Implementación de Chequeadores de Modelos para MAS

Autores:

A.C. Gastón Fournier

A.C. Leonardo Otonelo

Facultad de Informática, UNLP, Argentina

Directora:

Dra. Smith, Clara

Facultad de Informática, UNLP, Argentina

Implementación de Chequeadores de Modelos para MAS

Gastón Fournier[1] y Leonardo Otonelo[1]

Facultad de Informática, UNLP, Argentina

Resumen En este trabajo analizamos la combinación de lógicas modales por medio del fibrado e implementamos un chequeador de modelos para el fibrado *N(Does)* usando conceptos de programación orientada a objetos y definiendo las estructuras necesarias para la representación de las fórmulas modales y del frame para el fibrado. Analizamos su funcionamiento asumiendo que sólo es necesaria la semántica de mundos posibles para evaluar una fórmula en un frame, pero encontramos ejemplos en los que son necesarias realizar ciertas suposiciones muy fuertes sobre la forma del grafo de relaciones de accesibilidad del frame, que pueden ser muy costosas de analizar. Presentamos la estructura para el fibrado y una implementación del chequeador de modelos usando la semántica de mundos posibles para los operadores modales normales *Bel*, *Int* y *Goal*, además del operador no normal *Does*.
Creamos un ejemplo con una instancia de frame y una fórmula modal que demuestra que no alcanza usar la semántica de mundos posibles si no hacemos ciertas suposiciones sobre la estructura del frame.

1. Introducción

Integrar lógicas de propósitos especiales (con poder de expresión limitado) para lograr una lógica resultante de mayor poder de expresión [AMdNdR99], es un tema actualmente en expansión. Este concepto de combinación de lógicas para propósitos de modelado esta inspirado en las ideas de modularidad con el fin de permitir la integración de diferentes tipos de información. Un aspecto importante a tener en cuenta al realizar una combinación es determinar si las propiedades individuales de cada lógica se transfieren correctamente a la lógica resultante.

Las lógicas modales o multi-modales son, por lo general, una combinación de lógicas específicas que dan lugar a teorías complejas con varios operadores modales *normales* y eventualmente *no normales*. Una lógica modal es normal si contiene el axioma de distribución (usualmente llamado K) y es cerrada bajo la regla de generalización [Che80,BdRV01]. Mientras que una lógica es no normal si no satisface el axioma K [Elg97,Che80].

La estructura de un chequeador de modelos para una combinación de lógicas mediante fibrado es presentada en [SMAR2010,AM2011], combinando los chequeadores de cada una de las lógicas intervinientes. En estos trabajos se toma como lógica base la definida en [SR2010] y se demuestra que la combinación

de las lógicas es completa y decidible mediante la propiedad de modelo finito a través de filtraciones, lo que asegura la existencia de algoritmos e implementaciones computacionales correctas. Ambrossio y Mendoza presentan un algoritmo para el chequeador de modelos para la lógica combinada en un alto nivel de abstración y en los capítulos 6 y 7 de [AM2011b], se dan 2 implementaciones, una en Prolog [CM03] (usando programación declarativa) y otra en SPINdle [GHP09,Lam10] (programación declarativa basada en lógica rebatible).

2. Definiciones útiles

2.1. Modelo de Kripke

Cada modelo de Kripke para el lenguaje $\mathscr{L}$ se encuentra compuesto por un conjunto de mundos, un conjunto de relaciones de accesibilidad entre los mundos, y una valuación de los átomos proposicionales, de la siguiente manera.

Un modelo de Kripke para la teoría de trabajo es una tupla
$\mathfrak{M} = (W, \{B_i : i \in \mathcal{A}\}, \{G_i : i \in \mathcal{A}\}, \{I_i : i \in \mathcal{A}\}, Val)$, tal que

- W es un conjunto de estados (o mundos) posibles.
- Para todo $i \in \mathcal{A}$, se cumple que B_i, G_i, $I_i \subseteq W \times W$. Estas son las relaciones de accesibilidad para cada agente con respecto a *Beliefs (creencias)*, *Goals (objetivos)*, e *Intentions (intenciones)*, respectivamente. Por ejemplo, $(s,t) \in B_i$ significa que t es una alternativa epistémica para el agente i en el estado s.
- $Val : \mathcal{P} \times W \rightarrow \{0,1\}$ es la función de valuación que asigna valores de verdad a las proposiciones atómicas en estados del modelo. Es decir, dada una letra $p \in P$ y un estado $w \in W$, nos dice qsi p es verdadera o no en el estado w, indicando 1 ó 0, respectivamente. P es el conjunto de proposiciones atómicas p, q, r, ..., etc.

Un frame de Kripke $\mathfrak{F}$ es como un modelo de Kripke, pero sin la función de evaluación. En este punto, es posible definir las condiciones de verdad correspondiente al lenguaje $\mathscr{L}$. La expresión $\mathfrak{M}, s \vDash \varphi$ es leida como "la formula φ es válida en el mundo s en el modelo $\mathfrak{M}$".

