Y. Uzcátegui, and Kay Tucci. A multi-agent theory for simulation. In *Modelling, Simulation and Optimization*. http://iasted.org, 2005.

[18] Luc de Raedt. *Logical and Relational Learning. From ILP to MRDM*. Springer, 2006.

[19] M. Denecker and D. De Schreye. Sldnfa: an abductive procedure for normal abductive programs. *Proc. International Conference and Symposium on Logic Programming*, pages 686–700, 1992.

[20] Marc Denecker, Lode Missiaen, and Maurice Bruynooghe. Temporal reasoning with the abductive event calculus. In *Proc. European Conference on Artificial Intelligence*, 1992.

[21] Daniel C. Dennett. *Consciousness Explained*. Penguin Books, 1991.

[22] Carlos Domingo. Glider, a network oriented simulation language for continuous and discrete event simulation. In *International Conference on Mathematical Models*, Madras, India, August, 11-14 1988.

[23] Carlos Domingo. Proyecto glider. e-122-92. informe 1992-1995. Technical report, CDCHT, Universidad de Los Andes. Mérida. Venezuela, December 1995.

[24] Carlos Domingo and Marisela Hernández. Ideas básicas del lenguaje glider. Technical report, Instituto de Estadística Aplicada y Computación, Universidad de Los Andes. Mérida. Venezuela, October 1985.

[25] Carlos Domingo and Giorgio Tonella. Towards a theory of structural modeling. volume 10(3), pages 1–18. Elsevier, 2000.

[26] Carlos Domingo, Giorgio Tonella, and Martha Sananes. *GLIDER Reference Manual*. Mérida, Venezuela, 1 edition, August 1996. CESIMO IT-9608.

[27] Umberto Eco. *The Search For The Perfect Language*. Fontana Press. An Imprint of HarperCollinsPublishers, 1995.

[28] K. Eshghi. Abductive planning with the event calculus. In R. A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium of Logic Programming*, Seattle, USA, 1988.

[29] K. Eshghi and R. Kowalski. Abduction through deduction. Technical report, Deparment of Computing. Imperial College, London, UK, 1988.

[30] C.A. Evans. Negation as failure as an approach to the hanks and mcdermott problem. In F.J. Cantu-Ortiz, editor, *Proc. 2nd. International Symposium on Artificial Intelligence*, Monterrey, México, 1989. McGraw-Hill.

[31] Jacques Ferber and Jean-Pierre Müller. Influences and reaction: a model of situated multiagent systems. In *ICMAS-96*, pages 72–79, 1996.

[32] Melvin R. Fitting. A kripke-kleene semantics for logic programs. *The Journal of Logic Programming*, 2:295–312, 1985.

[33] Peter Flach. *Simply Logical: Intelligent Reasoning by Example*. Wiley, 1994. ISBN 0-471-94152-2.

[34] T Fung and R Kowalski. The iff proof procedure for abductive logic programming. July 1996. to appear.

[35] T. H. Fung and Robert A. Kowalski. The iff proof procedure for abductive logic programming. *Journal of Logic Programming*, July 1997.

[36] Tze Ho Fung. *Abduction by deduction*. PhD thesis, Imperial College, London, January 1996.

[37] Dov Gabbay. *What's a logical system?*.

[38] Dov Gabbay, C.J. Hogger, and J.A Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1. Oxford University Press Inc., New York, 1 edition, 1993.

[39] GALATEA Group. Galatea, plataforma de simulación de sistemas multiagentes, 2002.

[40] Oscar Garcia and Ricardo Gutierrez. An update approach to complexity from and agent-centered artificial intelligence perspective. *Encyclopedia of Library and Information Science*, 68. Supplement 31.

[41] Rolando García, editor. *La Epistemología Genética y la Ciencia Contemporánea.* Homenaje a Jean Piaget en su centenario. Gedisa, 1997.

[42] Nigel Gilbert and Klaus Troitzsch. *Simulación para las Ciencias Sociales.* 2005.

[43] GLIDER Development Group. GLIDER simulation language. Technical report, Interdisciplinary Research Center. Department of Mathematical Sciences. College of Sciences, San Diego State University. California. USA, August 1996.

[44] GLIDER Development Group. *GLIDER Reference Manual, Versión 5.0.* Cesimo & IEAC, Universidad de Los Andes, Mérida, Venezuela, 2000. CESIMO IT-02-00.

