Implementación Java de un Chequeador de Modelos para un Sistema Multiagente Multimodal

Clara Smith[1], Gastón Fournier[1], and Leonardo Otonelo[1]

[1] Facultad de Informática, Universidad Nacional de La Plata
calle 50 esq. 120, 1900 La Plata, Argentina
{csmith,gfournier,lotonelo}@info.unlp.edu.ar

Resumen. En este trabajo acercamos conceptos de la lógica modal -cuando es usada como herramienta para la modelización de sistemas multiagentes- al lenguaje de programación Java. Construimos un framework para definir estructuras de frames modales y de modelos modales, y chequear en ellos la validez de fórmulas bien formadas escritas en un lenguaje de agentes.

1 Introducción

Sistemas Multi-Agentes (MAS) es un paradigma computacional para modelar sistemas de inteligencia distribuida. En este paradigma, un sistema computacional es visto como una composición de un conjunto de componentes autónomos y heterogéneos, llamados agentes, que interactúan unos con otros. MAS apunta principalmente al modelado, entre otra cosas, de agentes cognitivos o reactivos que coexisten y dependen unos de otros para alcanzar sus objetivos. Los agentes imitan los atributos y capacidades humanas como son descriptos en psicosociología y más ampliamente en las ciencias cognitivas; los agentes pueden "razonar", "adaptarse", "aprender". Las estructuras son comúnmente descriptas con una terminología sociológica: "organización", "comunidad", "institución". En este contexto, las lógicas modales –vistas como extensiones decidibles de la lógica proposicional- son usadas como un enfoque formal para la construcción de MAS.

Uno de nuestros objetivos en este trabajo es mostrar cómo acercar conceptos y herramientas de la lógica modal –cuando es usada como lenguaje para la modelización- a un lenguaje de programación actual y de amplia difusión como Java.

Un chequeador de modelos es un programa tal que dado un modelo para una lógica y una fórmula escrita en el lenguaje de dicha lógica determina si la fórmula es verdadera en el modelo o no lo es.

Implementamos en Java el chequeador de modelos descripto en [1]. Proveemos, para las estructuras del chequeador, un framework lo suficientemente abstracto que puede ser instanciado con cualquier tipo de frame multimodal y que está implementado en un lenguaje de programación bien conocido como Java. Existen implementaciones de chequeadores de modelos (como MCK de la U. de Wales, MCMAS, del Imperial College, Mocha, de las U. de Berkeley, Pensylvannia y NYU) pero ellos, en su mayoría, fueron diseñados para ciertos tipos específicos de lógicas temporales (LTL Linear Temporal Logic y CTL Computational Tree Logic) y lógicas epistémicas.

El algoritmo que programamos tiene como referencia la lógica base descripta en [2,3]: una lógica multimodal multiagente útil para modelar sistemas donde intervienen múltiples agentes que interactúan para intentar cumplir sus objetivos. Dicha lógica

combina los operadores modales normales bien conocidos Bel_a x, Int_a x, y $Goal_a$ x (usados para modelar "el agente *a* cree x", "el agente *a* tiene la intención de que x sea verdadero" y "el agente *a* tiene el objetivo x"), con operadores no normales de la forma $Does_a$ x (para referirse a acciones intencionales como "el agente *a* lleva a cabo x exitosamente") [4].

En los trabajos [2,3] se estudia la transferencia de propiedades de lógicas pequeñas (o "de propósitos especiales" para modelar creencias, intenciones, etc) a la combinación multimodal multiagente resultante de ellas. Se analizan las propiedades de completitud y decidibilidad, concluyendo que ambas son preservadas por la combinación, en el sentido de conservatividad dado en [5,6]. Este resultado sirve de base para la búsqueda de algoritmos computacionales correctos para esa lógica combinada. En [1] ya se había descripto la exploración de un código fuente para implementaciones del chequeador, pero usando lenguajes declarativos, como Prolog. Nuestra motivación, a partir de los trabajos [1,2,3], consiste en lograr una implementación computacional flexible hecha esta vez en un lenguaje imperativo bien conocido, ampliamente difundido y usado en ámbitos comerciales como lo es Java.

El resto del artículo se organiza como sigue. En la Sección 2 presentamos el algoritmo chequeador de modelos que sirve de base para la construcción de un framework Java para MAS modales, framework que finalmente implementamos. En la Sección 3 presentamos el código Java para los módulos más relevantes del framework, con algunas discusiones interesantes referidas al código. La Sección 4 contiene nuestras conclusiones.