2.2. Modelo de Scott-Montague

De acuerdo con Hansson y Gardenfors [HG73], 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. Estas colecciones son los vecindarios (neighbourhoods) de w. Formalmente, un frame de Scott-Montague es un par órdenado $\langle W, N \rangle$ 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). Un modelo de Scott-Montague es una terna $\langle W, N, V \rangle$ donde $\langle W, N \rangle$ es un frame de Scott-Montague y V es una función de valuación

definida como la de los frames de Kripke, excepto para $\Box A$: es verdadero en w sii el conjunto de elementos de W donde A es verdadero es uno de los conjuntos en $N(w)$, un vecindario de w.

2.3. Chequeador de modelos

Un chequeador de modelos (model checker) es un programa que resuelve el problema del chequeo de modelos. El problema global de chequeo de modelos para $N(Does)$ consiste en chequear si, dada una fbf φ, y dado un modelo $\mathfrak{M}$ para $N(Does)$ existe un $w \in W$ tal que $\mathfrak{M}, w \models \varphi$.

3. Fibrado de lógicas normales y no normales

En el trabajo original [SR2010], la modalidad *Does* se integra al frame definido por $\mathcal{F}$, creando un nuevo frame $\mathfrak{F}$, como los modelos de Kripke (estandar, sin mundos especiales) no soportan modalidades no-normales los autores proponen solucionar este problema mediante la tecnica de fibrado [FG94]. Como la modalidad *Does* es aplicada a constantes proposicionales que representan acciones o comportamientos, por ejemplo, $Does_x(Pagar)$ (significa "el agente x paga"). Ello restringe el conjunto de fórmulas a usar, $Does_i(Does_j(Pagar))$ (significa "el agente i hace que el agente j pague") no puede escribirse en el sistema original. Intuitivamente un fibrado de dos lógicas L y M (anotamos $L(M)$)consiste en organizarlas en dos niveles, el nivel superior para la lógica L, y en segundo la lógica M. Según la definición de fibrado dada por Finger y Gabbay [FG94], (también véase [FMDR04]) una lógica basada en $\mathfrak{F}$ puede ser vista como una combinación donde la lógica modal normal es puesta sobre una lógica no normal. La parte no normal también es multi-modal ya que, existe una modalidad $Does_i$ para cada agente i.

Con respecto a la noción de fibrado usada, hay que notar que el método de fibrado combina las lógicas de una manera relativamente simple: no existen axiomas especiales para la combinación ni interacciones complejas entre los operadores modales. Esto permite asegurar resultados de adecuación, completitud y decibilidad para la lógica resultante si ambas lógicas involucradas lo son.

Como base para la construcción de nuestro chequeador de modelos utilizaremos el pseudo-código mostrado en [AM2011b]. Llamamos $\mathbf{N}$ a la restricción de $\mathfrak{F}$ a su parte normal y *Does* a la restricción de $\mathfrak{F}$ a su parte no normal. Se demuestra que el fibrado $N(Does)$ es completo ([AM2011b, cap 3]).

4. Implementación

Este es el pseudo-código del chequeador de modelos provisto en [AM2011b] para resolver el problema del chequeo de modelos para la lógica $N(Does)$ en el cual nos basamos para hacer nuestra implementación:

Sea φ una fórmula y MM$\mathscr{L}_{Does}(\varphi)$ el conjunto de *sub-fórmulas monolíticas maximales* de φ pertenecientes a $\mathscr{L}_{Does}$. Sea φ' la N-fórmula obtenida reemplazando cada sub-fórmula $\alpha \in$ MM$\mathscr{L}_{Does}(\varphi)$ por una nueva letra proposicional p_α.

Function MC$_{N(Does)}$ $((A,\ W,\ B_i,\ G_i,\ I_i,\ V',\ \{\mathrm{d}_i\}),\ \varphi)$
    input: un modelo fibrado $\mathfrak{M}$ y una fórmula $\varphi \in \mathscr{L}_{N(Does)}$
    computar MM$\mathscr{L}_{Does}(\varphi)$
    for every $\alpha \in MM\mathscr{L}_{Does}(\varphi)$
        i := identificar el agente involucrado en α
        for every $w \in W$
            if $(MC_{Does}(d_i(w),\ \alpha) = true)$ *then*
                $V'(w)\ :=\ V'(w) \bigcup \{p_\alpha\}$ */*fuzzling*/*

    construir φ'*/* sistemáticamente reemplazamos las variables generadas */*
    return $MC_N((A,\ W,\ B_i,\ G_i,\ I_i,\ V',\ \{d_i\}),\ \varphi')$*;/*llama al chequeador normal con* φ'**/*

Function $MC_{Does}(d_i(w),\ \alpha)$
    input: un modelo de Scott-Montague de estructura ,
    y una sub-fórmula monolítica maximal α
    while quedan neighbourhoods sin chequear en $d_i(w)$
        n_k *= set* $n_i \in d_i(w)$ */**n_k *itera sobre el conjunto de neighbourhoods*/*
        for every $w \in n_k$
            if $\alpha \notin v(w)$ *then return false*
    return true

Function $MC_N((A,\ W,\ B_i,\ G_i,\ I_i,\ V',\ d_i),\ \varphi')$
    input: un modelo $\mathfrak{M}$ = (A, W, B$_i$, G$_i$, I$_i$, V', d$_i$) y una fórmula φ'
    for every $w \in W$
        if check$((A,\ w,\ B_i,\ G_i,\ I_i,\ V'),\ \varphi')$
            return w
    return false

