TESINA DE LICENCIATURA

Titulo: Implementación de chequeadores de modelos para MAS
Autores: Fournier, Gastón – Otonelo, Leonardo
Director: Smith, Clara

Carrera: Licenciatura en informática

Resumen

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 en un medio ambiente.
Las lógicas modales son usadas, como un enfoque formal para la construcción de MAS. Se utilizan elementos de forma (entidades, reglas, axiomas, etc) para la descripción, inferencia y planificación de los agentes y su comportamiento en un ámbito de aplicación.
Nuestro trabajo consiste en implementar un chequeador de modelos en un lenguaje orientado a objetos, tratando de acercar los resultados obtenidos en los trabajos previos a una implementación flexible y en un lenguaje usado actualmente en ámbitos comerciales, que permita integrar la solución a sistemas existentes y a su vez que sea extensible a otras lógicas.

Palabras Claves

Sistemas multiagentes, agentes, lógica, modal, chequeador, modelo

Trabajos Realizados

Estudio de lógicas modales y combinación de lógicas modales.
Investigación de chequeadores de modelos.
Implementación del chequeador de modelos.
Presentación en JAIIO 2012 del paper "Implementación de chequeadores de modelos para MAS" (41 JAIIO - EST 2012 - ISSN: 1850-2946 - Página 302-320)
Presentación en IRIT Université Paul Sabatier, Toulouse

Conclusiones

Hemos implementado en un lenguaje orientado a objetos, un chequeador de modelos para la lógica N(Does) extensible a otras lógicas modales multiagente, usando la semántica de mundos posibles (o de Kripke) y la semántica de Scott-Montague. Si bien existen otras implementaciones de chequeadores de modelos, en su mayoría fueron diseñados para un tipo específico de lógica, generalmente para modalidades temporales o epistémicas y no se adaptaban fácilmente a nuestras necesidades.

Trabajos Futuros

Definir un lenguaje para la lógica de confianza colectiva, que sirva para ingresar un frame y una fórmula de N(Does) al chequeador.
Optimizar nuestro chequeador para reducir el uso de memoria y tiempo de ejecución.
Proveer una implementación distribuida del chequeador para aprovechar las facilidades actuales de cloud computing.

Fecha de la presentación: Diciembre 2014

Implementación de chequeadores de modelos para MAS

Facultad de Informática, UNLP, Argentina

Gastón Fournier - Leonardo Otonelo

15 de Noviembre 2014

Agradecemos a Benoit Gaudou y Frederic Amblard (IRIT, Universidad de Tolouse) por recibirnos e intercambiar con nosotros sus desarrollos y permitirnos presentarles nuestro trabajo.

A nuestra directora Clara Smith por su constancia y compromiso en el acompañamiento y guía de nuestra tesis.

A Agustín Ambrossio y Leandro Mendoza por ayudarnos con el estudio del tema y por sus comentarios y correcciones a nuestro trabajo.

A nuestras familias por acompañarnos y estar siempre con nosotros.

Índice general

Capítulo 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 en un medio ambiente.

MAS apunta principalmente al modelado, entre otra cosas, de agentes cognitivo o reactivo que interactuan en un medio ambiente donde coexisten y dependen uno de otro para alcanzar sus objetivos. Los agentes imitan los atributos y capacidades humanas como son descriptos en la psicosociología y más ampliamente en las ciencias cognitivas; los agentes pueden "razonar", "adaptarse", "aprender". Las estructuras son comunmente descriptas con una terminologia sociológica: "organización", "comunidad", "institución".

En este ámbito, las lógicas modales son usadas, como un enfoque *formal* para la construcción de sistemas multi-agente (MAS). Se utilizan elementos de forma (entidades, reglas, axiomas, etc) para la descripción, inferencia y planificación de los agentes y su comportamiento en un ámbito de aplicación.

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. El concepto de combinación de lógicas está inspirado en las ideas de modularidad con el fin de permitir la integración de diferentes tipos de información en la formalización/especificación de un

sistema.

La lógica modal es usada como una herramienta para razonar acerca del tiempo, creencias, sistemas computacionales, necesidad, posibilidad y muchas otras aplicaciones. También es bien aceptada en filosofía y en la comunidad de Inteligencia Artificial como herramienta para el análisis de conceptos. A pesar de la diversidad de estas aplicaciones, los lenguajes proposicionales modales tienen la particularidad de ser fácilmente explicados por medio de estructuras relacionales que representan la estructura lógica del sistema modal.

Las lógicas multi-modales son una combinación de lógicas específicas que dan lugar a teorías complejas con varios operadores modales *normales* y eventualmente *no normales*. 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.

Una lógica modal es normal si contiene el axioma de distribución (usualmente llamado K: $\Box(\phi \rightarrow \psi) \rightarrow (\Box\phi \rightarrow \Box\psi)$) 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].

Algunos operadores modales normales utilizados en el área de MAS son, por ejemplo, el de creencias (Bel_x), objetivos ($Goal_x$), intenciones (Int_x) y obligaciones (O). Otros operadores, como por ejemplo el de acción ($Does_x$) son no normales.

Un modelo para una lógica modal es una estructura relacional (llamado *Frame*) junto con una función de valuación.

El problema del chequeo de modelos consiste en determinar si una fórmula es verdadera en un modelo dado. La estructura de un chequeador de modelos para una combinación de lógicas es presentada en [FMDR04], combinando los chequeadores de cada una de las lógicas intervinientes.

Por medio de la representación abstracta de estos elementos, en este trabajo de grado, analizamos y diseñamos un chequeador de modelos desde un punto de vista más general, que nos da la posibilidad de adaptarlo y extenderlo para distintos requerimientos funcionales o computacionales. Esta generalización, muy común en la ingeniería de software, creemos que puede ser un aporte de valor para el campo de los chequeadores de modelos, en

particular para la combinación de lógicas modales, sirviendo de base para generar extensiones o adaptaciones específicas.

Los lenguajes orientados a objetos, por su parte, permiten representar de manera natural esta modularización usando mecanismos propios de estos lenguajes, como ser la herencia y el polimorfismo, lo cual hace de ellos una buena alternativa para alcanzar estos objetivos.

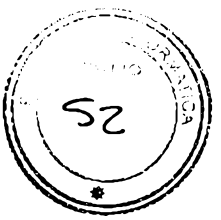
En este trabajo tomamos como lógica base la definida en [SR2010, AM2011], una lógica multi-modal multi-agente que permite modelar conceptos legales. La misma combina los operadores normales *Bel*, *Int* y *Goal*, con el operador no normal *Does*. En [SR2010, AM2011] se hace un estudio sobre la transferencia de propiedades de las lógicas particulares a la combinación. Se analizan las propiedades de completitud y decidibilidad, concluyendo que ambas son preservadas por la combinación, lo que asegura la existencia de algoritmos e implementaciones computacionales correctas para la lógica.

Ambrossio y Mendoza, en su tesis de grado [AM2011b], presentan un algoritmo para el chequeador de modelos para la lógica combinada en un alto nivel de abstracción. Luego, en los capítulos 6 y 7 de su tesis, dan dos implementaciones, una en Prolog [CM03] (usando programación declarativa) y otra en SPINdle [GHP09, Lam10] (programación declarativa basada en lógica rebatible).

Si bien los lenguajes de programación declarativos hacen fácil la implementación de estos algoritmos, la tarea no es tan sencilla con otros lenguajes de programación, lo que hace que estos tipos de chequeos sean rara vez usados en sistemas comerciales que usan, modernamente, lenguajes orientados a objetos.

Nuestro trabajo consiste en implementar un chequeador de modelos en un lenguaje orientado a objetos, tratando de acercar los resultados obtenidos en los trabajos previos a una implementación flexible y en un lenguaje usado habitualmente en ámbitos comerciales, que permita integrar la solución a sistemas existentes y a su vez que sea extensible a otras lógicas.

Comenzamos esta tesis haciendo un repaso de los conceptos generales asociados a las lógicas modales y a la combinación de lógicas (capítulos 2 y 3).



En el capítulo 4 diseñamos una solución para la representación de las fórmulas, el modelo y el chequeador, y lo describimos usando el lenguaje UML. Para el diseño nos basamos en las representaciones semánticas de las modalidades intervinientes. Luego realizamos la implementación de dicho diseño usando el lenguaje de programación Java.

En el capítulo 5 analizamos los tiempos de ejecución del algoritmos propuesto y exploramos las posibilidades de mejoras.

En el capítulo 6 hacemos un repaso sobre algunos chequeadores de modelos para otros tipos de lógicas y comentamos sobre otras herramientas aptas para desarrollos similares a los realizados en esta tesis.

Finalmente, en el capítulo 7, exponemos las concluciones que obtuvimos luego de desarrollar el trabajo.

Capítulo 2

Lógicas modales

2.1. Lógica modal

Los lenguajes de la lógica proposicional modal son lenguajes proposicionales a los que se les han añadido cierto tipo de operadores, usualmente llamados "modalidades" u "operadores modales". Gracias a su simplicidad sintáctica, estos lenguajes son buenas herramientas para describir y razonar acerca de estructuras relacionales. Una estructura relacional es un conjunto no vacío sobre el cual se han definido un cierto número de relaciones; tienen un amplio uso en matemáticas, ciencias de la computación, inteligencia artificial (IA) y lingüística, y también son utilizadas para interpretar lenguajes de primer orden [Blackburn].

La lógica modal es bien aceptada en filosofía y hoy en día en la comunidad de IA como herramienta para el análisis de conceptos. Una modalidad es una palabra o frase que puede aplicarse a una proposición $\mathscr{A}$ para crear una nueva proposición que hace una afirmación acerca del modo de verdad de $\mathscr{A}$, acerca de las circunstancias bajo las cuales $\mathscr{A}$ es verdadera: cuándo, dónde o cómo $\mathscr{A}$ es verdadera. Ejemplos son: "en el futuro sucederá $\mathscr{A}$" ($F\mathscr{A}$), "está permitido $\mathscr{A}$"($P\mathscr{A}$), "el agente sabe $\mathscr{A}$" ($K\mathscr{A}$), "es necesario $\mathscr{A}$" ($\Box\mathscr{A}$), "alguna ejecución finita del programa π deja al programa en un estado con información $\mathscr{A}$" ($\langle\pi,\mathscr{A}\rangle$), "es demostrable $\mathscr{A}$", entre muchas otras. En la actualidad los simbolismos modales se usan en computación

para formalizar esquemas de razonamiento. Para una buena y actual lectura introductoria de conceptos técnicos esenciales de la lógica modal, sugerimos abordar [BdRV01, Che80, Blackburn].