2 El Chequeador de Modelos

En lógica modal un modelo se define como un par M = (F, V) donde F = (W, R) es un frame, W es un conjunto de puntos o *mundos posibles*, R es un conjunto de relaciones de accesibilidad entre mundos y V es una función de valuación que asigna a cada proposición p del lenguaje modal un subconjunto V(p) de W [7].

Las lógicas multimodales son, por lo general, una combinación de lógicas monomodales específicas que da lugar a teorías complejas. La lógica para la que implementamos el chequeador de modelos se llama N(Does) [1]; es una lógica normal multimodal multiagente (N) que se combinó con operadores no normales (los operadores Does de acción, uno para cada agente) mediante una técnica de fibrado [8,9].

Intuitivamente, fibrar dos lógicas consiste en organizarlas en dos niveles. Es por eso que el algoritmo chequeador de modelos funciona así: dado un modelo para la lógica y dada una fórmula ϕ del lenguaje de la lógica, el chequeador parsea la fórmula y "navega" por la parte normal (de Kripke) del modelo; cuando encuentra una subfórmula no normal, el chequeador pasa a otr nivel y "navega" por un modelo especial llamado de Scott-Montague[1]. Es decir, en la lógica base hay una parte modal normal en un nivel y una parte "no normal" en otro nivel.

Técnicamente, se convierten en (nuevas) letras proposicionales todas las subfórmulas no normales que hay en la fórmula a chequear, logrando que el chequeo final se haga sobre un modelo único, "aplanado". Para lograr esto hay que chequear la

[1] Podemos generalizar la semántica tradicional de Kripke de la siguiente manera. En lugar de tener una colección de mundos conectados a un mundo w mediante una relación R, consideramos un conjunto de colecciones de mundos conectados a w [10]. Estas colecciones son los vecindarios (neighbourhoods) de w. Formalmente, un frame de Scott-Montague es un par ordenado <W, N> donde W es un conjunto de mundos y N es una función total que asigna para cada w en W un conjunto de subconjuntos de W (los vecindarios de w).

validez de cada fórmula no normal en su correspondiente modelo de Scott-Montague, transformarla a una nueva letra proposicional si es verdadera, y así tener un modelo único sobre el cual evaluar.

Tenemos entonces tres procedimientos principales:

```
Function MC_N(Does) (<A,W,B_i,G_i,I_i,V´,{d_i}>,φ)
input: modelo <A,W,B_i,G_i,I_i,V´,{d_i}>, fórmula φ
computar MM_LDoes(φ)
for every α ∈ MM_LDoes(φ)
     i := identificar el agente que aparece en α
     if (MC_Does(d_i(w),α) = true) then
            V'(w):= V'(w) ∪ {p_α}
            construir φ'
return MC_N (<A,W,B_i,G_i,I_i,V´,{d_i}>,φ')

Function MC_Does(d_i(w),α)
input: modelo Scott-Montague, subfórmula monolítica maximal α
while quedan neighbourhoods sin chequear en d_i(w)
     n_k = set n_i ∈ d_i(w)
     for every w ∈ n_k
            if α ∉ v(w) then return false
     return true

Function MC_N (<A,W,B_i,G_i,I_i,V´,{d_i}>,φ')
input: un modelo M=<A,W,B_i,G_i,I_i,V´,{d_i}>,φ')
for every w ∈ W
     if check(<A,W,B_i,G_i,I_i,V´>, φ')
            return w
return false
```

La descripción de estos tres módulos del chequeador es la siguiente: la función $MC_{N(Does)}$ (llamada así por "model checker para la lógica N(Does)") primero computa el conjunto $MM_{\mathcal{L}Does(\phi)}$ de subfórmulas monolíticas maximales de ϕ[2]. Para cada una de éstas, identifica el agente que está llevando a cabo la acción con el Does. Luego, averigua los mundos en los que dicha acción es llevada a cabo satisfactoriamente. Para esto, la función MC_{Does} es invocada con un modelo Scott-Montague como parámetro.

Sea ϕ una fórmula y $MM_{\mathcal{L}Does(\phi)}$ el conjunto de subfórmulas monolíticas maximales de ϕ pertenecientes al lenguaje $\mathcal{L}_{Does}$. Sea ϕ' la fórmula obtenida reemplazando cada subfórmula $\alpha \in MM_{\mathcal{L}Does(\phi)}$ por una nueva letra proposicional p_α [FG94]. La nueva fórmula ϕ' fue construida sin las modalidades Does, ya que éstas se reemplazaron, y será uno de los inputs del chequeador MC_N.