Function check$((A,\ w,\ B_i,\ G_i,\ I_i,\ V'),\ \alpha)$
    case on the form of α
        $\alpha = p_{\alpha'}$:
            if $p_{\alpha'} \notin V'(w)$
                return false
        $\alpha = \neg\alpha'$:
            if check$((A,\ w,\ B_i,\ G_i,\ I_i,\ V'),\ \alpha')$
                return false
        $\alpha = \alpha_1 \wedge \alpha_2$:
            if not check$((A,\ w,\ B_i,\ G_i,\ I_i,\ V'),\ \alpha_1)$ *or*
            or not check$((A,\ w,\ B_i,\ G_i,\ I_i,\ V'),\ \alpha_2)$
                return false
        $\alpha = \alpha_1 \vee \alpha_2$:

```
                if not check((A, w, B_i, G_i, I_i, V'), α_1) and
                and not check((A, w, B_i, G_i, I_i, V'), α_2)
                    return false
            α = BEL_i(α') :
                for each v such that wB_iv
                    if not check((A, v, B_i, G_i, I_i, V'), α')
                        return false
            α = GOAL_i(α') :
                for each v such that wG_iv
                    if not check((A, v, B_i, G_i, I_i, V'), α')
                        return false
            α = INT_i(α') :
                for each v such that wI_iv
                    if not check((A, v, B_i, G_i, I_i, V'), α')
                        return false
            others : return false
        return true
```

Estos procedimientos actúan de la siguiente manera. Dado un modelo fibrado y una fórmula φ, el chequedaor $\mathrm{MC}_{N(Does)}$ primero computa el conjunto $\mathrm{MM}\mathscr{L}_{Does}(\varphi)$ de sub-fórmulas monolíticas maximales de φ. Para cada una de estas, el chequeador identifica cuál es el agente que está llevando a cabo la acción. Luego, el chequeador establece los mundos donde esta acción es llevada a cabo satisfactoriamente. Para ello, MC_{Does} es invocado con un modelo de Scott-Montague $d_i(w)$ como parámetro (donde d_i es $\eta = \langle A, W_D, \{D_i\}, v\rangle$ donde $\{D_i\}_{i\in A}$ son las relaciones de accesibilidad para los operadores normales y $w \in W_D$ [AM2011b])

MC_{Does} es pseudo-código para la función de valuación v para *Does*: testea si existe un neighborhood n_i de w donde α es verdadera. De ser así, una nueva letra proposicional p_α es agregada a $V'(w)$ para registrar el éxito del actuar, utilizando el método de fuzzling (donde las fórmulas del sistema base son sustitutidas por átomos en el sistema superior, [FG94]). Esto es, si la acción es exitosa (es decir Does_i es verdadera) lo reemplazamos por una nueva variable proposicional verdadera. Finalmente, antes de llamar al chequeador normal MC_N, la nueva fórmula φ' es construida sin las modalidades Does, ya que estas han sido reemplazadas en la etapa de fuzzling.

El chequeador MC_N es pseudo-código de la función de valuación de la lógica N evaluándose recursivamente sobre las sub-fórmulas $\alpha \in \varphi$ mediante la función *check*.

5. Estructura

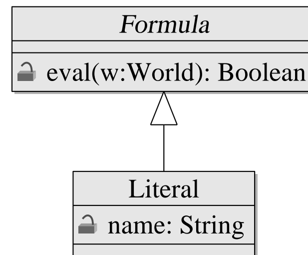
5.1. Representación de fórmulas

Como base para la representación de las fórmulas, partimos de la representación inductiva usada en [AM2011b](6.4.1):

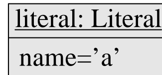
- Un hecho (proposición atómica) es una fórmula.
- Si F_1 y F_2 son fórmulas, también lo son: $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \rightarrow F_2$, $F_1 \leftrightarrow F_2$ y $\neg F_1$
- Si F es una fórmula, $\mathcal{A}$ un conjunto finito de agentes, i un agente tal que $i \in \mathcal{A}$ y G un subconjunto de agentes tal que $G \subseteq \mathcal{A}$, entonces las siguientes modalidades también son fórmulas:
  - Modalidades epistémicas: $BEL(i, F)$, $E\text{-}BEL_G(F)$, $C\text{-}BEL_G(F)$
  - Modalidades motivacionales: $GOAL(i, F)$, $INT(i, F)$, $E\text{-}INT_G(F)$, $M\text{-}INT_G(F)$
  - Modalidades *no normales* de acción: $DOES(i, F)$

Para implementar las fórmulas definidas anteriormente, utilizaremos las siguientes estructuras definidas en notación UML[UML]:

Para describir hechos (proposiciones que son verdaderas en algún mundo del modelo), utilizamos instancias de la clase *Literal*. Su atributo *name* se corresponde con la proposición que representa.

