Implementando Múltiples Estrategias de Razonamiento en E-MOBI

Gerardo Rossel[1] Andrea Manna[2]
Facultad de Tecnología Informática - UAI
Departamento de Computación - FCEyN - UBA
(1) grossel@computer.org
(2) amanna@dc.uba.ar

Resumen. E-MOBI es la implementación, en el lenguaje Eiffel, de un modelo multi-paradigma para la incorporación de procesamiento basado en reglas y conocimiento a lenguajes orientados a objetos llamado MOBI (*M*odelo de *OB*jetos *I*nteligentes). E-MOBI soporta múltiple herencia de conocimiento y las principales características arquitecturales de MOBI. En este trabajo, mostramos una extensión de E-MOBI que permite soportar diferentes estrategias de razonamiento sobre la misma base de conocimiento. La extensión presentada permite, a su vez, que diversas instancias de la misma clase utilicen distintos mecanismos de razonamiento. De esta forma, podremos construir agentes inteligentes que soporten múltiples estrategias.

Palabras Claves: Inteligencia Artificial, Ingeniería de Software, Orientación a Objetos, Agentes Inteligentes, Estrategias de Inferencia.

1. Introducción

La variedad de problemas que pueden ser enfrentados mediante sistemas basados en reglas exigen que las estrategias de razonamiento puedan adaptarse para lograr un mejor resultado. Por ejemplo, el razonamiento manejado por el objetivo (*goal-driven*) o encadenamiento hacia atrás, es útil cuando se intentan resolver problemas que pueden modelarse como selección estructurada [11]. Este tipo de problemas tratan de tomar la mejor elección (o las mejores) de un conjunto posibilidades. Los sistemas de diagnóstico, identificación, etc. son problemas que se ubican en esta categoría. Los problemas de configuración, distribución o todos aquellos en los cuales no es posible o sencillo enumerar todas las respuestas posibles, son ideales para tratarlos con un mecanismo de razonamiento de encadenamiento hacia adelante (*data-driven)*. El ejemplo más emblemático de la utilización de encadenamiento hacia adelante, es el sistema de Digital Equipament Corporation XCON, originalmente denominado R1 [13], y que permite establecer configuraciones de computadoras.
Un agente inteligente que cuente con procesamiento basado en reglas puede necesitar, en determinados contextos, aplicar diversas estrategias, e incluso varias simultáneamente. Proveer un mecanismo que permita seleccionar la estrategia de razonamiento más adecuada, aumenta considerablemente la potencia de los agentes.
Tomaremos como base nuestro desarrollo llamado E-MOBI [1], cuya base de conocimiento está implementada como cláusulas de Horn y utiliza *SLD resolution* (Linear resolution for Definite clauses with Selection function)[2][3] como mecanismo de inferencia. Modificaremos el diseño original para posibilitar la creación de agentes inteligentes con múltiples estrategias de razonamiento. Es importante

notar que además es posible, con la misma arquitectura que proponemos, utilizar diversas funciones de selección.
El objetivo de este trabajo, es mostrar una solución que no solamente permita la implementación de múltiples estrategias de razonamiento, sino que además dicha solución sea extensible y pueda evolucionar conforme cambie el contexto y los requerimientos.
La estructura del trabajo es la siguiente: primero mostraremos brevemente los aspectos relevantes de E-MOBI, luego las soluciones de diseño que nos permitirían incorporar diversas estrategias de razonamiento. Posteriormente, mostraremos como es posible que varias instancias de una clase mantengan diversas estrategias y como esto facilita la construcción de agentes inteligentes con múltiples estrategias. Por último, presentamos algunas conclusiones y el trabajo futuro. Todas las explicaciones están acompañadas de fragmentos de código para clarificar los conceptos.