Notar entonces que el chequeador principal MC_N es, en términos generales, la implementación de la función de valuación sobre el modelo multimodal resultante de haber "aplastado" los modelos no normales de Scott-Montague.

[2] Las fórmulas monolíticas son las que empiezan con un operador modal. Ej: $Does_x(A)$. $\mathcal{L}_{Does(\phi)}$ se refiere al lenguaje de la lógica restringido a las fórmulas que tienen solo operadores no normales

3 Representación en Java de Frames y Modelos Modales

La tarea de codificar en Java el chequeador de la Sección 2 no es directa. Una lógica modal usada como lenguaje declarativo de representación de conocimiento por un lado, un lenguaje de programación imperativo por otro. Sin embargo, encontramos que la organización modal de *mundos posibles* cuadra con la visión orientada a objetos de Java: los mundos son objetos relativamente abstractos, están relacionados entre sí por una relación dirigida de asociación, y hay enunciados que son ciertos en cada mundo. Estos conceptos de objeto con estado interno y relacionado con otros objetos son naturales para el paradigma orientado a objetos: cada mundo posible w de W en un frame F es un objeto, con propiedades, que se relaciona con otros mundos.

Definimos algunas clases como sigue. La clase *World* describe mundos posibles que constituyen el dominio W de un frame de la lógica modal. Cada mundo tiene asociado un conjunto de literales que son verdaderos en dicho mundo, representando así la función de valuación V de un modelo. Definimos la clase *Literal* para describir hechos proposicionales. El atributo name de cada instancia de esta clase es su identificador. Por ejemplo, la instancia con name='p' se corresponde con el hecho p. Definimos la clase abstracta *Formula* que representa la noción de verdad local [7, Definición 1.20]. Definimos la clase *Modality,* donde cada instancia de *Modality* tiene una lista de relaciones. Las relaciones de Kripke (*KripkeRelationship*), asocian un agente y un mundo con una colección de mundos adyacentes, mientras que las relaciones de Scott-Montague (*ScottMontagueRelationship*) asocian un agente y un mundo con una colección de vecindarios (conjuntos de mundos representados por la clase *Neighborhood*). Finalmente, representamos agentes, mundos y hechos definiendo las clases *Agent*, *World* y *Literal* respectivamente.

Definimos la clase *Frame*, formada por un conjunto de mundos posibles y por los agentes que interactúan en el sistema. Así, los agentes están en los mundos. Para definir las relaciones de accesibilidad entre mundos decidimos que éstas estén "cableadas" según la definición de cada modalidad. De este modo, *Frame* no queda acoplada a una implementación concreta ni a un conjunto particular de modalidades. El cómo recorrer cada relación de accesibilidad (es decir, cómo moverse por el "cableado" del frame) es información que maneja cada modalidad. Para crear un frame instanciamos, entonces, como surge naturalmente de la definición de frame, los mundos que integran el universo y las relaciones entre esos mundos.

Naturalmente, imitando la realidad, cada agente tiene su propio esquema de mundos posibles. Es decir, un agente a puede tener Bel-relacionados los mundos w_1 y w_2 pero otro agente a_2 podrá no tenerlos Bel-relacionados. Así, cada agente tiene su propio grafo de relaciones cableado en el frame.

Evaluación de fórmulas. El mensaje aceptado por la clase *Formula*: *eval(w: World): Boolean* retorna el valor verdadero cuando la instancia de la fórmula ϕ es verdadera en el mundo w.

Usamos el patrón Builder [11] para construir una instancia de un frame con sus agentes, mundos y su función de valuación, esto es, para construir la estructura propiamente dicha de un sistema particular. El método *withAgents* crea una instancia de *FrameBuilder* y define los nombres de los agentes del frame, luego retorna la instancia de *FrameBuilder*. Con el método *withWorlds* definimos los mundos, con *withLiteralValid* definimos en qué mundos son válidos qué literales (tomando como primer parámetro el nombre del literal, los siguientes parámetros son los nombres de los mundos en los cuales es válido el literal):
Instanciación de un frame:

```
  frame1 = FrameBuilder.withAgents("a1","a2")
       .withWorlds("w1","w2","w3")
       .withLiteralValid("A","w1","w2")
       .withLiteralValid("B","w2","w3").build();
```