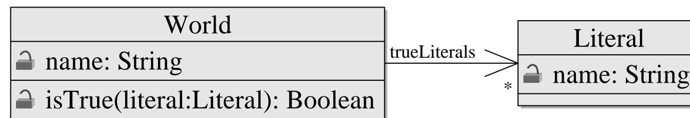


donde por ejemplo:

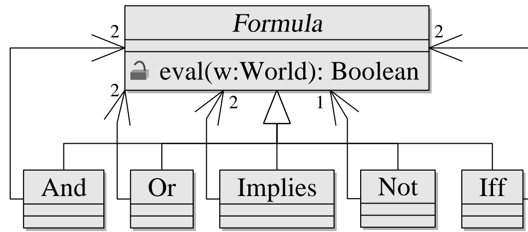


representa el hecho a.

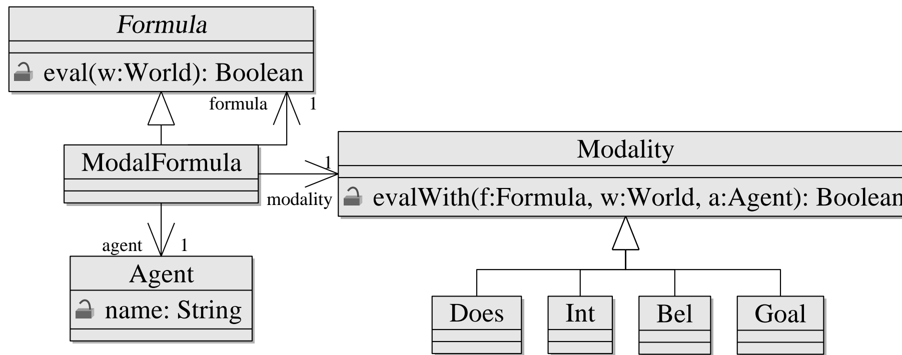
Para representar la función de valuación del modelo, asociamos a cada mundo una lista de hechos que son verdaderos en dicho mundo. Un literal es considerado verdadero en un mundo si está instanciado y será falso en caso contrario. Análogamente cada *World* responderá que un *Literal* es verdadero si está asociado a dicho mundo y responderá falso en caso contrario.



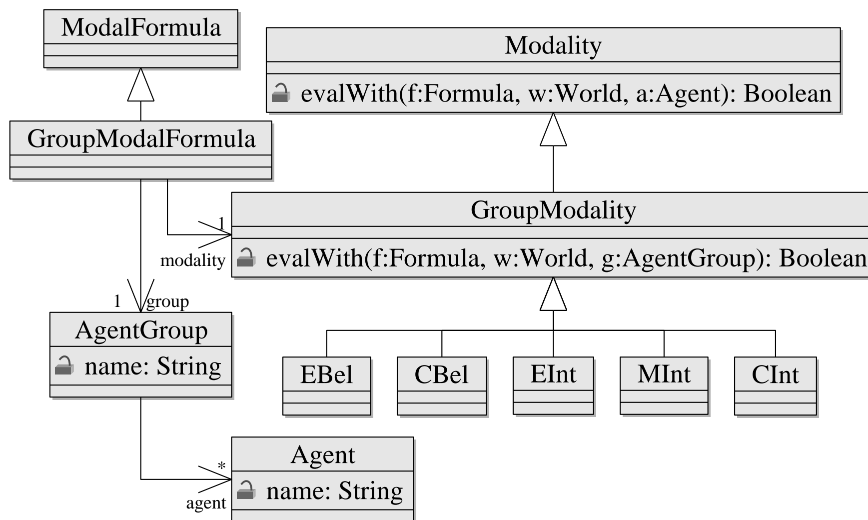
Para las fórmulas booleanas definimos una clase por cada fórmula, donde las instancias de estas clases representan fórmulas concretas



Para las fórmulas modales definimos una *ModalFormula* que tiene asociado un agente (*Agent*), una modalidad (*Modality*) y una fórmula sobre la cual se aplica la modalidad. Las instancias de estas clases representan fórmulas modales concretas. Por cada operador modal definimos una subclase de *Modality*



Análogamente para las fórmulas modales colectivas definimos una extensión de *ModalFormula* y *Modality* que permiten tener (y evaluar respectivamente), una fórmula modal con un grupo de agentes:



5.2. Representación de frames y modelos

Para definir nuestras estructuras de datos, partimos de las definiciones hechas en el capítulo 6 de [AM2011b] y las adaptamos al paradigma de programación orientada a objetos. Representamos los agentes, mundos y hechos utilizando instancias de las clases *Agent*, *World* y *Literal* respectivamente.

- Definimos la clase *Agent* para describir agentes. El atributo *name* de cada instancia lo usamos como identificador del agente. Por ejemplo, la instancia con *name*='$a1$' representa al agente $a1$. Todas las instancias de *Agent* representan el conjunto $\mathcal{A}$.
- Definimos la clase *World* para describir mundos o situaciones del frame. El atributo *name* de cada instancia lo usamos como identificador. Por ejemplo, la instancia con *name*='w_1' representa al mundo w_1. Todas las instancias de *World* representan el conjunto W. Cada mundo tiene asociado un conjunto de literales que son verdaderos en dicho mundo.
- Definimos la clase *Literal* para describir hechos. El atributo *name* de cada instancia lo usamos como identificador. Por ejemplo, la instancia con *name*='a' se corresponde con el hecho a.