2. E-MOBI

E-MOBI [1] es una implementación de MOBI (Modelo de OBjetos Inteligentes) realizada en el lenguaje Eiffel [4][5][6]. La implementación intenta respetar la esencia de MOBI es decir:

- Conocimiento basado en reglas
- Conocimiento privado para la instancias
- Herencia (múltiple) de conocimiento
- Operaciones de actualización dinámica de la base de conocimiento

La figura 1 muestra un diagrama UML[7] simplificado de la implementación inicial de E-MOBI [1]. Las clases principales están agrupadas en un cluster llamado MOBI en la forma de librería precompilada, facilitando la utilización en diferentes dominios.

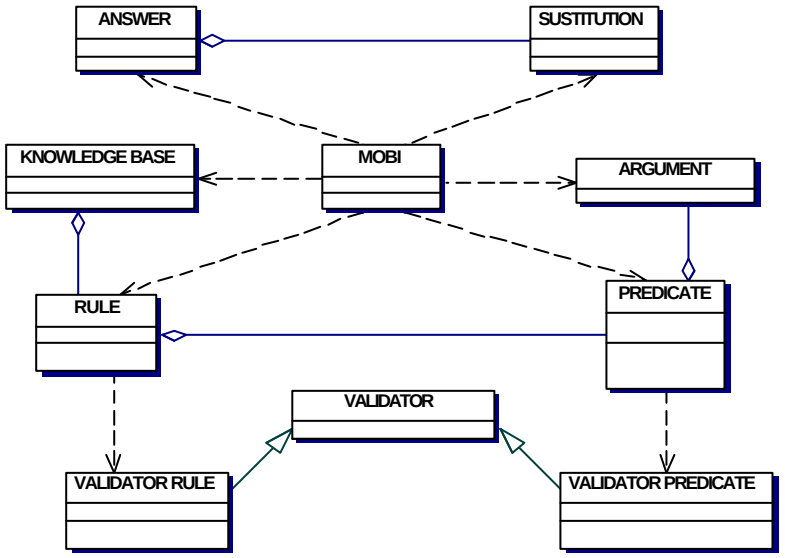

Figura 1. Cluster MOBI

El núcleo central de la implementación está dado por la clase MOBI. En ella se encuentra implementado el algoritmo de evaluación y las rutinas de manipulación de la base de conocimiento. Para comprender su utilización mostramos la *contract format*[5] de la clase.

```
class interface MOBI
inherit
  RTS_SERVER
     export  {NONE} all
  end
feature -- Operations
  eval (sgoal: STRING): ANSWER
                -- Main call to evaluate a predicate on the KB
    require
      ((sgoal) /= (void)) and then (is_predicate (sgoal))

  all_answers (sgoal: STRING): ARRAY [ANSWER]
                        --call to obtain all the possible true answers
                       -- if there is no answer the Array is empty
    require
      ((sgoal) /= (void)) and then (is_predicate (sgoal))
  re_eval: ANSWER
       -- like to semicolon in Prolog,
       -- it requests another answer
  is_predicate(p: STRING): BOOLEAN
         -- it returns true if p conforms to the syntax
         -- established for predicates
  name_kb: STRING
        --file's name of Knowledge Base
  true_answer : ANSWER
       --routine ONCE that returns a answer with the value in True
  false_answer: ANSWER
       --routine ONCE that returns a answer with the value in False
feature -- Operations with Knowledge Base
  save_bc
  expand (r: RULE)
     -- To expand the Knowledge Base with the rule  r is added to the Knowledge Base
  contract (r: RULE)
       -- To contract the Knowledge Base with rule r,
       -- r takes off of the Knowledge Base  consolidate
      -- It adds to the Class Knowledge Base the  instance knowledge
invariant
  knowledgeprivate_necessary: (kb_private) /= (Void)
end interface -- class MOBI
```

La rutina *eval* recibe como parámetro una consulta y devuelve una instancia de la clase ANSWER que representa la primera respuesta encontrada mediante la ejecución del algoritmo de evaluación. Se permite obtener otras respuestas o todas, si es posible. Otras operaciones como *expand* y *contract* permiten la manipulación dinámica de la base de conocimiento privada de las instancias. La rutina *consolidate*, permite que el conocimiento privado de una instancia sea agregado a la base de conocimiento de la clase. E-MOBI soporta además herencia múltiple de conocimiento[1]. La clase MOBI no tiene rutina de creación ya que se trata de una clase abstracta. En general las clases que desean utilizar instancias de clases con conocimiento esperan que éstas tengan el contrato definido en la clase MOBI. Para lograrlo, se puede utilizar el pattern *Adapter o Wrapper* [8][9], cuyo objetivo es convertir la interface de una clase a otra que los clientes esperan. El invariante de clase asegura que las subclases de MOBI deban crear la Base de Conocimiento privada en sus rutinas de creación (constructores). En un articulo anterior [1] detallamos un ejemplo práctico de utilización de E-MOBI.