En este trabajo comenzamos describiendo los conceptos de lógica modal usando un lenguaje lógico, que luego en el capítulo 4 servirán de fundamentos para realizar la implementación del chequeador de modelos en un lenguaje de programación. El lenguaje modal básico que adoptamos se funda sobre un conjunto numerable de proposiciones, denotadas como $P = p, q, r, ...$ Expresiones complejas se forman sintácticamente a partir de aquellas de la forma inductiva usual usando el operador $\bot$(la constante *false*), el operador binario $\vee$(disyunción), y el operador unario $\neg$ (negación). Como el comportamiento proposicional de esta lógica es clásico, asumimos que la constante *true* ($\top$), $\wedge$(conjunción), $\rightarrow$ (condicional) se definen del modo esperado a partir de los símbolos ya provistos. Además se agregan 2 símbolos ($\Box$ y $\Diamond$) que representan las modalidades que pueden preceder a cualquier fórmula. El lenguaje por lo tanto queda definido como:

$$\varphi :: p|\top| \bot |\neg\varphi|\varphi \wedge \psi|\varphi \vee \psi|\varphi \rightarrow \psi|\varphi \leftrightarrow \psi|\Box\varphi|\Diamond\varphi \text{ donde } \varphi \text{ es una fbf.}$$

Dado el lenguaje, pasemos a las estructuras sobre las que tradicionalmente trabaja una lógica modal.

Definición. Frame: un frame es una dupla $\mathfrak{F} = (W, R)$, tal que W es un conjunto no vacío llamado universo (o dominio) de $\mathfrak{F}$ y R es una relación binaria sobre W. Los elementos de W se llaman puntos, situaciones, mundos o estados.

Definición. Modelo: un modelo es un par $\mathfrak{M} = (\mathfrak{F}, V)$ donde $\mathfrak{F}$ es un frame y V es una función de valuación que asigna a cada proposición p del lenguaje un subconjunto $V(p)$ de W. Así, $V(p)$ es el conjunto de situaciones en el modelo donde p es verdadera.

Ejemplo: Podemos representar gráficamente un modelo como sigue:
$\mathfrak{F} = (\{w_1, w_2, w_3, w_4, w_5\}, R)$
$R(w_i, w_j)$ si y sólo si $j = i + 1$
Función de valuación:
$V(p) = \{w_1, w_2, w_3\}$

$V(q) = \{w_5\}$
$V(r) = \{\}$

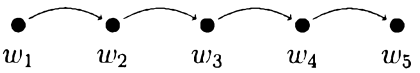


Figura 2.1: Ejemplo de modelo

2.2. Semántica relacional (o de Kripke)

La invención de la semántica relacional basada en grafos (por Jaakko Hintikka, Stig Kanger y Saul Kripke) por los años '50/'60, mostró que la lógica modal standard puede ser considerada como fragmentos de la lógica de predicados de primer o segundo orden. La idea subyacente es sencilla: suponiendo que leemos $\Box\varphi$ como "es necesario φ" y $\Diamond\varphi$ como "es posible φ", podemos ver a $\Box\varphi$ como la afirmación de que φ es verdadero en todos los *mundos posibles*, y a $\Diamond\varphi$ como la afirmación de que φ es verdadero en algunos *mundos posibles*.

Expresada matemáticamente, esta idea significa que la modalidad de necesidad $\Box$ se asocie con el cuantificador universal $\forall$ y que la posibilidad $\Diamond$ se asocie con el cuantificador existencial $\exists$. La diferencia con los cuantificadores tradicionales es que en los lenguajes modales, los cuantificadores están asociados a situaciones 'relevantes' o 'accesibles' a partir de la actual. En otras palabras, hay una estructura de situaciones o mundos accesibles donde el acceso a los mismos está mediado.

Sea R una relación binaria, R cumple la siguientes reglas para los operadores modales:

$w \vDash \Box\varphi \leftrightarrow \forall v$ tal que wRv, $v \vDash \varphi$

y

$w \vDash \Diamond\varphi \leftrightarrow \exists v$ tal que wRv, $v \vDash \varphi$

Entonces la semántica de cada operador normal está dada por la estructura de sus relaciones. Esta definición será importante más adelante cuando

definamos la implementación de nuestro chequeador.

Lógicas más "ricas" emergen al nivel de los *frames*, via el concepto de *validez de los frames* [Blackburn, p.33] (en el capítulo 5 analizamos como influye esto en nuestro chequeador de modelos). Las relaciones que se establecen para cada modalidad suelen cumplir ciertos axiomas, los cuales determinan la estructura del frame, la forma de sus relaciones. Por ejemplo el sistema **K**, tiene como axioma $\Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$, pero si estamos trabajando con la noción de tiempo, es natural asumir que las relaciones de accesibilidad son transitivas, y cualquier instancia de $\Box\psi \rightarrow \Box\Box\psi$ es válida en la clase de los frames transitivos. Sin embargo, ninguna instancia de este esquema es válido en **K**, entonces si queremos una lógica para trabajar con flujos temporales, debemos agregar todas las instancias como axiomas extras. Haciendo esto obtenemos una nueva lógica conocida como **K4**.

Existen otros tipos de semánticas con los que puede verse una lógica modal, por ejemplo la *semántica algebraica* [Blackburn, 1.7, p. 72] que permite describir álgebras booleanas con operadores o la semántica topológica que permite describir topologías, pero desde el punto de vista semántico, la semántica de Kripke (o relacional) es una de las más conocidas y más estudiadas y en particular representa bien la semántica de nuestros operadores modales normales.

Definición. Semántica de Kripke: Sea $\mathfrak{F} = (W, R)$ un frame y sea $w \in W$ un mundo en un modelo $\mathfrak{M} = (\mathfrak{F}, V)$. Sea $\mathscr{A}$ una fórmula cualquiera, tenemos entonces ($\mathfrak{M}, w \vDash \mathscr{A}$ se lee "la fórmula $\mathscr{A}$ es válida en el mundo w del modelo $\mathfrak{M}$"):

- $\mathfrak{M}, w \nvDash \bot$
- $\mathfrak{M}, w \vDash \mathscr{A}$ si y sólo si $w \in V(\mathscr{A})$
- $\mathfrak{M}, w \vDash \neg\mathscr{A}$ si y sólo si $\mathfrak{M}, w \nvDash \mathscr{A}$
- $\mathfrak{M}, w \vDash \mathscr{A} \vee \mathscr{B}$ si y sólo si $\mathfrak{M}, w \vDash \mathscr{A}$ o $\mathfrak{M}, w \vDash \mathscr{B}$
- $\mathfrak{M}, w \vDash \Box\mathscr{A}$ si y sólo si para todo v tal que wRv, $\mathfrak{M}, v \vDash \mathscr{A}$

2.2.1. Semántica de Scott-Montague

De acuerdo con Hansson y Gärdenfors [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 ordenado $\langle W, N\rangle$ 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.

Para la semántica del operador no normal *Does*, tomamos la definición usada en [GR04b]:

- $Does_x\mathscr{A}$ "*agent x brings it about* $\mathscr{A}$"
- La anterior es la modalidad E de Elgesem con semántica de Scott-Montague [Elg97]

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 consiste en chequear si, dada una fbf φ, y dado un modelo $\mathfrak{M}$, existe un $w \in W$ tal que $\mathfrak{M}, w \vDash \varphi$. Dicho problema también puede plantear localmente: dada una fbf φ, un modelo $\mathfrak{M}$ y un w, es cierto que φ es válida en w para el modelo $\mathfrak{M}$? o lo que es lo mismo: es válido $\mathfrak{M}, w \vDash \varphi$?

En general esto puede verse como un programa o función que recibe como entrada un modelo y una fórmula y responde verdadero o falso si dicha fórmula es satisfactible en dicho modelo. Gráficamente:

El chequeo de modelos es de gran importancia práctica. Una amplia gama

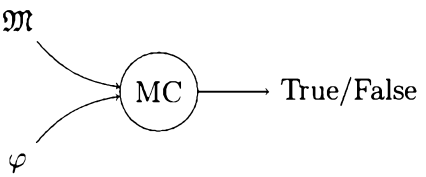


Figura 2.2: Chequeador de modelos

de tareas prácticas pueden ser modeladas de manera naturalmente computacional y resueltas de manera eficiente vía el chequeo de modelos. Un ejemplo clásico es la verificación de hardware. A pesar de que el chip es un objeto concreto, es posible modelar todos los estados en los que este puede estar y las posibles transiciones entre ellos. Para que funcione correctamente el chip debe poseer ciertas propiedades, por ejemplo no debe entrar en situaciones de *deadlock*. Si tenemos un lenguaje modal que permite expresar estas propiedades, entonces chequeando la fórmula en el modelo que representa al chip podemos determinar si el diseño es satisfactorio o no [Blackburn, 4.1].

El algoritmo tradicional para chequear modelos consiste en usar un algoritmo de marcado (*labeling algorithm*) desde abajo hacia arriba, o de lo más simple a lo más complejo, (*bottom-up*). Para chequear una fórmula φ, se marcan todos los puntos en el modelo con las subfórmulas de φ que son verdaderas en esos puntos. Se comienza con los símbolos de proposición: la valuación es la que determina si son verdaderos en un punto, en cuyo caso son marcados. Luego se marcan las fórmulas más complejas. Las booleanas son marcadas de la manera usual: por ejemplo, se marca w con $\psi \wedge \theta$ si w ya está marcado con ψ y θ. Para las modalidades, marcamos w con $\Diamond\varphi$ si **alguno** de los puntos relacionados está marcado con φ, y lo marcamos con $\Box\varphi$ si **todos** los puntos relacionados están marcados con φ [Blackburn, 4.1].

Este proceso es una tarea evidentemente computacional, pero es también una tarea de razonamiento. El modelo es esencialmente un almacén de información, consiste en un conjunto de entidades, junto con una especificación de qué propiedades son válidas en qué entidades, y cuales son las otras entidades con las que se relaciona. Una fórmula modal por el otro lado es un árbol construido recursivamente. El anidamiento de conectivos y modalidades permite que fórmulas relativamente cortas hagan afirmaciones interesantes que pueden ir más allá de un mero enunciado de verdades atómicas. En general,

dada una aplicación típica, las fórmulas son mucho más chicas que el modelo, por lo que chequear el modelo consiste en sincronizar dos tipos muy diferentes de información: nos determina si la información abstracta que conlleva la fórmula está implícitamente presente en el modelo y nos da los puntos en que esta información está [Blackburn, 4.1].

Como mencionamos antes, las lógicas más ricas emergen al nivel de los "frames", y como es indispensable contar con frames válidos, algunos chequeadores de modelos proveen un lenguaje de entrada, el cual permite describir los axiomas de una lógica dada (algunas implementaciones se describen en sección 6), las cuales pueden ser traducidas a un frame de manera que el resultado sea un frame válido para la lógica. Esto supone tener un lenguaje por cada lógica (o conjunto de lógicas), pero decidimos separar esta responsabilidad de nuestro chequeador de modelos, de manera que pueda servir como un módulo para chequear distintas lógicas, cada una con un lenguaje particular.