[45] Erasmo Gómez. Desarrollo de un prototipo del módulo agente para la plataforma de simulación galatea, September 2002. Tutor: Uzcátegui, Mayerlin.

[46] John Hertz, Anders Krough, and Richard G. Palmer. *Introduction To The Theory of Neural Computation.* Addison Wesley, 6 edition, 1993.

[47] Christopher John Hogger. *Essentials of Logic Programming.* Claredon Press, Oxford, 1990.

[48] Nicholas R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, April 2001.

[49] A. C. Kakas, Robert A. Kowalski, and F. Toni. The role of abduction in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming.* Oxford University Press, Oxford, UK, 1995.

[50] Tarun Khanna. *Foundations of Neural Networks.* Addison Wesley, 1990.

[51] S.C. Kleene. On notation for ordinal numbers. *Journal of Symbolic Logic*, 3:150–155, 1938.

[52] S.C. Kleene. *Introduction to Metamathematics.* Van Nostrand, Princeton, 1952.

[53] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica*, (31):249–268, 2007.

[54] Robert Kowalski. Using metalogic to reconcile reactive with rational agents. In K. Apt and F. Turini, editors, *Meta-Logics and Logic Programming.* MIT Press, 1995. Also at http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/recon-abst.html.

[55] Robert Kowalski. *How to be Artificially Intelligent.* 2005. Disponible aquí: http://webdelprofesor.ula.ve/ingenieria/jacinto/kowalski/logica-de-agentes.html.

[56] Robert A. Kowalski. *Logic for Problem Solving*. Elsevier North Holland, New York, 1979.

[57] Robert A. Kowalski. *How to be artificially intelligent*. Cambridge University Press, 2010. (English) http://www.doc.ic.ac.uk/ rak/ (Español) http://webdelprofesor.ula.ve/ingenieria/jacinto/kowalski/logica-de-agentes.html.

[58] Robert A. Kowalski and Fariba Sadri. Towards a unified agent architecture that combine rationality with reactivity. In Dino Pedreschi and Carlos Zaniolo, editors, *LID'96. Workshop on Logic in Databases*, San Miniato, Italy, July 1996.

[59] Kenneth Kunen. Negation in logic programming. *The Journal of Logic Programming*, 4(4):289–308, December 1987.

[60] Klaudia Laffaille. gspaces meta-modelo para simular espacios urbanos y arquitectonicos basado en galatea. Master's thesis, Maestría en Modelado y Simulación, Universidad de Los Andes. Mérida. Venezuela, 2005. Tutores: Tucci Kay, Uzcátegui Mayerlin, Dávila, Jacinto.

[61] Victor Lesser, editor. *ICMAS-95: Proceedings First International Conference on Multi-Agent Systems*, Menlo Park - Cambridge - London, June 1995. American Association for Artificial Intelligence, AAAI Press/The MIT Press.

[62] J.C. Martinez Coll. A bioeconomic model of hobbes' state of natura. *Social Science Information*, (25):493–505, 1986.

[63] Rob Miller. Notes on deductive and abductive planning in the event calculus. http://www-lp.doc.ic.ac.uk/UserPages/staff/rsm/rsm1.html, July 1996.

[64] Lode Missiaen, Maurice Bruynooghe, and Marc Denecker. Chica, an abductive planning system based on event calculus. *Journal of Logic and Computation*, 5(5):579–602, October 1995.

[65] J. Molina, G. Díaz, J. Carrero, and J. Dávila. Ulanux/ulanix: Software académico a la medida. In *Presentado en el Primer Encuentro Venezolano sobre Tecnologías de la Información e Ingeniería del Software. EVETIS'07*, 2007.

[66] Niandry L. Moreno. Diseño e implementación de una estructura, para el soporte de simulación espacial en GLIDER. Master's thesis, Maestría en Computación, Universidad de Los Andes. Mérida. Venezuela, 2001. Tutor: Ablan, Magdiel.

[67] Stephen Muggleton. Inverse entailment and progol. Technical report, The University of York, York, UK, 2002.

[68] Stephen Muggleton and Jhon Firth. *Cprogol4.4: A tutorial introduction.* Department of Computer Sciences, The University of York, United Kingdom, 2002.

[69] David Page. Ilp: Just do it. *Lectures Notes in Artificial Intelligence*, (1866):3–18, 2000.