3. Múltiples estrategias de razonamiento

Nos proponemos extender la implementación de E-MOBI resumida anteriormente, para soportar múltiples estrategias de razonamiento. La rutina *eval* es la responsable de ejecutar los algoritmos correspondientes (actualmente SLD-Resolution con encadenamiento hacia delante). Para soportar múltiples estrategias es necesario encapsular los algoritmos de manera que sean intercambiables. Una solución elegante puede obtenerse utilizando el patrón de diseño *Strategy* [8][9], que nos permite desacoplar los algoritmos de la implementación. En nuestro caso, MOBI actuaría como *Context* y definiríamos una nueva clase abstracta REASONING_STRATEGY que jugaría el rol de *Strategy*.
El agregado de múltiples estrategias nos obliga a modificar la interfaz de la clase MOBI. La implementación original presenta, en la rutina *eval,* un parámetro que es la pregunta o el objetivo. No en todas las estrategias es necesario. Particularmente, en el razonamiento con encadenamiento hacia adelante no existen consultas en el sentido de los algoritmos *goal-driven*. En vez de ello, se aplican las reglas de inferencia a partir de modificaciones en la base de conocimiento tratando de encontrar un resultado. La inferencia se activa, generalmente, a partir de incorporar un nuevo hecho a la base de conocimiento. Como afirman Russell y Norvig, desde el punto de vista del diseño de un agente " *en cada ciclo se incorporan las percepciones a la base de conocimientos y se ejecuta el encadenador hacia adelante..*" [12]. Por ello definimos la rutina *eval* sin parámetros en la clase REASONING_STRATEGY.
Es necesario, a su vez, contar con una estrategia de razonamiento por defecto. Si no se elige ninguna estrategia particular, los objetos aplicarán una por defecto. Esto evitaría obligar a los clientes a elegir una estrategia antes de realizar una pregunta. Si no la tuviéramos, el contrato de la rutina *eval* debería ser el siguiente:

```
eval: ANSWER
   require
     (strategy /= void)
```

Trabajando acorde al diseño por contratos [6], el cliente es el responsable de cumplir con las precondiciones. De todas formas, es más práctico tener una estrategia por defecto (el cliente puede no

saber o no necesitar conocer que estrategia elegir). Para ello, debemos modificar el invariante de clase de la clase MOBI, trasladando la aserción:

```
invariant
    knowledgeprivate_necessary: (kb_private) /= (Void)
    strategydefault_necessary: strategy /= (Void)
```

En la implementación utilizamos un diseño similar al *Null Object Pattern* [10] pero con la diferencia que el *Null Object* es en realidad un *Real Object* con la estrategia de razonamiento por defecto. Además las subclases de REASONING_STRATEGY podrían comportarse como un *Singleton*[8][9], salvo que se quiera hacer uso de ejecución multihilo. En la figura 2, vemos un esquema UML simplificado de lo que acabamos de describir, en el cual se muestran sólo dos estrategias de razonamiento: una lo hace en forma *backward* y otra en forma *forward.*