El chequeador podría además incluir un validador que chequee que el frame de entrada satisfaga todos los axiomas de la lógica que estamos modelando. Esto podría ser útil en los casos en que un mismo frame sea usado para chequear varias fórmulas, por lo que se podría validar una única vez el frame y luego usarlo para chequear las fórmulas.

Capítulo 3

Combinación de lógicas normales y no normales

3.1. Una lógica multi-modal para MAS

Las lógicas multi-modales son por lo general una combinación de lógicas específicas (de propósitos especiales) que dan lugar a teorías complejas. En este trabajo usamos la teoría expuesta en [SR2010], que define el concepto de confianza colectiva en un sistema multi-agente, mediante la combinación de operadores modales normales y no normales. Los operadores usados son:

- Operadores individuales
  - $BEL_i(\varphi)$: El agente i cree φ
  - $INT_i(\varphi)$: El agente i tiene la intención φ
  - $GOAL_i(\varphi)$: El agente i tiene el objetivo φ
  - $DOES_i(\varphi)$: El agente i lleva a cabo φ
- Operadores grupales
  - $E-BEL_G(\varphi)$: Todo agente en el grupo G cree φ
  - $E-INT_G(\varphi)$: Todo agente en el grupo G tiene la intención individual de hacer φ verdadera

  - $M - INT_G(\varphi)$: El grupo G tiene la intención mutua φ
  - $C - INT_G(\varphi)$: El grupo G tiene la intención colectiva de hacer φ verdadera

- Operador genérico
  - $OBL(\varphi)$: Es obligatorio φ

Las modalidades BEL, $GOAL$, INT, OBL (normales), $DOES$ (no normal) y las modalidades colectivas $C-BEL$ y $C-INT$, $E-BEL$, $E-INT$ y $M-INT$ están definidas y estudiadas en [DKV02]. Por simplicidad, en lo que sigue trabajaremos sólo con las modalidades BEL, $GOAL$, INT, OBL y $DOES$

3.2. Modelo multi-relacional

La estructura del MAS definido en [SR2010, Def 1] es un frame multi-relacional de la forma[1]

$$\mathfrak{F} = \langle\ A,\ W,\ \{B_i\}_{i\in A},\ \{G_i\}_{i\in A},\ \{I_i\}_{i\in A},\ \{D_i\}_{i\in A}\ \rangle$$

donde:

- A es un conjunto finito de agentes
- W es un conjunto de puntos, mundos o situaciones posibles
- $\{B_i\}_{i\in A}$ es un conjunto de relaciones respecto de BEL, las cuales son transitivas[2], euclidianas[3] y seriales[4],
- $\{G_i\}_{i\in A}$ es un conjunto de relaciones respecto de GOAL, con semántica K_n estándar,

[1] No se incluyen las modalidades de grupo ni la modalidad O para mantener manejable el conjunto de modalidades

[2] Una relación R es transitiva si $\forall x\forall y\forall z(xRy \wedge yRz \rightarrow xRz)$ [Blackburn]

[3] Una relación R es euclideana si $\forall x\forall y\forall z(xRz \wedge xRy \rightarrow yRz)$ [Blackburn]

[4] Una relación R es serial si $\forall x\exists y(xRy)$ [Blackburn]

- $\{I_i\}_{i \in A}$ es un conjunto de relaciones respecto de INT, las cuales son seriales,
- $\{D_i\}_{i \in A}$ es una familia de conjuntos de relaciones de accesibilidad D_i con respecto a DOES, que son cerradas punto a punto con respecto a la intersección, reflexivas[5] y seriales [GR04b]

Para todo agente $i \in \mathcal{A}$, 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.

Definición: Un modelo basado en el frame $\mathfrak{F}$ tiene la forma $\langle \mathfrak{F}, V \rangle$ donde V es la función de valuación correspondiente [SR2010, Def 2]. La modalidad $DOES$ está restringida a constantes proposicionales que representan acciones o comportamientos, por ejemplo $DOES_x Pagar$, significa "el agente x paga". Esta restricción limita las fórmulas que pueden escribirse. Por ejemplo la fbf $DOES_i(DOES_j Pagar)$ no puede escribirse en este MAS.

3.3. Fibrado de lógicas

Si bien existen distintas técnicas para combinar lógicas modales, de las usadas en [AM2011], nos limitaremos a implementar un chequeador para un fibrado de lógicas modales.

Definición. Fibrado: Intuitivamente fibrar dos lógicas L y M (anotamos $L(M)$) consiste en organizarlas en dos niveles, el nivel superior para la lógica L y en el segundo nivel la lógica M.

Según Finger y Gabbay [FG94, FMDR04], la 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. Es posible identificar dos redes de relaciones sobre el mismo conjunto W. La primera "red" corresponde a las relaciones de accesibilidad de los operadores normales mientras que

[5]Una relación R es reflexiva si $\forall x(xRx)$ [Blackburn]

la segunda corresponde a las relaciones de acesibilidad de las modalidades $DOES_i$.

Con respecto a la noción de fibrado usada, el método 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.

Tenemos entonces que una lógica modela el comportamiento interno de los agentes (sus creencias, objetivos e intenciones) y la otra modela la interacción que los agentes tienen con su entorno (la habilidad de realizar acciones). $\mathfrak{F}$ puede verse dividido en un frame multi-modal exterior, y frames de Scott-Montague interiores. Gráficamente:

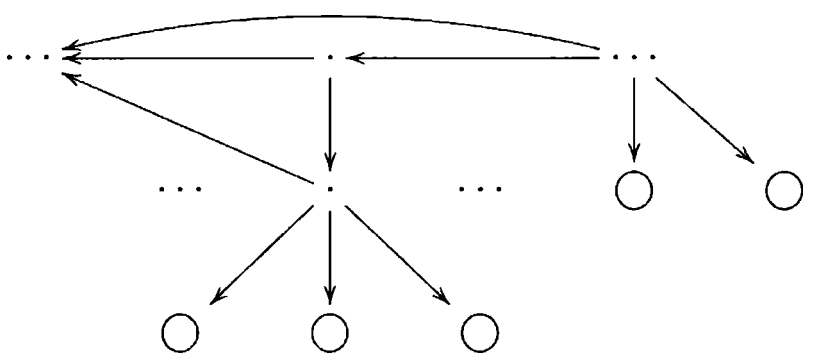


Figura 3.1: Organización del fibrado

Donde los puntos pequeños son elementos del modelo de Kripke y los círculos grandes son modelos de Scott-Montague.

A continuación organizaremos a $\mathfrak{F}$ como un fibrado tratando los aspectos técnicos del mismo. Por cuestiones de practicidad dejaremos de lado los operadores modales grupales.

3.4. Fibrado: Sintaxis y semántica:

Llamamos **N** a la restricción de $\mathfrak{F}$ a su parte normal. Llamamos **Does** a la restricción de $\mathfrak{F}$ a su parte no normal. Asumimos que **Does** es una lógica proposicional ya que sólo se aplica la modalidad DOES a fórmulas atómicas que pueden ser reemplazadas por constantes proposicionales sin perder expresividad [Elg97, GR04b]. Llamaremos **N(Does)** al fibrado de **Does** mediante **N**.

N(Does): Sintaxis. Sea $\mathscr{L}_{Does}$ el lenguaje lógico asociado a los agentes (sin modalidades normales y sin las correspondientes reglas sintácticas de formación), y $\mathscr{L}_N$ el lenguaje asociado a **N** (sin la modalidad DOES y sin su regla de formación sintáctica). El lenguaje $\mathscr{L}_{N(Does)}$ de **N(Does)** (sobre el conjunto de letras proposicionales P), es obtenido reemplazando la regla de formación de las reglas en $\mathscr{L}_N$ que dice "cada letra proposicional en P es una fórmula" por la regla:

cada fórmula monolítica en $\mathscr{L}_{Does}$ *es una fórmula*

Como expone [FG94], este reemplazo puede ser visto como un proceso llamado "fuzzling" o *layering*: fórmulas en el sistema base (en este caso las de **Does**) pueden ser sustituídas por átomos en el sistema superior (en este caso las de **N**).

Para describir formalmente la semántica del fibrado, necesitamos reformular los modelos basados en $\mathfrak{F}$ en términos de modelos restringidos.

Definición. Modelo fibrado: Un modelo para **N(Does)** tiene la estructura:

$$\langle A, W, B_{i i\in A}, G_{i i\in A}, I_{i i\in A}, d_i, V' \rangle$$

donde:

- $A, W, \{B_i\}_{i\in A}, \{G_i\}_{i\in A}, \{I_i\}_{i\in A}$ están definidos como en la definición previa del frame
- V' es la función de valuación V restringida a los operadores normales, definida como sigue:

  - condiciones booleanas estándar
  - $V'(w, BEL_i\mathscr{A}) = 1$ sii $\forall v \in W$ (si wB_iv entonces $V'(v, \mathscr{A}) = 1$)
  - $V'(w, GOAL_i\mathscr{A}) = 1$ sii $\forall v \in W$ (si wG_iv entonces $V'(v, \mathscr{A}) = 1$)
  - $V'(w, INT_i\mathscr{A}) = 1$ sii $\forall v \in W$ (si wI_iv entonces $V'(v, \mathscr{A}) = 1$)
- cada d_i es una función total que mapea, para cada mundo w en W, para cada agente i, un modelo con estructura:

$$\langle W, D_i, v \rangle$$

donde:

- W es el mismo conjunto de mundos
- D_i es una familia de relaciones de accesibilidad $\mathbf{D}_i$ con respecto al accionar del agente i
- v es V restringida a los operadores no normales (los $DOES_i$). Esto es, la función de valuación que dice cuando $DOES_I\mathscr{A}$ es verdadera en alguno de los vecindarios de w. Formalmente v está definida como:
  - condiciones booleanas estándar
  - $v(DOES_i\mathscr{A}) = 1$ sii $\exists \mathbf{D}_i \in D_i$ tal que $\forall u(w\mathbf{D}_iu$ sii $v(u, \mathscr{A}) = 1$

Gráficamente el fibrado tiene la siguiente forma:

Donde los puntos pequeños son elementos del modelo de Kripke y los círculos grandes son modelos de Scott-Montague.

La lógica basada en $\mathfrak{F}$ puede ser vista como una combinación donde la lógica normal es puesta sobre una lógica no normal.