[70] Francisco J. Palm. Simulación combinada discreta/continua orientada a objetos: Diseño para un lenguaje glider orientado a objetos. Master's thesis, Maestría en Matemática Aplicada a la Ingeniería, Universidad de Los Andes. Mérida. Venezuela., 1999.

[71] Richard N. Pelavin. Planning with simultaneous actions and external events. In J. F. Allen, H. Kautz, R. Pelavin, and J. Tenenberg, editors, *Reasoning About Plans*. Morgan Kauffmann Publishers, Inc., San Mateo, California, 1991. ISBN 1-55860-137-6.

[72] Jean Piaget. *La Psicología de la Inteligencia*. Critica. ISBN 84-7423-980-X.

[73] Jean Piaget and Rolando García. *Hacia una Lógica de Significaciones*. Gedisa, 2 edition, 1997.

[74] David Poole. Logic programming for robot control. In Chris S. Mellish, editor, *Proc. International Joint Conference on Artificial Intelligence*, pages 150–157, San Mateo, California, 1995. Morgan Kaufmann Publishers, Inc.

[75] M. Ramírez, J. Dávila, and Colina E. Intelligent supervision of petroleum processes based on multi-agent systems. *WSEAS Transactions On Systems and Control*, 4(9), September 2009.

[76] Alfredo Ramos. Gide. un ambiente de desarrollo integrado para la plataforma de simulación galatea. Master's thesis, Maestría en Modelado y Simulación, Universidad de Los Andes. Mérida. Venezuela, 2005. Tutores: Tucci Kay, Uzcátegui Mayerlin.

[77] Anand Rao and Michael Georgeff. Bdi agents: From theory to practice. Technical note 56, Australian Artificial Intelligence Institute, April 1995.

[78] Raymond Reiter. A formal account of planning with concurrency, continous time and natural actions. In Ute Sigmund and Michael Thielscher, editors, *Reasoning About Actions and Planning in Complex Environments*, Alexanderstrasse 10, D-64283 Darmstadt, Germany, 1996. Technische Hochschule Darmstadt. (Also at http://www.cs.toronto.edu/~ cogrobo/).

[79] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Inc, 1995.

[80] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs - New Jersey, 1995.

[81] Murray Shanahan. Prediction is deduction but explanation is abduction. In N.S. Sridharan, editor, *Proc. International Joint Conference on Artificial Intelligence*, pages 1055–1060. Morgan Kaufmann, Detroit. Mi, 1989.

[82] Murray Shanahan. Explanation in the situation calculus. In *Proc. International Joint Conference on Artificial Intelligence*, pages 160–165. Morgan Kaufmann, 1993.

[83] E. Y. Shapiro. Algorithmic program debugging. Master's thesis, MIT Press, 1983.

[84] Giorgio Tonella, Miguel Acevedo, Magdiel Ablan, Carlos Domingo, Herbert Hoeger, and Marta Sananes. The use of glider as a tool for the simulation of ecological systems. In M.H. Hamza, editor, *IASTED International Conference*, number ISBN 0-88986-196-X, pages 463–367, Anaheim, California, October 1995. Acta Press.

[85] Giorgio Tonella, Carlos Domingo, Marta Sananes, and Kay Tucci. El lenguaje glider y la computación orientada hacia objeto. In Acta Científica Venezolana, editor, *XLIII Convención Anual ASOVAC*, Mérida, Venezuela, November, 14-19 1993.

[86] Kay A. Tucci. Prototipo del compilador glider en c++. Tesis de Grado. Escuela de Ingeniería de Sistemas. Facultad de Ingeniería. Universidad de Los Andes. Mérida. Venezuela, 1993.

[87] Mayerlin Y. Uzcátegui. Diseño de la plataforma de simulación de sistemas multi-agentes galatea. Master's thesis, Maestría en Computación. ULA, Universidad de Los Andes. Mérida. Venezuela, 2002. Tutor: Dávila, Jacinto.

[88] Yaritza Vargas. Traductor de modelos de simulación galatea a código java, December 2003. Tutores: Tucci, Kay and Uzcátegui, Mayerlin.

[89] J. Vaucher. The simula web site, 1998.

[90] Ludwig Wittgenstein. *Philosophical Investigations.* Blackwell, 1953, 1958, 1967. ISBN 0-631-14670-9 Pbk.

[91] Bernard P. Zeigler. *Theory of modelling and simulation.* Interscience. Jhon Wiley& Sons, New York, 1976.

[92] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modelling and Simulation.* Academic Press, second edition, 2000.