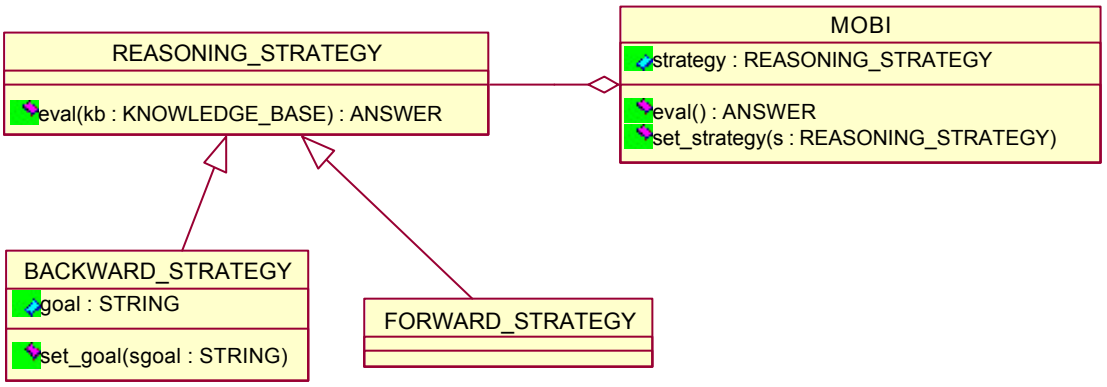

Figura 2. Múltiples Estrategias

Un atributo *strategy* es el encargado de tener una referencia a la estrategia adecuada. Una instancia de la clase FORWARD_STRATEGY podría ser la estrategia por defecto. La clase BACKWARD_STRATEGY cuenta con un atributo *goal* que representa la consulta realizada. En este caso es necesario redefinir el contrato de la rutina *eval* para obligar a asignar un objetivo. Los siguientes fragmentos de código (el contrato de la rutina eval y el invariante de clase) de la clase BACKWARD_STRATEGY lo muestran:

```
....
eval: ANSWER is
    require else
        ((goal) /= (void))
…
invariant
    ( goal = Void  or is_predicate(goal))
```

La clase FORWARD_STRATEGY , que implementa el encadenamiento hacia adelante, básicamente selecciona una regla, aplica la misma y luego repite el proceso hasta que no tenga más reglas para aplicar. La implementación por defecto de la función de selección toma la primer regla que coincide,

pero es posible implementar estrategias más sofisticadas. La distintas funciones de selección se implementan mediante subclases de FORWARD_STRATEGY.
Tal cual se plantea en [8], es posible la implementación del patrón *Strategy* mediante clases genéricas. En este caso, la clase MOBI sería una clase genérica restringida, parametrizada en la estrategia de razonamiento declarada de la siguiente manera:

```
class MOBI [S-> REASONING_STRATEGY]
…
feature
        strategy : S
        ….
end – class MOBI
```

De todas formas, la primera solución es más flexible ya que permite cambiar la estrategia de razonamiento en tiempo de ejecución. Supongamos un agente inteligente que quiere usar dos estrategias de razonamiento distintas. Para ello cuenta con dos referencias a objetos que son instancias de algunas subclases de MOBI, llamémoslos *obj1* y *obj2*. Lo primero es asignar a cada objeto la estrategia adecuada, suponiendo *obj1* con estrategia de encadenamiento hacia atrás y *obj2* con encadenamiento hacia adelante.

```
bs: BACKWARD_STRATEGY

create bs
obj1.set_strategy( bs)
```

Al objeto *obj2* no es necesario asignarle una estrategia ya que la estrategia por defecto es justamente la que queremos que aplique. La invocación, con *ans1* y *ans2* declarados del tipo ANSWER, sería

```
obj1.strategy.set_goal(goal)
ans1 := obj1.eval()
ans2 :=  obj2.eval()
```