Es posible identificar dos "redes" (o multi-grafo) correspondientes al "cableado" de los operadores normales y otra a las relaciones de accesibilidad para las modalidades $DOES$. La primera de las redes puede verse como un frame multi-modal con frames multi-modales des Scott-Montague interiores.

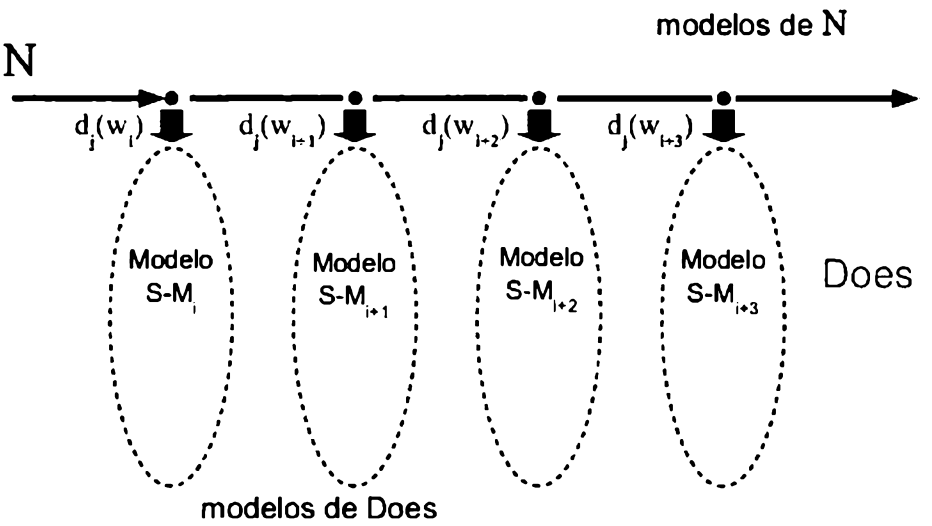


Figura 3.2: Estructura del fibrado

Bajo esta organización, la intuición detrás de la valuación de las fórmulas en el sistema es la siguiente: cuando evaluamos operadores normales, navegamos por el modelo de Kripke exterior. Cuando un $DOES$ aparece en algún punto w para ser evaluado, navegamos por un modelo de Scott-Montague. Luego de evaluar la fbf en el modelo de Scott-Montague, substituiremos el resultado con algún objeto (fbf, variable proposicional, etc) que pertenezca al dominio de los modelos de Kripke (nivel superior), para continuar con la evaluación.

3.5. Correctitud y completitud de la lógica base

Dos propiedades importantes de un procedimiento de deducción son su corrección y su completitud. Un procedimiento de deducción es correcto si cualquier consecuencia lógica que haya sido probada con él es válida. Un pro-

cedimiento de deducción que permita probar cualquier consecuencia válida es completo. Evidentemente, la corrección es una propiedad deseable en todo procedimiento deductivo, ya que en caso contrario el uso del procedimiento podría conducir a conclusiones inválidas. La completitud es también una propiedad deseable, aunque no es posible garantizarla para todas las lógicas. [AM2011]

La prueba de completitud para la lógica base consiste en probar la completitud de cada uno de los operadores modales y luego utilizar los resultados obtenidos para lograr una prueba de completitud de la lógica resultante de la combinación de dichos operadores. Las pruebas de completitud para todos los operadores normales puede verse en [HM92], excepto para el operador $C - BEL$ que es probado en [AM2011, Cap 3]. Luego, también en [AM2011, Cap 3] se prueba la completitud de la restricción de $N(Does)$ a su parte normal.

Basándonos en la definición [BdRV01, 4.24] (estabablece como construir un modelo canónico para una lógica modal normal) y el teorema [BdRV01, 4.22] (establece que una lógica modal normal es fuertemente completa con respecto a su modelo canónico), podemos establecer que la signatura **N** construida a partir de las modalidades normales tiene modelo canónico y que respecto al mismo, esta lógica es completa.

La lógica del *Does* tiene la propiedad de *modelo finito (fmp)* [GR04b, SP08]. Esta propiedad es muy útil pues nos permite desentendernos de modelos arbitrarios infinitos, ya que siempre podemos encontrar uno finito equivalente [GR04b]. Además esta lógica posee la propiedad de *modelo de tamaño múltiple de un paso* (OSPMP es una propiedad de modelo pequeño para lógicas sin transiciones semánticas) [SP08]. Gracias a esta propiedad podemos asegurar que la lógica también es decidible (y está acotada a P-SPACE).

Capítulo 4

Implementación de un chequeador de modelos para $N(Does)$

4.1. Definiciones previas

Este es el pseudo-código del chequeador de modelos provisto en [AM2011b] para la lógica $N(Does)$ que nos sirvie de punto de partida:

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

Function $\mathrm{MC}_{N(Does)}$ $((A,\ W,\ B_i,\ G_i,\ I_i,\ V',\ \{\mathrm{d}_i\}),\ \varphi)$
  input: un modelo fibrado $\mathfrak{M}$ y una fórmula $\varphi \in \mathscr{L}_{N(Does)}$
  computar $\mathrm{MM}\mathscr{L}_{Does}(\varphi)$
  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($(A,\ w,\ B_i,\ G_i,\ I_i,\ V'),\ \alpha_1$) and
and not check($(A,\ w,\ B_i,\ G_i,\ I_i,\ V'),\ \alpha_2$)
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 chequeador $\mathrm{MC}_{N(Does)}$ primero computa el conjunto $\mathrm{MM}\mathscr{L}_{Does}(\varphi)$ de sub-fórmulas monolíticas maximales de φ. Para cada una de éstas, 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 $\eta = \langle A, W_D, \{D_i\}, v\rangle$ y 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 vecindario (*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 (las fórmulas del sistema base son sustituidas 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*.

4.2. Programación orientada a objetos

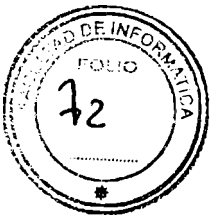
La programación orientada a objetos (en adelante OOP) es un tipo de paradigma de programación dentro del paradigma de programación imperativa, [Rum91]. La principal ventaja frente a otros paradigmas está dada en la forma de diseño, desarrollo y mantenimiento del software ofreciendo una solución a largo plazo a los problemas y preocupaciones que han existido desde el comienzo en el desarrollo de software: la falta de portabilidad del código y reusabilidad, código que es dificil de modificar, ciclos de desarrollo largos y tecnicas de codificacion no intuituvas.

OOP se basa en la idea natural de la existencia de un mundo lleno de objetos. La resolución de los problemas se realiza en términos de objetos y sus interacciones. Para lograr esta representación del mundo se utiliza la definición de clases, las cuales definen el molde para crear objetos. Los objetos se instancian a partir de una clase y mantienen un estado interno y un comportamiento asociado. Los objetos exponenen su comportamiento por medio de métodos que pueden ser invocados por otros objetos para comunicarse.

Para resolver un problema, hay que identificar los objetos que colaborarán en la solución del mismo y definir su comportamiento. Uno de los objetivos para un buen diseño es crear objetos con funciones bien definidas y precisas dentro del sistema, esto se denomina cohesión, manteniendo un bajo acoplamiento entre las clases.

Uno de los lenguajes más representativos de la programación orientada a objetos es JAVA. Una de las grandes ventajas de este lenguaje es su potencia y facilidad para describir soluciones de problemas respetando la metodoligía OO. Sus caracteristicas de concurrencia y multiplataforma permiten implementar sistemas de alto poder de computo en diversas plataformas.

En este capítulo utilizamos las herramientas del lenguaje de programación JAVA para implementar las nuestro chequeador de modelos.



4.3. Definiciones básicas sobre Java

[JAVA] es un lenguaje orientado a objetos ampliamente usado en la industria del software, [TIOBE]. El código Java se compila a un código de más bajo nivel llamado *bytecode*, el cual se ejecuta en la Java Virtual Machine (JVM), la cual puede ser ejecutada en cualquier plataforma independientemente de la arquitectura subyacente. De esta manera el mismo código compilado a bytecode, puede ejecutarse en las distintas plataformas sin ser recompilado ya que sólo depende de la JVM que es la encargada de proveer la transparencia de la plataforma.

Hay muchas herramientas desarrolladas en Java que facilitan la distribución de tareas, lo que permite distribuir el procesamiento del chequeador de modelos en varios nodos que contribuyen a resolver de manera más rápida el problema:

- [Hazelcast]: implementa el framework de colecciones de Java pero de manera distribuida, proveyendo una memoria compartida entre los distintos nodos de un cluster.
- [Akka]: Es un framework que puede ser usado en Java implementa el patrón de actores, en el cual los procesos son denominados actores y se comunican enviándose mensajes. El concepto es similar a la programación orientada a objetos sólo que los actores pueden residir en otras ubicaciones o nodos, pero su ubicación es transparente para el programador.
- [Scala]: Es un lenguaje de programación multi-paradigma diseñado para expresar patrones comunes de programación en forma concisa, elegante y con tipos seguros. Integra sutilmente características de lenguajes funcionales y orientados a objetos. La implementación actual corre en la máquina virtual de Java y es compatible con las aplicaciones Java existentes.

Con el auge de la computación en la nube, se ven más a menudo clusters de computadoras (de menor poder de procesamiento que un supercomputador), pero que combinadas proveen un gran poder de cómputo. Estas y

otras herramientas proveen facilidades que no encontramos en otros lenguajes para el desarrollo de el chequeador y especialmente para poder hacer uso de la distribución en el procesamiento del chequeador, algo que debería tener un impacto considerable en los tiempos de respuesta del mismo.

4.4. Principales dificultades

Uno de nuestros objetivos en este trabajo es el de acercar conceptos y herramientas de la lógica modal, usada como lenguaje de modelización de sistemas, a un lenguaje de programación actual y de amplia difusión.

Materializamos nuestras expectativas de trabajar en un paradigma orientado a objetos algunos conceptos que son fácilmente modelados con lenguajes declarativos. La flexibilidad y las capacidades de abstracción de los lenguajes lógicos, sumados a la posibilidad de definir fácilmente nuevos conceptos nos puso ante el desafío de utilizar un lenguaje orientado a objetos para implementar dichas abstracciones y definiciones formales.

La principal dificultad es proveer y emular esta flexibilidad de un lenguaje lógico usado como lenguaje de representación de conocimiento (o KRL) en Java. En un lenguaje declarativo como Prolog, una sentencia dice mucho. Por ejemplo:

$$hombre(h).$$

Afirma que h es hombre. En un lenguaje orientado a objetos, h es entendido como una instancia de hombre, ya que como h existen otros hombres. Lo que corresponde hacer es crear una instancia de la clase Hombre y referenciarla (o nombrarla) h.