Para crear un frame instanciamos los mundos y las relaciones que lo componen. Cada agente puede tener relaciones distintas de cada mundo por cada modalidad. Es decir, el agente A puede tener Bel-relacionados los mundos w_1 y w_2 pero otro agente $A2$ podría no tenerlos Bel-relacionados. Por lo tanto cada agente tendrá un grafo de relaciones por cada modalidad.

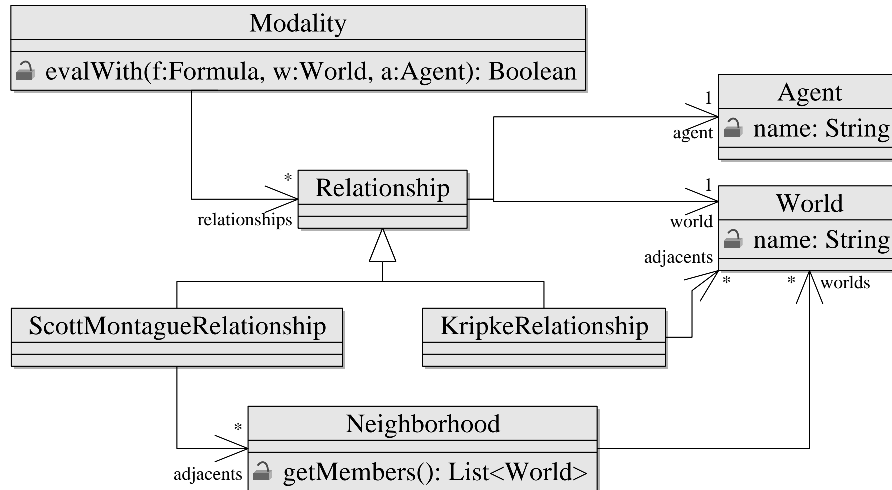
Definimos la clase *Frame* que conoce todos los mundos y agentes del frame. Para las relaciones de accesibilidad decidimos que cada modalidad conozca sus relaciones de accesibilidad de manera que la clase *Frame* no quede acoplada a una implementación concreta (con un conjunto de modalidades específicas). Esto nos permite reusar la definición de frame para otro conjunto de modalidades.

Como contrapartida, esta definición nos obliga a mantener una única instancia de cada modalidad por cada frame para asegurarnos que todas las fórmulas modales tienen acceso a las mismas relaciones de accesibilidad, y lo implementamos usando el patrón Singleton [GHJV94].

Esta forma de modelar las relaciones nos permite extender el chequeador con otras modalidades que podrían llegar a tener relaciones de accesibilidad distintas a las planteadas con anterioridad. También dejamos la responsabilidad de conocer como recorrer esas relaciones a la modalidad de manera que la modalidad define si usa una semántica de necesidad, de posibilidad o alguna otra semántica particular.

Dada una modalidad cualquiera, y sean:

- A una instancia de la clase *Agent* que representa un agente.
- W una instancia de la clase *World* que representa un mundo del frame.
- $W(A)$ un conjunto de instancias de la clase *World* que representa la lista de mundos adyacentes.

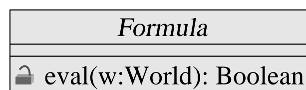


Cada instancia de *Modality* (son *Singleton* [GHJV94]), tiene una lista de relaciones. Es decir la modalidad *Bel* conoce todas las relaciones del frame (en particular va a conocer relaciones de Kripke). Las relaciones de Kripke (*KripkeRelationship*), asocian un agente y un mundo con una lista de mundos adyacentes, mientras que las relaciones de Scott-Montague (*ScottMontagueRelationship*) asocian un agente y un mundo con vecindarios (conjuntos de mundos representados por la clase *Neighborhood*).

Esta implementación tiene la ventaja de ser eficiente, dado que el frame no tiene que buscar entre las relaciones aquellas que se correspondan con cada modalidad, sino que las relaciones son directamente accesibles desde la modalidad. Por otro lado la estructura de datos que mantiene las relaciones, puede ser optimizada en particular para cada modalidad con el objetivo de encontrar rápida y eficientemente una relación particular para un par (*Agent*, *World*).

6. Evaluación de fórmulas

Al representar una fórmula como un objeto, estamos asumiendo que el objeto fórmula representa tanto su estructura como su comportamiento, en otras palabras, su semántica. Por este motivo decidimos que la evaluación de las fórmulas será responsabilidad de cada una, y como dicha evaluación depende del mundo en que se evalúe, deberá recibir como parámetro para evaluarse un objeto *World*. De esta manera nuestra clase *Formula* tendrá un método abstracto *eval* que deberá ser implementado por sus subclases dando la semántica específica de cada una, retornando *True* si la fórmula es verdadera en el mundo y *False* en caso contrario:

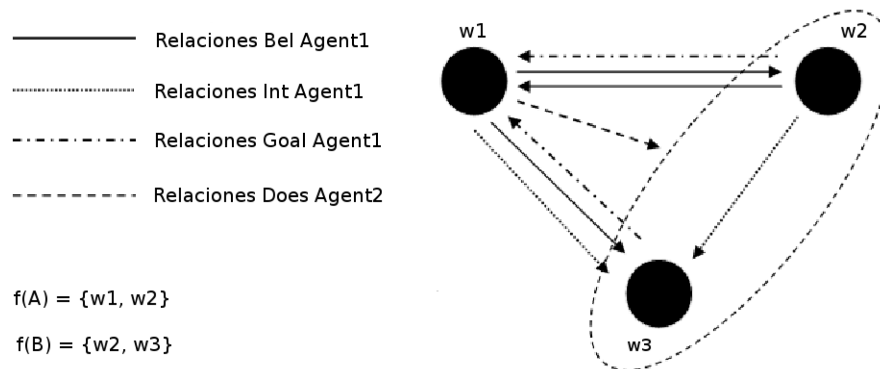


6.1. Implementación del chequeador de modelos

Para poder utilizar el chequeador de modelos, debemos primero construir un modelo, y para esto necesitamos indicar:

- los agentes involucrados,
- los mundos del frame,
- las relaciones entre los mundos definidos y,
- la función de valuación que determina los hechos que son verdaderos en cada mundo

Tomando como ejemplo siguiente modelo:



Hacemos uso del patrón Builder [GHJV94] para construir, de manera más simple y legible, un Frame con sus agentes, mundos y función de valuación. El método *withAgents* crea una instancia de *FrameBuilder* y define los nombres de los agentes del frame, luego retorna la instancia de *FrameBuilder*. Los métodos siguientes modifican *FrameBuilder* y retornan la instancia modificada, hasta el método build que toma todos los datos cargados en *FrameBuilder* y con ellos crea la instancia del frame. Con el método *withWorlds* definimos los nombres de los mundos y con *withLiteralValid* se define en que mundos son validos los literales, tomando como primer parámetro el nombre del literal y 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();
```

Para definir las relaciones de accesibilidad debemos recuperar los mundos creados en el frame, para hacer referencia a esos objetos.

Recuperamos los mundos creados

```
World w1 = frame1.getWorld("w1");
World w2 = frame1.getWorld("w2");
World w3 = frame1.getWorld("w3");
```

Como dijimos, cada modalidad conoce las relaciones de accesibilidad. Para esto, la clase *Modality* de la cual heredan las modalidades, tiene una lista de relaciones. Las modalidades normales instancian esta 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 de agente, mundo.

Relaciones de accesibilidad normales

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

Para las modalidades no normales, la lista de relaciones se instancia con una lista de relaciones de Scott-Montague que dado un agente y un mundo, definen cuales son los vecindarios accesibles para el par agente, mundo. Para definir las vecindades, hacemos uso de la clase *Neighborhood* que representa un conjunto de mundos.

Relaciones de accesibilidad no normales

```
doesModality = new Does();
doesModality.setRelationships(Lists.newArrayList(
    new ScottMontagueRelationship("a2", w1,
        new Neighborhood(w2, w3))));
```

Para evaluar una fórmula en un modelo, nos basamos en la definición de verdad [DVDK07, Definition 2.4]. No evaluamos la parte sintáctica dada por los axiomas ya que estos están definiendo una clase de frames en los cuales dichos axiomas son válidos, de acuerdo a la semántica de las modalidades que utilizamos.

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 de la siguiente manera:

ModelChecker

```
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);
}
```

La función devuelve un listado de mundos en los cuales la fórmula es verdadera. También se puede evaluar si la fórmula es verdadera en un mundo particular, en este caso devolviendo *true* o *false* si la misma es verdadera o falsa (respectivamente). La evaluación de las fórmulas es dependiente del mundo en el cual se evalúan, por lo que la función *eval* recibe como parámetro el mundo en el cual se evalúa.

Como dijimos la semántica de las fórmulas booleanas es la tradicional. A modo de ejemplo mostramos la implementación para la clase *And*, siendo similar la implementación en las otras clases:

And.eval

```
public Boolean eval(World world) {
  return left.eval(world) && right.eval(world);
}
```

Para las fórmulas modales, delegamos el conocimiento de las relaciones de accesibilidad en las modalidades (*Modality*). Por lo tanto la evaluación de una fórmula modal se delega en la modalidad:

ModalFormula.eval

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

En nuestro caso, las modalidades normales están modeladas con modelos de Kripke y por lo tanto su semántica es similar. Las diferencias entre las mismas están dadas por los axiomas, que definen estructuras de grafos distintas para cada modalidad.

La evaluación de una modalidad la implementamos de la siguiente manera:

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 false;
}
```

Para fórmulas no normales, en nuestro caso *Does*, evaluamos la fórmula en un modelo de Scott-Montague:

NormalModality.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 = neighbor.
            getWorlds().size() > 0;
        while (resultInNeighborhood && it.hasNext()) {
          World worldInNeighborhood = it.next();
          resultInNeighborhood = formula.eval(
              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 false;
}
```

Para las fórmulas modales colectivas, nos basamos en las definiciones semánticas dadas en [DKV02, Sections 3 and 4]

- Definiciones semánticas de *E-BEL* y *C-BEL*:
  $\mathfrak{M}, s \vDash E\text{-}BEL_G(\varphi)$ sii $\forall i \in G$, $\mathfrak{M}, s \vDash BEL(i,\, \varphi)$
  $\mathfrak{M}, s \vDash C\text{-}BEL_G(\varphi)$ sii $\forall t$ que sea BEL-alcanzable desde s: $\mathfrak{M}, t \vDash \varphi$
- Definiciones semánticas de *E-INT* y *M-INT*:
  $\mathfrak{M}, s \vDash E\text{-}INT_G(\varphi)$ sii $\forall i \in G$, $\mathfrak{M}, s \vDash INT(i,\, \varphi)$
  $\mathfrak{M}, s \vDash M\text{-}INT_G(\varphi)$ sii $\forall t$ que sea INT-alcanzable desde s: $\mathfrak{M}, t \vDash \varphi$
- Para *C-INT* no hay una definición semántica, pero la podemos derivar de la semántica de *M-INT* y *C-BEL*:
  $\mathfrak{M}, s \vDash C\text{-}INT_G(\varphi)$ sii $M\text{-}INT_G(\varphi) \wedge C\text{-}BEL_G(M\text{-}INT_G(\varphi))$

Dado que cada una tiene una semántica particular, implementamos la evaluación de la modalidad en cada subclase. Para las relaciones de accesibilidad de las modalidades, cada modalidad colectiva conoce a la modalidad individual (es decir, *E-BEL* conoce a *BEL*). A continuación mostramos la implementación para *E-BEL*:

E-BEL.eval

```
public Boolean evalWith(Formula formula, World world,
    AgentGroup group) {
  Boolean result = false;
  for (Agent a : group.getAgents()) {
    result = belModality.evalWith(formula, world, a);
  }
  return result;
}
```

6.2. Ejemplo de aplicación

Para probar el evaluador, nos preguntamos si $\mathfrak{M}$, $w_1 \vDash Bel(a1,\, p)$ donde $\mathfrak{M}$ = ($\{w_1,\, w_2\}$, $(w_1,\, w_2) \in B_i$, $\text{Val}(p) = \{w_1\}$)

Haciendo correr nuestro chequeador con la evaluación semántica con la fórmula $Bel(a1,\, p)$ y el modelo $\mathfrak{M}$, obtuvimos que $\mathfrak{M}$, $w_1 \nvDash Bel(a1,\, p)$ (ver apéndice)

Esto es semánticamente correcto dado que w_2 es Bel-adyacente a w_1 y por la función de valuación $\mathfrak{M}$, $w_2 \nvDash p$

Sin embargo, si hacemos un chequeo sintáctico usando los axiomas, se ve que por **R2** (Belief Generalization [DKV02]) vale $\mathfrak{M}, w_1 \vDash Bel(a1, p)$ ya que $\mathfrak{M}, w_1 \vDash p$.

Esta diferencia surge porque el frame sobre el cual probamos no cumple con los requisitos para *Bel*, es decir, no pertenece a la clase de frames *KD45*. Pero esto no quita que en algunos casos la respuesta que demos pueda ser correcta y se corresponda con la respuesta de un chequeo sintáctico. Si el frame cumple con las condiciones semánticas dadas por los axiomas modales K, D, 4 y 5, nuestro chequeador dará una respuesta que se corresponde con la semántica de los operadores modales, pero si el frame no es adecuado, no podemos afirmar que nuestra respuesta sea correcta.

En nuestro caso, asumimos que el frame que recibimos está bien formado y cumple con los requisitos de la clase de frames correspondientes a cada modalidad. Otro enfoque implica verificar que el frame cumple las propiedades deseadas para cada modalidad antes de hacer el chequeo, pero esto es un problema que es de un orden de magnitud mayor al de nuestro chequeador de modelos. Otra opción es completar el frame o la función de valuación usando los axiomas para construir un frame válido. En el ejemplo anterior se ve que es posible modificar la función de valuación original por $\mathrm{Val}(p) = \{w_1, w_2\}$ para que $w_2 \vDash p$, pero en este caso estaríamos resolviendo un problema diferente al que nos plantean, salvo que pudiéramos verificar que el nuevo frame es equivalente al original.

7. Conclusión y trabajo futuro

Hemos implementado un chequeador de modelos para el fibrado *N(Does)* usando únicamente la semántica de mundos posibles y analizado su funcionamiento con algunos ejemplos, llegando a la conclusión de que es necesario asumir ciertas precondiciones para su correcto funcionamiento, sin las cuales las respuestas pueden ser incorrectas.

Si bien esto parece ser una precondición muy ambiciosa, tiene sentido hacerla ya que si analizamos otro tipo de modalidades como ser las temporales, es más evidente el hecho de que si el frame no cumple ciertas restricciones, no se corresponde con el modelo temporal conocido.

Como trabajo queremos poder decir de manera eficiente si el frame cumple las propiedades requeridas para que el chequeador de una respuesta correcta. Dichas propiedades son dependientes de las modalidades que estamos evaluando. En particular fuimos capaces de detectar una inconsistencia con la regla **R2** para modalidades normales, pero deberíamos ser capaces o bien de comprobar las propiedades o de detectar inconsistencias. Cualquiera de las dos alternativas nos dan la posibilidad de garantizar las respuestas de nuestro chequeador.

Referencias

[AM2011] Agustín Ambrossio Leandro Mendoza. Combination of normal and non-normal modal logic, 2011.

[AM2011b] Agustín Ambrossio Leandro Mendoza. Completitud e implementación de modalidades en mas. Licenciate's thesis, UNLP, 2011.

[AMdNdR99] Carlos Areces, Christof Monz, Hans De Nivelle, and Maarten De Rijke. *The Guarded Fragment: Ins and Outs*. Vossiuspers. Amsterdam University Press, 1999.

[BdRV01] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, Cambridge, England, 2001.

[CM03] C. S. Mellish and W. F. Clocksin. *Programming in Prolog: Using the ISO Standard*. Springer Verlag, 2003.

[Che80] B. F. Chellas. *Modal logic, an introduction*. Cambridge University Press, 1980.

[DKV02] Dunin-Keplicz and Verbrugge. Collective intentions. *Fundamenta Informatica*, 2002.

[DVDK07] Marcin Dziubinski, Rineke Verbrugge, and Barbara Dunin-Keplicz. Complexity issues in multiagent logics. *Fundam. Inform*, 2007.

[Elg97] Dag Elgesem. The modal logic of agency. *Nordic Journal of Philosophical Logic*, 2:1–46, 1997.

[FG94] Marcelo Finger and Dov Gabbay. Combining temporal logic systems. *Notre Dame Journal of Formal Logic*, 37, 1994.

[FMDR04] M. Franceschet, A. Montanari, and M. de Rijke. Model checking for combined logics with an application to mobile systems. 11, 2004.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[GHP09] *Rule Interchange and Applications, International Symposium, Rule ML 2009, Las Vegas, Nevada, USA , November 5-7, 2009. Proceedings*, volume 5858 of *Lecture Notes in Computer Science*. Springer, 2009.

[HG73] Bengt Hansson and Peter Gäerdenfors. A guide to intensional semantics. In Bengt Hansson, editor, *Modality, Morality, and Other Problems of Sense and Nonsense: Essays Dedicated to Sören Halldén*. Liber Förlag, 1973.

[Lam10] Ho-Pun Lam. Spindle - user guide. Technical report, NICTA QRL, 2010.

[SMAR2010] Clara Smith, Agustín Ambrossio, Leandro Mendoza, and Antonino Rotolo. Combinations of normal and non-normal modal logics for modeling collective trust in normative mas. *AICOL XXV IVR, Forthcoming LNAI*, 2011.

[SR2010] Clara Smith and Antonino Rotolo. Collective trust and normative agents. *Logic Journal of the IGPL*, 18(1):195–213, 2010.

[UML] http://www.uml.org.

A. Apéndice: Ejemplos de uso del chequeador de modelos

Para probar el chequeador de modelos implementamos un caso de test que mostramos a continuación:

Test para el model checker