Es posible, si el problema y el diseño de la base de conocimiento lo permiten, tener dos instancias de la misma clase cada una de las cuales utilice una estrategia de razonamiento diferente.
La inferencia puede realizarse sobre la misma base de conocimiento (en el caso en que las instancias no cuenten con conocimiento privado) o sobre bases de conocimiento distintas (si el conocimiento privado en cada instancia es distinto).
Es posible implementar agentes con múltiples estrategias pero que cuenten con una sola instancia de una clase con conocimiento. Ello se puede lograr haciendo que el agente exponga varias rutinas que permitan invocar la estrategia más conveniente, cambiando la estrategia dinámicamente sobre el mismo objeto. La implementación puede lograrse de dos formas, utilizando la relación de agregación o de herencia. El siguiente fragmento de clase muestra un ejemplo usando agregación:

```
class
        MULTI_AGENT
create
        make
feature {NONE} -- Initialization
    make is
                do
                  create obj1
                end
feature -- Basic operations

  eval_forward(): ANSWER is
                local
                   fs     : FORWARD_STRATEGY
                do
                   create fs
                   obj1.set_strategy(fs)
                   Result := obj1.eval()
                end

  eval_backward( goal: STRING):ANSWER is
                require
                    ((goal) /= (void)) and then (is_predicate (goal))
                local
                   bs     : BACKWARD_STRATEGY
                do
                   create bs
                   bs.set_goal(goal)
                   obj1.set_strategy(bs)
                   Result := obj1.eval()
                end

   is_predicate( pred: STRING)is
                do
                        Result := obj1.is_predicate
                end

feature {NONE} -- Implementation
    obj1 : EXPERT
end – class MULTI_AGENT
```

La clase MULTI_AGENT mantiene una instancia de la clase EXPERT, que es a su vez una subclase de MOBI y cuenta conocimiento experto sobre algún dominio particular. MULTI_AGENT asigna en cada momento la estrategia de evaluación adecuada. El diagrama UML de la figura 3 muestra las relaciones entre las clases.

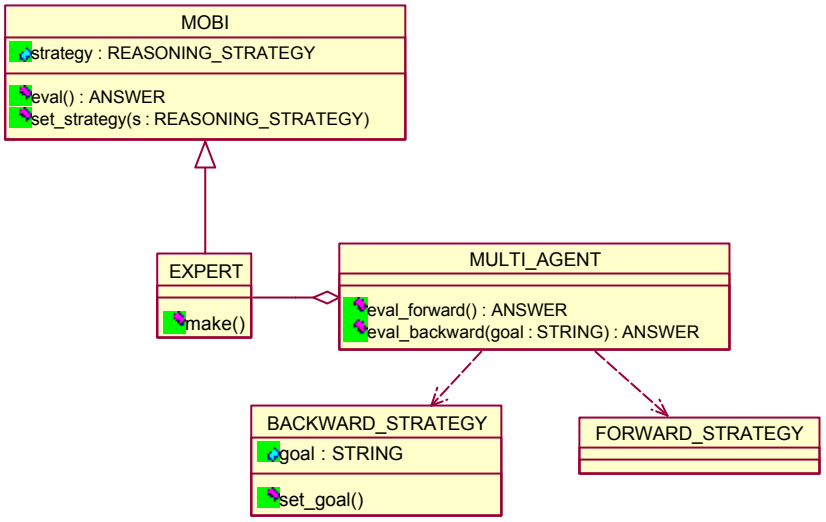

Figura 4. Ejemplo MULTI-AGENT

En la implementación utilizando herencia, simplemente MULTI_AGENT hereda de MOBI (en este caso no existe EXPERT). Para adecuar los nombres y la interfaz al nuevo contexto redefinimos y renombramos *eval* como *eval_backward* e implementamos *eval_forward.* El siguiente fragmento de código lo muestra:

```
class MULTI_AGENT inherit
   MOBI
        rename
            eval as eval_backward
        redefine
            eval_ backward
        end
create
     make
feature { ANY } -- Operations

  eval_forward:  ANSWER is
        do
             create { FORWARD _STRATEGY} strategy
             Precursor
        end


  eval_backward( goal: STRING ): ANSWER is
        require
              ((goal) /= (void)) and then (is_predicate (goal))
        do
               create {BACKWARD_STRATEGY} strategy
```

```
            strategy.set_goal(goal)
            Result := strategy.eval(kb)
        end
.....
.....
.....
end – class MULTI_AGENT
```