La herencia de la OOP puede verse desde un punto de vista de la programación declarativa como una regla de inferencia. Por ejemplo:

$$mamifero(x) :- hombre(x).$$

Representando que si x es un hombre, entonces x es mamífero. O lo que es lo mismo $\forall x : hombre(x) \rightarrow mamifero(x)$. Esta implicación es representada

por una relación de herencia donde Hombre es subclase de Mamifero. Pero a su vez, todo hombre es vertebrado, pero no todo vertebrado es mamífero. Por lo tanto usar la herencia para representar esta fórmula implica la necesidad de hacer uso de *herencia múltiple*, algo que Java no soporta. Por otro lado, por ser Java un lenguaje pre-compilado, representar de esta manera las fórmulas, no permite que la misma implementación se adapte fácilmente a otros dominios diferentes al usado al crear el chequeador.

Sumado al problema de la representación de conceptos simples está también el problema de como razonar sobre estos conceptos. Sabemos que los seres vivos respiran y que los hombres son seres vivos, nuestra modelización tiene que permitirnos inferir que *"si h es hombre, entonces h respira"*. La solución en POO es enviarle un mensaje al objeto h, consultando si h respira o no. Si hombre extiende de una clase SerVivo, entonces también hereda el "comportamiento" de respirar. Nuevamente, esta forma de modelar implica tener que diseñar objetos específicos por cada dominio de aplicación, respondiendo a distintos mensajes de acuerdo a su dominio. Esta forma de modelar tampoco nos permite responder si el enunciado $hombre(x) \rightarrow (vivo(x) \rightarrow hombre(x))$ es cierto, y sabemos que lo es por ser una tautología, instancia del axioma L1.

Evidentemente la tarea de diseñar el chequeador de modelos que nos ocupa, usando objetos e intentando usar las mismas definiciones que usaríamos con un lenguaje declarativo no es tarea sencilla. Sin embargo encontramos que la representación semántica de los mundos posibles encaja adecuadamente en el modelo de objetos, ya que los mundos son objetos abstractos, con fórmulas o enunciados que son ciertos en ellos y están relacionados entre sí mediante una relación dirigida de asociación. Estos conceptos de objeto con un estado interno y con relaciones hacia otros objetos son naturales para el paradigma: cada mundo se representa como un objeto, con propiedades específicas y própias de cada *"instancia"*, y con *"relaciones de conocimiento"* hacia otros objetos de su misma clase. Luego, con esta representación, recorrer el grafo de relaciones para chequear la validez de una fórmula, es una tarea simple aunque computacionalmente compleja.

Dado que la semántica relacional de mundos posibles, tanto para Kripke como para Scott-Montague, representa correctamente a la lógica modal, y sabiendo que es posible obtener un frame que satisface los axiomas de una lógica particular, decidimos que nuestro chequeador implemente dicha

semántica de manera general, en vez de resolver de qué manera implementar o representar los axiomas própios de la lógica subyacente. Este cambio de enfoque nos da la posibilidad de comparar los resultados obtenidos por nuestra implementación, con otros chequeadores que siguen el enfoque tradicional.

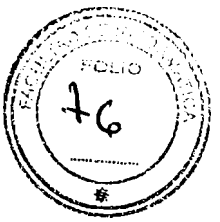
4.5. Representación del fibrado

4.5.1. 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. Un modelo se define como $\mathfrak{M} = (\mathfrak{F}, V)$ donde $\mathfrak{F} = (W, R)$ es un frame, W es el conjunto de mundos, R es un conjunto de relaciones de accesibilidad y V es una función de valuación que asigna a cada proposición p del lenguaje un subconjunto $V(p)$ de W.

- 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 finito $\mathcal{A}$.
- 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, representando la función de valuación del modelo $\mathfrak{M}$.
- 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.

Dado que en la POO los objetos están compuestos de estado más comportamiento decidimos que la función de valuación V quedara representada



dentro de cada mundo w como el estado interno del mundo. Es decir, el mundo w conoce las proposiciones que son verdaderas en él.

Definimos la clase *Frame* que conoce todos los mundos y agentes del frame. Para las relaciones de accesibilidad decidimos, de la misma manera que con lo mundos, que cada modalidad conozca sus relaciones de accesibilidad. De esta manera la clase *Frame* no quede acoplada a una implementación concreta (con un conjunto de modalidades específicas) y 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 y conocen exáctamente a las mismas relaciones de accesibilidad.

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 cómo 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.

Definimos las relaciones de accesibilidad como sigue:

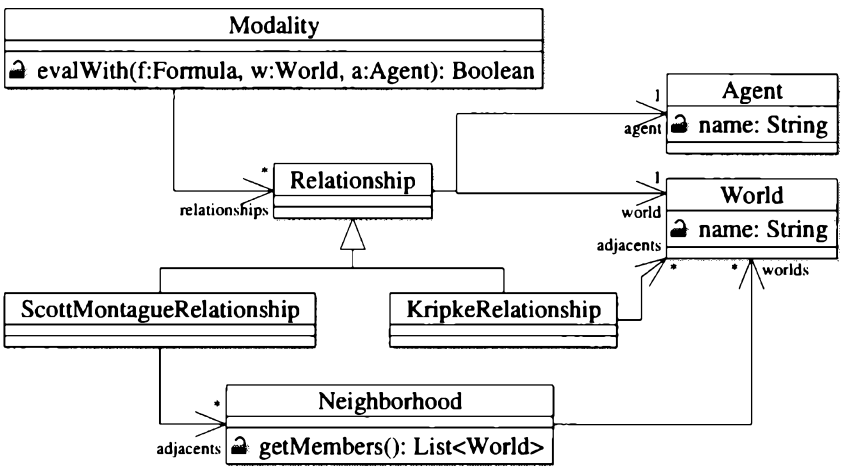


Figura 4.1: Representación de las relaciones de accesibilidad

Cada instancia de *Modality* (recordar que debe haber una única instancia por frame), 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*).

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.

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*).

4.5.2. 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: $DOES(i, F)$

Antes de implementar las fórmulas definidas anteriormente, utilizaremos el Lenguaje Unificado de Modelado ([UML]) para describir nuestra implementación del chequeador de modelos. UML es un lenguaje gráfico que ayuda a especificar, visualizar y documentar un sistema, incluyendo su estructura y diseño. Está construido usando los conceptos fundamentales de la POO, por lo que se adecúa al lenguaje que elegimos.

Definimos la clase abstracta *Formula* que representa la noción de verdad local [BdRV01, Definición 1.20], es decir, cuando dado un modelo $\mathfrak{M}$, una fórmula ϕ es verdadera en un mundo o estado w. El mensaje aceptado por la clase *Formula*: *eval(w: World): Boolean* representa esta definición, devolviendo verdadero cuando la instancia de la fórmula (ϕ) es verdadera en el mundo o estado (w) pasado como parámetro al mensaje. Cada subclase de *Formula* implementa un caso particular de la definición. En esta representación no se hace referencia al modelo, la razón quedará más clara en 4.5.1 donde veremos la relación existente entre el mundo w y el modelo $\mathfrak{M}$.

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.

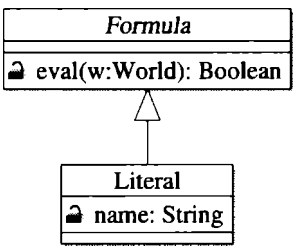


Figura 4.2: Representación de letras de proposición o literales

donde una instancia de literal podría ser por ejemplo:

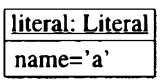


Figura 4.3: Representación de la proposición a

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.

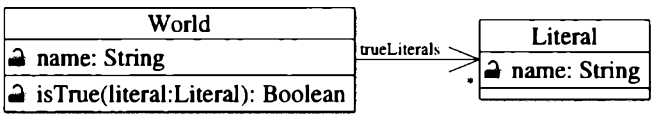


Figura 4.4: Representación de *World*

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

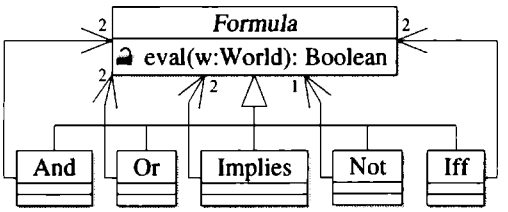


Figura 4.5: Representación de fórmulas booleanas

Para las fórmulas modales definimos una *ModalFormula* que tiene asociado un agente (*Agent*) y una fórmula sobre la cual se aplica la modalidad (4.6). Como cada modalidad puede tener una semántica diferente, hicimos uso del patrón de diseño *Strategy* [GHJV94] y separamos dicha implementación en una clase modalidad (*Modality*) asociada a la *ModalFormula*. Para el diseño de la composición entre fórmulas modales, que aplican sobre otras fórmulas, usamos el patrón *Composite* [GHJV94].Las instancias de estas clases representan fórmulas modales concretas. Por cada operador modal definimos una subclase de *Modality*

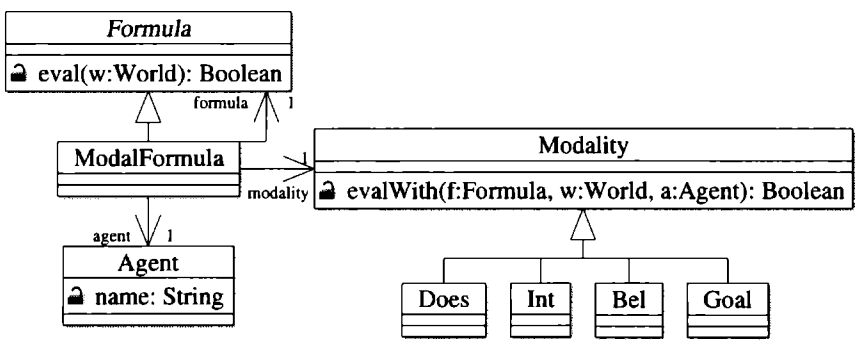


Figura 4.6: Representación de fórmulas modales

Las fórmulas modales colectivas las definimos como una extensión de *ModalFormula* redefiniendo la estrategia, la cual extiende de *Modality*, pero es una estrategia distinta que definimos como *GroupModality*, que permite asociar una fórmula modal a un grupo de agentes (4.7). Como en el caso anterior, cada implementación de *GroupModality* puede tener una estrategia diferente de evaluación.