Donde w1, w2, y w3 son mundos; a1 y a2 son agentes; A y B son proposiciones.
Seguidamente, para cada modalidad normal definimos sus arcos dentro del frame. Así, los mundos creados se van relacionando entre sí por medio de diferentes relaciones de accesibilidad según la semántica particular de cada modalidad. La clase *Modality* tiene una lista de relaciones. Las modalidades normales instancian dicha lista con una lista de relaciones de Kripke. Al instanciar una relación de Kripke, indicamos un agente, un mundo y luego la lista de mundos accesibles desde ese par (agente, mundo):

Relaciones de accesibilidad de operadores normales:

```
belModality = new Bel();
belModality.setRelationships(Lists.newArrayList(
  new KripkeRelationship("a1", w1, w2, w3),
  new KripkeRelationship("a1", w2, w1)));
intModality = new Int();
intModality.setRelationships(Lists.newArrayList(
    new KripkeRelationship("a1", w1, w3),
    new KripkeRelationship("a1", w2, w3)));
goalModality = new Goal();
goalModality.setRelationships(Lists.newArrayList(
    new KripkeRelationship("a1", w3, w1),
    new KripkeRelationship("a1", w2, w1)));
```

Para las modalidades no normales (los operadores Does) instanciamos una lista de relaciones de tipo Scott-Montague: dado un agente y un mundo, se definen los vecindarios para el par (agente, mundo). Para definir los vecindarios usamos la clase *Neighborhood*, que representa un conjunto de mundos:

```
Relaciones de accesibilidad para operadores no normales:
doesModality = new Does();
doesModality.setRelationships(Lists.newArrayList(
     new ScottMontagueRelationship("a2", w1,
       new Neighborhood(w2, w3))));
```

Definimos la clase *ModelChecker* que conoce un objeto de tipo *Frame* y puede evaluar una fórmula en el frame. La lógica de como evaluar la fórmula queda en la fórmula, por lo que el método que evalúa una fórmula en el frame está definido así:

```
public List<World>check(Formula f) {
    List<World> list = new ArrayList<World>();
    for (World world : frame.getWorlds()) {
        if (check(f, world)){
             list.add(world);
        }
    }
    return list;
}

public Boolean check(Formula f, World world) {
```

```
   return f.eval(world);
}
```

ModelChecker acepta dos mensajes. El primero recibe una fórmula a chequear y devuelve un listado de los mundos en los que la fórmula es verdadera. Si todos los w ∈ W del frame están dentro del conjunto resultante, la fórmula es una verdad global en el modelo [ver 7, 1.21]. El segundo mensaje recibe como parámetros una fórmula y un mundo. Retorna verdadero si la fórmula es satisfactible en el mundo y falso en caso contrario [ver 7, 1.20].

La semántica de las fórmulas básicas proposicionales es la booleana tradicional. Para las fórmulas modales proposicionales delegamos el conocimiento de las relaciones de accesibilidad en las modalidades (clase *Modality*). Por lo tanto, la evaluación de una fórmula modal se delega en la modalidad, como sigue:

ModalFormula.eval

```
public Boolean eval(World world) {
   return modality.evalWith(formula, world, agent);
}
```

Debe quedar claro entonces que las modalidades normales "corren" sobre modelos de Kripke y todas respetan –por definición- la semántica de necesidad; se diferencian entre unas y otras por sus axiomas: para nosotros entonces las diferencias entre ellas estarán dadas a nivel "cableado" de los frames [ver 7, Cap 5]. Los axiomas definen entonces, como es de esperar, la estructura, la parte "estática" del frame. Esto es porque hay una correlación entre los axiomas que definen la modalidad y las relaciones de accesibilidad de la misma, estableciendo la clase de frames en los cuales la modalidad es válida (Tabla 1).

Axioma	Semántica
□A → ◊A	Serial
□A → □ □A	Transitivo
◊A → □ ◊A	Euclideano
□A → ◊A	Reflexivo

Tabla 1. Algunos axiomas modales

La siguiente es la implementación de la evaluación de las modalidades normales:

NormalModality.eval

```
public Boolean evalWith(Formula formula, World world, Agent
    agent) {
    // relationships es la lista de relaciones de
    accesibilidad para la modalidad
    for (KripkeRelationship rel: relationships) {
    if (rel.getAgent().equals(agent) &&
rel.getWorld().equals(world)){
        for (World adyacent: rel.getAdyacents()) {
          if (!formula.eval(adyacent)){
            // si hay un adyacente donde no es verdadera
          retornar falso
              return  false;
           }
```

```
        }
        // Si es verdadero en todos los adyacentes retornar
          verdadero
        return  true;
      }
    }
    if  (!formula.eval(world)){
      // para ser consistentes con la semántica de Bel, Int y
  Goal generalization (R2)
      // deberíamos retornar verdadero si la fórmula es
  verdadera en el mundo
      LOGGER.warn( this + ": Inconsistency between semantics,
        wrong Frame");
    }
    return true;
  }
```