```
public class ModelCheckerTest {
  private static final Logger LOGGER = Logger.getLogger
     (ModelCheckerTest.class);
  private Frame frame1;
```

```
private ModelChecker mc;
private Bel belModality;
private Int intModality;
private Goal goalModality;
private Does doesModality;

@Before
public void setup() {
  frame1 = FrameBuilder.withAgents("a1", "a2")
      .withWorlds("w1", "w2", "w3").withLiteralValid(
         "A", "w1", "w2")
      .withLiteralValid("B", "w2", "w3").build();

  World w1 = frame1.getWorld("w1");
  World w2 = frame1.getWorld("w2");
  World w3 = frame1.getWorld("w3");

  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)));

  doesModality = new Does();
  doesModality.setRelationships(Lists.newArrayList(
      new ScottMontagueRelationship("a1", w1,
          new Neighborhood(w2, w3)),
      new ScottMontagueRelationship("a1", w3,
          new Neighborhood(w1, w2))));

  frame1.setModalities(Lists.newArrayList(belModality
     , doesModality));
  this.mc = new ModelChecker();
```

```
  }

  @Test
  public void
      bela1AIsTrueInW1ByBeliefGeneralizationInFrame1()
      {
    mc.setFrame(frame1);
    // check A in w1
    Literal a = new Literal("A");
    World w1 = frame1.getWorld("w1");
    Assert.assertTrue("A should be true in w1", mc.
        check(a, w1));
    // check Bel(A) in w1
    Formula belA1A = new ModalFormula(belModality, "a1"
        , a);
    List<World> validInWorlds = mc.check(belA1A);
    LOGGER.info(belA1A + " valid in: " + validInWorlds)
        ;
    Assert.assertTrue("Bel(a1, A) should be true in w1
        (R2)", validInWorlds.contains(w1));
  }
}
```

El test falla por lo expuesto en 6.2 y la salida del test es:

```
INFO [NormalModality]: Adyacents for World(w1) Agent(a1)
are [World(w2), World(w3)]
WARN [NormalModality]: Bel: Inconsistency between semantics,
we have a wrong Frame
INFO [NormalModality]: Adyacents for World(w2) Agent(a1)
are [World(w1)]
INFO [ModelCheckerTest]: Bel [agent=Agent(a1), formula=A]
valid in: [World(w2)]
```

Como se ve en la salida del test, fuimos capaces de detectar una inconsistencia entre la semántica procedural que analizamos y la semántica dada por las reglas de derivación. En principio esto nos dice que el frame no pertenece a la clase *KD45*, ya que de pertenecer debería existir una correspondencia entre ambas.