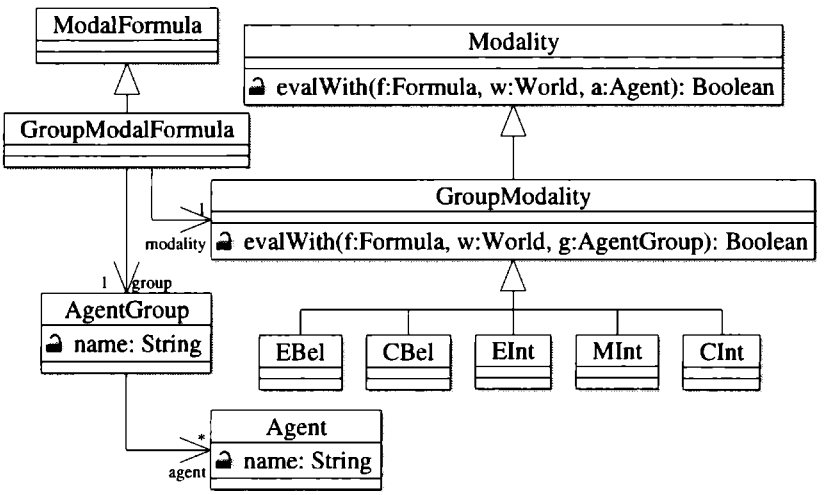


Figura 4.7: Representación de fórmulas modales colectivas

4.6. Evaluación de fórmulas

Como mencionamos anteriormente en la POO todos los objetos están compuestos de un estado interno y de un comportamiento asociado. En el caso de las fórmulas, cada objeto o instancia que hereda de *Forumula* define su semántica implementando la función $eval(w : World) : Boolean$. Para saber si una fórmula es *localmente verdadera* en un mundo w, hay que enviarle un mensaje al objeto fórmula, pasándo como parámetro el objeto w que representa al mundo. La fórmula responderá verdadero sólo si la fórmula es satisfactible en w. Esta forma de definir la semántica de cada fórmula sigue la definición de verdad local [BdRV01, def 1.20].

Dado que el frame conoce a todos los $w \in W$, implementar la definición de verdad global o universal en un modelo $\mathfrak{M}$ ([BdRV01, def 1.21]) implica repetir la evaluación anterior en todos los $w \in W$.

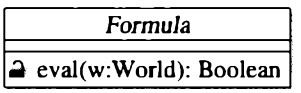


Figura 4.8: Clase para representar una fórmula

La implementación de nuestro chequeador da la posibilidad de realizar ambos chequeos sobre una fórmula: en un mundo particular (*verdad local*) o en todos los mundos pertenecientes al modelo (*verdad universal*)

4.7. Implementación Java

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

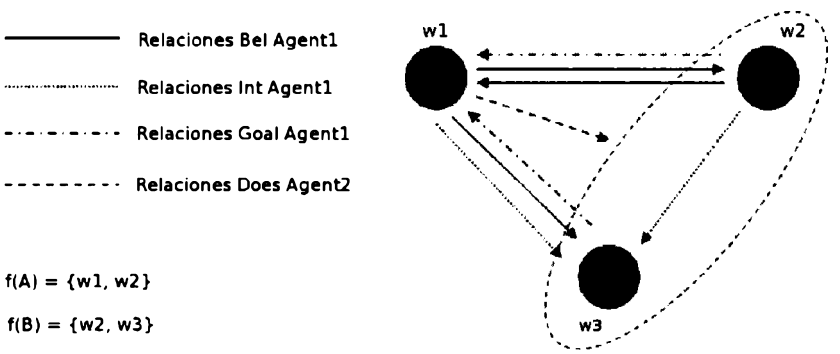
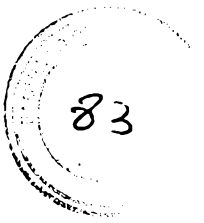


Figura 4.9: Modelo de ejemplo

Usando como ejemplo el modelo 4.9, 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 válidos 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 sus 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)));
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, 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);
}
```

ModelChecker acepta dos mensajes. El primero recibe sólo una fórmula a chequear y devuelve un listado de los mundos donde la fórmula es verdadera. Si todos los $w \in W$ del frame están dentro del conjunto resultante, la fórmula es una verdad global en el modelo [BdRV01, 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 [BdRV01, 1.20].

Puede verse en el código anterior que la responsabilidad de evaluar la fórmula en un mundo recae en la fórmula própiamente, ya que como mencionamos anteriormente decidimos que cada fórmula define su semántica.

Ya mencionamos que 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 todas respetan la semántica de necesidad. Las únicas diferencias

entre las mismas están dadas por los axiomas, es decir a nivel de los frames que se forman y que definen las estructuras de grafos válidas para cada modalidad [Blackburn, Cap 5].

Cada modalidad en nuestra lógica de referencia, tiene axiomas característicos como mencionamos en 3.2. Estas características definen la estructura del frame, lo que es útil para validar un frame, pero no varía la forma en que se evalúa. Esto es porque hay una corelación 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. Por ejemplo, sea W un conjunto de mundos, con $w, v, u \subset W$ y R una relación de accesibilidad entre ellos, estas son algunas de las propiedades que definen los axiomas de la lógica $N(Does)$:

Nombre	Axioma	Propiedad del frame
K	$\Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$	No aplica
D	$\Box A \rightarrow \Diamond A$	serial: $\forall w \exists v(wRv)$
4	$\Box A \rightarrow \Box\Box A$	transitivo: $wRv \wedge vRu \rightarrow wRu$
5	$\Diamond A \rightarrow \Box\Diamond A$	euclideano: $wRu \wedge wRv \rightarrow uRv$
T	$\Box A \rightarrow A$	reflexivo: $\forall w_i \in W w_i R w_i$

No fue necesario especializar el comportamiento de cada modalidad normal y por lo tanto su evaluación 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 true;
}
```

Para fórmulas no normales, en nuestro caso *Does*, evaluamos la fórmula 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 = 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 true;
}
```

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

- Definiciones semánticas de *E-BEL* y *C-BEL*:

  ([DKV02, C1]) $\mathfrak{M}, s \vDash E\text{-}BEL_G(\varphi)$ sii $\forall i \in G$, $\mathfrak{M}, s \vDash BEL(i, \varphi)$

  ([DKV02, C2]) $\mathfrak{M}, s \vDash C\text{-}BEL_G(\varphi)$ sii $\forall t$ que sea BEL-alcanzable desde s: $\mathfrak{M}, t \vDash \varphi$

- Definiciones semánticas de *E-INT* y *M-INT*:

  ([DKV02, M1]) $\mathfrak{M}, s \vDash E\text{-}INT_G(\varphi)$ sii $\forall i \in G$, $\mathfrak{M}, s \vDash INT(i, \varphi)$

  ([DKV02, M2]) $\mathfrak{M}, s \vDash M\text{-}INT_G(\varphi)$ 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*:

  ([DKV02, M3]) $\mathfrak{M}, s \vDash C\text{-}INT_G(\varphi)$ 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 (por ejemplo *E-BEL* conoce a *BEL*). Esto es requerido para todas las modalidades colectivas ya que están definidas en términos de las modalidades individuales (a diferencia de *C-INT* que está definida en términos de las modalidades colectivas *M-INT* y *C-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;
}
```

C-BEL.eval

```
public Boolean evalWith(Formula formula, World world
    , AgentGroup group) {
  Boolean result = true;
  // for every agent
  for (Agent a : group.getAgents()) {
    // compute the path of accessibility relationships
    Collection<World> path = getRelPath(world, a,
       Lists.<World> newArrayList());

    // and for every world in that path
    for (World member : path) {
      // evaluate the formula in that world
      result = result && belModality.evalWith(
         formula, member, a);
    }
  }
  return result;
}
```

```
private Collection<World> getRelPath(World
   startWorld, Agent agent, Collection<World>
   processed) {
  Collection<World> result = Lists.newArrayList();
  // collect all the worlds bel-reachables from this world
     for this agent
  for (KripkeRelationship rel: belModality.
     getRelationships()) {
    if (rel.getAgent().equals(agent) && rel.getWorld
       ().equals(startWorld) && !processed.contains(
       startWorld)){
      result = rel.getAdyacents();
    }
  }

  processed.add(startWorld);
  for (World adjacent : result) {
    result.addAll(getRelPath(adjacent, agent,
       processed));
  }
  return result;
}
```

E-INT.eval

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

M-INT.eval

```
public Boolean evalWith(Formula formula, World world
   , AgentGroup group) {
```

```
  Boolean result = true;
  // for every agent
  for (Agent a : group.getAgents()) {
    // compute the path of accessibility relationships
    Collection<World> path = getRelPath(world, a,
       Lists.<World> newArrayList());

    // and for every world in that path
    for (World member : path) {
      // evaluate the formula in that world
      result = result && intModality.evalWith(
         formula, member, a);
    }
  }
  return result;
}

private Collection<World> getRelPath(World
   startWorld, Agent agent, Collection<World>
   processed) {
  Collection<World> result = Lists.newArrayList();
  // collect all the worlds bel-reachables from this world
     for this agent
  for (KripkeRelationship rel: intModality.
     getRelationships()) {
    if (rel.getAgent().equals(agent) && rel.getWorld
       ().equals(startWorld) && !processed.contains(
       startWorld)){
      result = rel.getAdyacents();
    }
  }

  processed.add(startWorld);
  for (World adjacent : result) {
    result.addAll(getRelPath(adjacent, agent,
       processed));
  }
  return result;
```


```
}
```

C-INT.eval

```
public Boolean evalWith(Formula formula, World world
    , AgentGroup group) {
  Collection<String> agentNames = Collections2.
     transform(group.getAgents(), new Function<Agent
     , String>(){
    public String apply(Agent agent) {
      return agent.getName();
    }});

  GroupModalFormula mutualIntention = new
     GroupModalFormula(mutualIntentionModality,
     agentNames, formula);
  And and = new And();
  and.setLeft(mutualIntention);
  and.setRight(new GroupModalFormula(
     commonBeliefModality, agentNames,
     mutualIntention));
  return and.eval(world);
}
```

4.8. Consideraciones sobre la implementación

Desde el punto de vista semántico cada modalidad se diferencia de las otras por las relaciones de accesibilidad que la definen (3.2), las cuales quedan definidas por sus axiomas o propiedades. Desde el punto de vista de la semántica de Kripke (2.2), la validez de una fórmula modal es la misma para todas las modalidades *normales*:

- $V'(w, BEL_i\mathscr{A}) = 1$ sii $\forall v \in W$ (si wB_iv entonces $V'(v, \mathscr{A}) = 1$)
- $V'(w, GOAL_i\mathscr{A}) = 1$ sii $\forall v \in W$ (si wG_iv entonces $V'(v, \mathscr{A}) = 1$)
- $V'(w, INT_i\mathscr{A}) = 1$ sii $\forall v \in W$ (si wI_iv entonces $V'(v, \mathscr{A}) = 1$)

Mientras que para la modalidad *no normal* la validez está dada por la semántica de Scott-Montague (2.2.1):

- $v(DOES_i\mathscr{A}) = 1$ sii $\exists \mathbf{D}_i \in D_i$ tal que $\forall u(w\mathbf{D}_i u$ sii $v(u, \mathscr{A}) = 1$

donde D_i es una familia de relaciones de accesibilidad $\mathbf{D}_i$ con respecto al accionar del agente i

En esta tesis nos enfocamos en el chequeo de las fórmulas modales, asumiendo que el frame de entrada al chequeador es válido. Si bien también es posible verificar que las relaciones de accesibilidad del modelo sean válidas para cada modalidad, asegurando un modelo de entrada válido para el chequeador, nos enfocamos únicamente en la parte del chequeo de la fórmula.