Las fórmulas no normales de la lógica se evalúan en un modelo de Scott-Montague:

NonNormalModality.eval

```
public Boolean evalWith(Formula formula, World world, Agent
    agent) {
    // relationships es la lista de relaciones de accesibilidad
    para la modalidad
   for (ScottMontagueRelationship rel: relationships) {
     if (rel.getAgent().equals(agent) &&
     rel.getWorld().equals(world)){
        for (Neighborhood  neighbor: rel.getNeighbours()){
          Iterator <World> it = neighbor.getWorlds().iterator();
          boolean resultInNeighborhood = neigh-
              bor.getWorlds().size () > 0;
          while (resultInNeighborhood && it.hasNext()) {
              World worldInNeighborhood = it.next();
              resultInNeighborhood = formula. ev-
            al(worldInNeighborhood);
          }
          if  (resultInNeighborhood){
            // si encontramos un vecindario en el cual es
            verdadero
              return true;
            }
          }
       // si no encontramos un vecindario en el cual es
       verdadero, return false ;
       }
     }
     return true;
  }
```

4 Conclusiones

Hemos implementado en Java un chequeador de modelos para una lógica multimodal multiagente. Codificamos, en una estructura compleja, la semántica de mundos posibles y la semántica de Scott-Montague. El chequeador fue implementado de manera modular previendo que sea extensible a otras modalidades distintas de las de

la lógica base. Esto permite flexibilidad para explorar el diseño de nuevos chequeadores y extensiones a otras diferentes combinaciones de lógicas.

Hemos realizado con éxito algunas pruebas de performance con frames de pocos mundos y pocos agentes, obteniendo rendimientos aceptables. Si bien el objeto del presente trabajo es presentar el chequeador de modelos en cuanto vincula elementos de la lógica modal usada como lenguaje declarativo de representación de conocimiento con un lenguaje de programación imperativo –con influencia del paradigma de objetos- nos proponemos optimizar la implementación computacional concreta mejorando el código del chequeador y reduciendo el uso de memoria, analizando especialmente algunos métodos en [12] para reducir la complejidad limitando la profundidad de las fórmulas y restringir el número de átomos. También analizamos programar algunos módulos directamente en C o C++.

Referencias

1. Agustín Ambrosio, Leandro Mendoza. Completitud e Implementación de modalidades en MAS. Tesis de Licenciatura, UNLP, 2011.
2. Clara Smith, Antonino Rotolo. Collective trust and normative agents. Logic Journal of the IGPL, 18(1):195–213, 2010.
3. Agustíin Ambrossio Leandro Mendoza. Combination of normal and non-normal modal logics for Normative Multi-Agent Systems, Anales del EST 2011, ISSN 1850-2946, pp. 171—185.
4. Guido Governatori, Antonino Rotolo. On the Axiomatization of Elgesem´s Theory of Agency and Ability. In Advances in Modal Logic, U. of Manchester, Dept. Computer Science, pp 130-144, 2004.
5. Joao Rasga, Amílcar Sernadas, Cristina Sernadas. Importing logics: Soundness and completeness preservation. Studia Logica, 101(1):117–155, 2013.
6. Joao Rasga, Amílcar Sernadas, Cristina Sernadas. Fibring as biporting subsumes asymmetric combinations. Studia Logica, 102(5):1041–1074, 2014.
7. Patrick Blackburn, Maarten de Rijke, Yde Venema. Modal Logic. Cambridge University Press, Cambridge, England, 2001.
8. Marcelo Finger and Dov Gabbay. Combining temporal logic systems. Notre Dame Journal of Formal Logic, 37, 1994.
9. M. Franceschet, A. Montanari, and M. de Rijke. Model checking for combined logics with an application to mobile systems. 11, 2004.
10. Bengt Hansson and Peter Gaerdenfors. A guide to intensional semantics. In Bengt Hansson, editor, Modality, Morality, and Other Problems of Sense and Nonsense: Essays Dedicated to Soren Hallden. Liber Forlag, 1973.
11. Design Patterns: Elements of Reusable Object-Oriented Software (Professional Computing). E. Gamma et al., Addison Wesley, 1998.
12. Marcin Dziubinski, Rineke Verbrugge, and Barbara Dunin-Keplicz. Complexity issues in multiagent logics. *Fundamenta Informatica*, 2007.