Es importante, en este caso, que el diseño de la base de conocimiento permita aplicar las dos estrategias.
Finalmente, es posible que un agente inteligente cuente con una lista de objetos con conocimiento (es decir instancias de subclases de MOBI) e itere sobre ellos aplicando una estrategia de evaluación determinada. Ello se logra utilizando el mecanismo provisto por el lenguaje Eiffel [14] llamado *agent.* Este mismo mecanismo es el que utilizamos para que las estrategias de inferencia devuelvan acciones a desarrollar (particularmente la estrategia de encadenamiento hacia adelante). Hay dos formas de implementar esto. Por un lado modificando la clase FORWARD_STRATEGY de tal manera que luego de la ejecución del motor de inferencia guarde la acción recomendada y por otro lado creando una subclase de ANSWER (ej. ANSWER_WITH_ACTION), que incorpore la acción. Hemos elegido esta ultima opción para lo cual redefinimos *eval* en FORWARD_STRATEGY para que devuelva ANSWER_WITH_ACTION. Esto es posible ya que Eiffel tiene un sistema de tipos que permite la redefinición *covariante* [6][15]. Luego, la clase ANSWER_WITH_ACTION tiene un atributo declarado como:

```
action: ROUTINE[ANY,TUPLE]
```

De esta forma es posible que el cliente del agente decida invocar la acción sugerida cuando lo crea conveniente.

4. Conclusiones y Trabajos Futuros

Hemos presentado una extensión a E-MOBI que permite la construcción de agentes inteligentes con múltiples estrategias de razonamiento. La utilización de patrones de diseño bien conocidos, nos permitió crear una implementación flexible y adaptable. El desacoplamiento de las estrategias nos permite la posibilidad de crear objetos que cuenten con diversas reglas de inferencia y además con diversas estrategias de evaluación.
Actualmente, estamos desarrollando las clases necesarias para agregar nuevas estrategias a las mencionadas en el presente artículo. Además, estamos trabajando para ampliar el E-MOBI con soporte de concurrencia para las distintas estrategias, de forma tal que sea posible disparar en paralelo varios procesos de inferencia. En este aspecto, estamos estudiando la posibilidad de implementar múltiples estrategias *compatibles* que puedan ser ejecutadas concurrentemente sobre una misma instancia.

Referencias

[1] Gerardo Rossel, Andrea Manna. *E-MOBI Smart Object Model and Implementation*, scheduled to be published in the Nov/Dec, 2003 at Journal of Object Technology.

[2] Robinson J.A. *A Machine Oriented Logic Based on the Resolution Principle* Journal of the ACM, 12, 1965.

[3] Loveland, D.W. *Mechanical theorem-proving by model elimination* Journal of the ACM,15. 1968

[4] Bertrand Meyer. *Eiffel the Language Second Edition.* Prentice Hall, 1992.

[5] Bertrand Meyer. *Eiffel the Language Third Edition.* Trabajo en Progreso (versión borrador Septiembre 2002) .

[6] Bertrand Meyer *Object-Oriented Software Construction 2da Edition* Prentice Hall, 1997.

[7] Unified Modeling Language (UML),version 1.5 (formal/03-03-01) En http://www.omg.org

[8] Erich Gamma, Richard Helm, Ralph Johnson, Jhon Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software* Addison-Wesley,1995

[9] Jean-Marc Jézéquel, Michel Train, Christine Mingins. *Design Patterns and Contracts* Addison-Wesley, 1999.

[10] Bobby Woolf. *The null object pattern.* Proceedings of PloP'96 Pattern Languages of Program Design, 1996

[11] Dennis Merritt *Building Expert Systems in Prolog* Amzi! Inc. 2000 (on-line edition)

[12] Stuart Russell, Peter Norvig *Artificial Intelligence: A Modern Approach (2dn Edition)* Prentice Hall (2002)

[13] McDermontt ,D. *R1: A rule-based configurer of computer systems* Artificial Intelligence,19(1) 1982

[14] Bertrand Meyer *Agents, iteration and introspection Chapter 25 Eiffel the Language Third Edition*. Trabajo en Progreso. http://archive.eiffel.com/doc/manuals/language/agent/page.html

[15] Martin Abadi, Luca Cardelli *A Theory of Objects (Monographs in Computer Science)* Springer-Verlag New York Inc., 1996