Esta separación nos permitió simplificar la representación y enfocarnos en el recorrido del frame, decisión que fundamos sobre los siguientes conceptos:

- **Generalización:** Al no tener embebida la semántica de las modalidades, el chequeador permite chequear fórmulas en modelos multimodales fibrados "arbitrarios"
- **Separación de intereses (*separation of concerns*):** la validación o armado de un modelo para una lógica específica puede externalizarse, pudiendo tener 2 módulos independientes que cumplen tareas complementarias.
- **Optimización y reutilización:** el chequeador hace menos procesamiento y además, un modelo ya validado puede reutilizarse para chequear diversas fórmulas sin necesidad de volver a validarlo.

Cada modalidad conoce sus relaciones de accesibilidad para cada mundo y al no tener información sobre sus propiedades o axiomas, no se requiere agregar ningún comportamiento específico asociado a las mismas (ver 4.5.2). El algoritmo no tiene información sobre las propiedades de las modalidades específicas, de manera que se pueden incorporar nuevas modalidades al sistema, siempre que las mismas puedan representarse con alguna de las semánticas definidas. En particular las modalidades normales, usan la semántica de Kripke y todas las modalidades normales comparten dicha implementación. La

modalidad *Does* que es *no normal*, extiende de una implementación del recorrido de las relaciones de accesibilidad para la semántica de Scott-Montague.

Nota: En caso de un frame de entrada inválido, el chequeador podría darnos una respuesta sin sentido. Para evitar esto sería necesario validar previamente el frame, pero esta tarea puede ser tan costosa como chequear la fórmula en el modelo, por lo que queda como un trabajo separado.

4.8.1. Validación de las relaciones de accesibilidad

Para validar un frame, hay que ejecutar un proceso validador que verifique las propiedades definidas para la modalidad en el frame. Dicho validador puede extenderse para completar automáticamente un frame agregando las propiedades faltantes y requeridas para que el frame sea válido. Esto último escapa al chequeo de modelos ya que el modelo en el que se chequea la fórmula es diferente al modelo de entrada.

Por ejemplo, dado el siguiente frame que no es euclideano (se ve que $w_1 B w_2 \wedge w_1 B w_2$, pero no se cumple $w_2 B w_3$ ni tampoco $w_3 B w_2$):

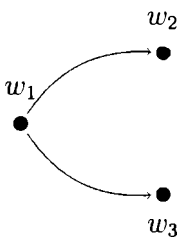


Figura 4.10: Frame no euclideano

Es posible reportar como inválido el frame o con el otro enfoque se pueden agregar las dos relaciones que se identifican como faltantes para cumplir con la propiedad de ser euclideano:

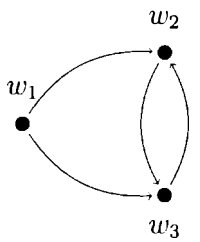


Figura 4.11: Frame euclideano

Se puede hacer lo mismo con las demás propiedades que tiene que cumplir el frame. Este acercamiento permite generar un modelo por comprensión, a partir de propiedades conocidas y no dar el detalle extensivo de lo que representa (Ej: $w_0 \vDash Bel(p)$ en vez de especificar para cada w_i BEL-adyacente a w_0: $w_i \vDash p$).

En [AM2011b], el trabajo de grado que sirve de base a nuestro trabajo, en la implementación Prolog del chequeador de modelos, se implementa la semántica de los operadores modales siguiendo la estructura del grafo de relaciones de cada uno. Esto puede verse en la sección del código *"EVALUACIÓN DE FÓRMULAS MODALES INDIVIDUALES NORMALES"*. Pero también se agregaron reglas generales de cada modalidad, por ejemplo en la página 89 se define el *"AXIOMA K PARA OPERADORES MODALES"*. Esto en principio es sólo un atajo sintáctico, dado que si se encuentra un esquema con esta forma, no es necesario hacer el recorrido semántico de las modalidades.

En el chequeador MCMAS ([LO2009] y [LO2014]), se proveen herramientas para que el usuario pueda detectar la correctitud del modelo. Esto se logra por medio de simulaciones interactivas, en la cual el usuario selecciona un estado inicial y sigue la evolución del modelo eligiendo las acciones de cada agente en cada estado, y también por medio de casos testigo y contraejemplos. MCMAS provee al usuario con ejecuciones de ejemplo justificando el valor de verdad de la fórmula, lo cual podría permitirle identificar casos en los que el modelo no satisface la especificación.

Conclusión: Desacoplar la tarea de obtener un frame válido del chequeo de una fórmula en dicho frame, nos permite tener un chequeador de modelos genérico, que no requiere ninguna regla especial para las modalidades. Este

fue nuestro objetivo principal por el cual quisimos implementar el chequeador en un lenguaje orientado a objetos y el motivo por el cual no usamos un lenguaje declarativo.

Dadas estas consideraciones, se desprende como trabajo, el analisis de la forma de obtener un frame válido para poder modelar situaciones de la realidad: cómo se definen las situaciones o mundos posibles y cómo se relacionan unos con otros de acuerdo a las distintas modalidades existentes. Definir un validador que sirva para validar el frame de entrada o usarlo para completar un frame, de manera que puedan darse las relaciones de accesibilidad por comprensión en vez de por extensión.

4.9. Ejemplo de aplicación

Si bien no es necesario hacer una prueba de funcionamiento del chequeador, dado que fue diseñado según las definiciones semánticas de las modalidades, decidimos hacer este ejemplo para que didácticamente se vea la forma de usar el chequeador.

Comenzamos preguntándonos si $\mathfrak{M}$, $w_1 \vDash Bel(a1, p)$ donde $\mathfrak{M} = (\ \{w_1, w_2\}, (w_1, w_2) \in B_i$, 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* (3.2). Si el frame cumple con las condiciones semánticas dadas por los axiomas de la modalidad, nuestro chequeador dará una respuesta que se corresponde con la semántica del operador modal, pero si el frame no es válido,

no podremos afirmar que nuestra respuesta sea correcta.

Capítulo 5

Análisis de tiempos de ejecución

5.1. Análisis de ejecución del chequeador

En esta sección mostramos que el tiempo de ejecución del algoritmo en el peor caso está acotado por $\mathcal{O}(N * n^2 * k) \in \mathcal{O}(2^n)$.

Para demostrarlo, repasemos primero la estrucutra del modelo multi relacional para $N(Does)$ presentado en [SR2010]:

$\langle \mathfrak{F}, V \rangle = \langle\ A,\ W,\ \{B_i\}_{i \in A},\ \{G_i\}_{i \in A},\ \{I_i\}_{i \in A},\ \{D_i\}_{i \in A},\ V'\ \rangle$

Dado un frame $\mathfrak{F}$ y una fórmula φ, utilizamos las siguientes variables para el cálculo del tiempo de ejecución del model checker:

- $n = |W|$
- $k = |\varphi|$
- $M = \sum_{R \in (B_i, G_i, I_i)} |R|$
- $N = max_{w \in W}\ |d_i(W)|$

El tiempo que tarda nuestro algoritmo en chequear φ en $\mathfrak{F}$ es el tiempo que tarda en ejecutar:

ModelChecker.java

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

El tiempo de *check* es $c1 + n * T_{eval}(k)$ donde $T_{eval}(k)$ es el órden de ejecución de *eval* para la fórmula φ (con $k = |\varphi|$) en un mundo. Por lo tanto, el orden de *check* $\in \mathcal{O}(n * T_{eval}(k))$

Para calcular el órden de ejecución de la fórmula, debemos analizar los distintos casos de *eval* que surgen de las variantes polimórficas de la fórmula:

5.1.1. Orden de ejecución de `eval` para fórmulas booleanas

El peor caso es $c2+2*T_{eval}(k-1)$ que surge de evaluar ambas subfórmulas (exepto el caso del *Not*, pero por simplicidad tomaremos el caso de una fórmula binaria). Mostramos por ejemplo el código de *And*:

And.java

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

Por lo tanto el orden de ejecución para fórmulas modales $\in \mathcal{O}(T_{eval}(k-1))$

5.1.2. Orden de ejecución de eval para fórmulas modales

El tiempo de ejecución depende de la modalidad. El tiempo de eval es constante ya que se delega su ejecución en la modalidad.

ModalFormula.java

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

Llamamos $T_m(k)$ al tiempo de ejecución de la modalidad. El orden de ejecución para fórmulas modales $\in \mathcal{O}(T_m(k))$

Para modalidades normales

NormalModality.java

```
public Boolean evalWith(Formula formula, World world
   , Agent agent) {
  for (KripkeRelationship rel : relationships) {
    if (rel.getAgent().equals(agent) && rel.getWorld
       ().equals(world)){
(1)   for (World adyacent : rel.getAdyacents()) {
        if (!formula.eval(adyacent)){
          if (formula.eval(world)){
            LOGGER.warn(this + ": Inconsistency
               between semantic and syntactic, we
               have a wrong Frame");
          }
          return false;
        }
      }
      return true;
    }
  }
  if (!formula.eval(world)){
```

```
    LOGGER.warn(this + ": Inconsistency between
       semantic and syntactic, we have a wrong Frame
       ");
  }
  return true;
}
```

El orden de ejecución es $\mathcal{O}(M * n * T_{eval}(k-1))$

Prueba: El orden de ejecución de (1) es $\mathcal{O}(n*(T_{eval}(k-1)+T_{eval}(k-1)))$ $\in \mathcal{O}(2n * T_{eval}(k-1)) \in \mathcal{O}(n * T_{eval}(k-1))$

(1) se ejecuta una vez por cada relación de Kripke definida y luego se evalúa la fórmula en el mundo actual para verificar si existe una inconsistencia en el modelo, por lo que el orden de ejecución de $evalWith$ es $\mathcal{O}(M * n * T_{eval}(k-1) + T_{eval}(k-1)) \in \mathcal{O}((M * n + 1) * T_{eval}(k-1))$

Para modalidades no normales

Recordar que la semántica de Scott-Montague (2.2.1) es una generalización de la semántica de Kripke donde en lugar de tener una única colección de mundos conectados a un mundo w mediante una relación R, consideramos un conjunto de colecciones de mundos conectados a w.

Está formalmente definido como $\langle W, N \rangle$ 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. Su semántica está dada por: $\Box A$ es verdadero en w sii el conjunto de elementos de W donde A es verdadero es uno de los conjuntos en $N(w)$.

Nuestra implementación sigue esta definición:

NonNormalModality.java

```
public Boolean evalWith(Formula formula, World world
   , Agent agent) {
  for (ScottMontagueRelationship rel : relationships
     ) {
    if (rel.getAgent().equals(agent) && rel.getWorld
       ().equals(world)){
```

```
(1)     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){
           return true;
         }
       }
       return false;
     }
   }
   return true;
}
```

El tiempo de ejecución de (1) es $N * n * T_{eval}(k-1)$

(1) se ejecuta una vez por cada par $(mundo, agente)$, por lo cual el orden de ejecución (**asumiendo 1 agente**) es: $n * N * n * T_{eval}(k-1) \in \mathcal{O}(N * n^2 * T_{eval}(k-1))$

Dado que asumimos un modelo correcto, hay que notar que la fórmula que resta evaluar es o bien una instancia ground o una fórmula booleana y su cardinalidad es $\mathcal{O}(k)$. Con lo cual el orden de ejcución de $T_{eval}(k-1)$ en el peor caso es $\mathcal{O}(2 * (k-1)) \in \mathcal{O}(k)$

Notar además que en el peor de los casos $N = POW(n) = 2^n$, por lo que el tiempo de ejecución es $\mathcal{O}(N * n^2 * k) \in \mathcal{O}(2^n)$

5.2. Propuestas de mejoras

Para $M, w \vDash \phi$, podemos determinar la *profundidad modal* de la fórmula ϕ ("Constructing the least models for positive modal logic programs" [NG2000]), y a partir de esa medida cortar la profundidad del modelo M, achicando de esa manera el espacio de búsqueda.

Escalabilidad horizontal: es posible distribuir el procesamiento en varios nodos para mejorar la performance. Dada la naturaleza del algoritmo esto no sería difícil de hacer.

Capítulo 6

Estado del arte en chequeadores de modelos para MAS

La mayoría de los chequeadores de modelos para sistemas multi-agentes de la actualidad son orientados a lógicas temporales (LTL: Linear Temporal Logic, CTL: Computational Tree Logic o CTL*: Full Branching Time Logic) y algunos incluyen modalidades epistémicas. En general las técnicas para chequear estas lógicas son basadas en Diagramas de decisión binarios (BDD: Binary Decision Diagrams), chequeo restringido de modelos (BMC: Bounded Model Checking) o en una reducción al problema de satisfactibilidad (SAT).

Esta sección da una visión general de algunos chequadores de modelos para lógicas multi-agentes existentes:

6.1. MCK

MCK es una herramienta de chequeo de modelos para sistemas multi agentes desarrollada en la School of Computer Science and Engineering en la Universidad de New South Wales. Usa principalmente BDDs pero también soporta Bounded Model Checking y chequeo explícito de modelos (ESMC: Explicit State Model Checking). Soporta las siguientes lógicas para especifi-

car propiedades:

- LTL, CTL, CTL*
- μ-calculus
- Modalidades epistémicas incluído "conocimiento general" (*common knowledge*)
- Restricciones de equidad

Adicionalmente a la semática observacional, MCK soporta otras semánticas como *clock*, *asynchronous perfect recall* y *synchronous perfect recall*. Provee una interfaz gráfica de usuario y generación de contraejemplos. Está implementado en Haskell.

6.2. MCMAS

Está basado en BDD para la verificación de sistemas multi-agentes, fue desarrollado por el Imperial College London y distribuído bajo la licencia GNU General Public Licence (GPL). Las descripciones de los sistemas (MAS), son dadas en el lenguaje de programación ISPL (Interpreted Systems Programming Language). Soporta las siguientes lógicas:

- CTL con restricciones de equidad
- ATL con restricciones de equidad
- Modalidades epistémicas
- Modalidades deónticas para expresar el correcto comportamiento de un agente.

También soporta la generación de contraejemplos y provee una interfaz de usuario como un plugin de Eclipse, cuya funcionalidad soporta la edición de ISPL con chequeo de sintaxis dinámico, un modo de ejecución interactiva y la visualización de casos testigos o contraejemplos. Está implementado en C++.

6.3. Mocha

Está basado en BDD desarrollado en conjunto en la Universidad de California en Berkley, la Universidad de Pensilvania y la Universidad estatal de Nueva York en Stony Brook. Difiere de los chequeadores de modelos tradicionales en que su propósito principal es facilitar el desarrollo de nuevas técnicas de verificación. Los sistemas son modelados usando módulos reactivos, los cuales representan sus componentes sincrónicos, asincrónicos y de tiempo real. [Mocha] soporta los siguientes formalismos para especificar propiedades:

- ATL
- Invariantes (fórmulas proposicionales que deben ser verdaderas en todos los estados)
- Módulos abstractos que deben ser implementados por el sistema (refinamiento)

Provee una interfaz de usuario para simulación interactiva. Actualmente existen 2 versiones cMocha y jMocha, implementados en C y Java respectivamente. El último actualmente no soporta el chequeo de modelos ATL.

6.4. NuSMV

Es una herramienta de código abierto liberada bajo la licencia GNU Lesser GEneral Public License 2.1 (LGPL) que soporta el chequeo de modelos con BDD y SAT. Es una reimplementación y extensión de otro chequeador llamado SMV. Procesa archivos en el formato SMV, el cual primero se traduce a una máquina de estados finita (FSM) y luego realiza el chequeo con BDD, o un chequeo con basado en SAT: BMC (Bounded Model Checking). La herramienta soporta las siguientes lógicas:

- LTL
- CTL con restricciones de equidad

- RTCTL (Real Time CTL) que agrega restricciones de tiempo real a CTL
- un subconjunto de PSL (Property Specification Language)

Soporta la generación de contraejemplos y prove un shell interactivo. Está implementado en C.

6.5. PRISM

Es un chequeador de modelos probabilístico bajo la licencia GNU General Public License. Usa varias técnicas incluídas BDDs, MTBDDs, simulación de eventos discretos, refinamiento por abstracción cuantitativa y reducción de simetría. Soporta un amplio rango de modelos:

- Determinista discreta: Cadenas de tiempo discreto de Markov (DTMCs)
- No determinista discreta: proceso de decisión de Markov (MDPs) y autómatas probabilísticos (PAs)
- Determinista discreta: Cadenas de tiempo contínuo de Markov (CTMCs)
- No determinista contínua: Autómata de tiempo probabilístico (PTAs) y autómata de tiempo valuado probabilístico (PPTAs)

y extensiones de estos modelos con costos y beneficios. El lenguaje de especificación subsume las siguientes lógicas:

- PCTL (Probabilistic computation tree logic) y PCTL* que aumentan CTL y CTL* con límites probabilísticos
- CSL (Lógica estocástica contínua) para sistemas contínuos inspirado por CTL
- LTL
- Un subconjunto de CTL

Provee la generación de adversario óptimo o estrategia óptima para modelos no deterministas y una interfaz de usuario. Esta implementado en una mezcla de Java y C++

6.6. VerICS

Es una herramienta para el chequeo de modelos basada en SAT desarrollada en el Instituto de Ciencias de la Computación de la Academia de Ciencias de Polonia. Usa Bounded Model Checking (BMC) y Unbounded Model Checking (UMC) para verificar modalidades temporales, epistémicas y deónticas en autómatas temporales y redes de Petri temporales. Las lógicas que soporta son:

- $\text{CTL}_p K$: CTL con modalidad temporal de pasado y la epistémica de conocimiento (K), usando UMCinsit
- EC_{TLKD}: fragmento existencial de CTL con conocimiento y modalidades deónticas usando BMC
- TEC_{TLK}: fragmento existencial de CTL temporal con conocimiento usando BMC

Proporciona una interfaz de usuario para modelar autómatas temporales y las redes de Petri temporales. Está implementado en Java.

6.7. Otras herramientas para MAS

A continuación enumeramos otras herramientas que no implementan un chequeador semántico de modelos, pero que por estar relacionadas con lógicas modales multi-agentes resulta interesante tener en cuenta para analizar otros enfoques:

6.7.1. LoTREC

LoTREC permite manejar casi cualquier lógica con semántica de Kripke, o más generalmente una semántica basada en transiciones, mientras un constructor de modelos para esta pueda ser expresado como un sistema de reescritura de grafos, en otras palabras, mientras sea posible expresar sus restricciones semánticas con reglas de reescritura de grafos.

Está desarrollado en Java por el Instituto de Investigación en informática de Toulouse (IRIT) y es provisto con sus fuentes y una interfaz gráfica: http://www.irit.fr/Lotrec

6.7.2. MAELIA

El projecto MAELIA (http://maelia1a.wordpress.com), es una plataforma digital para la similación de efectos socio-ambientales de la implementación de normas relacionadas al gobierno y manejo de aguas y otros recursos, territorios y el medio ambietne.

Estas normas (red conjunta de reglas sociales y legales, redes de interación de organizaciones y partes interesadas, uso de recursos, etc.) son la expreción mas importante de objetivos politicos públicos y reflejan las aspiraciónes de ciertos grupos sociales. Son una herramienta esencial para una búsqueda efectiva de desarrollo sustentable, pero también puéden ser la causa principal del exito o fracaso de dicha búsqueda.

6.7.3. G A M A

G A M A (https://code.google.com/p/gama-platform) es un ambiente de modelado y desarrollo de simulaciones para construir simulaciones basadas en agentes explicitamente espaciales. Soporta:

- Construir grandes modelos definidos en el lenguaje orientado a agentes GAML, como opcional ofrece una herramienta gráfica de modelado para soportar diseños agiles y prototipado.

- Intanciar agentes desde información GIS, base de datos o archivos y ejecutar simulaciones a gran escala (más de un millon de agentes).
- Acoplamiento entre un mismo modelo discreto o capas topologicamente continuas, multiple niveles de agentes y paradigmas multiples (ecuaciones matematicas, arquitecturas de control, maquinas de estados finitos).
- Definir diferentes experimentos en modelos, con distintas entradas y salidas, y explorar el espacio de parametros para calibración y validación.
- Diseñar interfaces de usuarios que soporten inspeciones profundas en agentes, aciones y paneles controlados por el usuario, y mostrarlo en multiples capas 2D/3D.

Capítulo 7

Conclusiones y trabajo futuro

Hemos implementado en un lenguaje orientado a objetos, un chequeador de modelos para la lógica *N(Does)* extensible a otras lógicas modales multi-agentes, usando únicamente la semántica de mundos posibles (o de Kripke) y la semántica de Scott-Montague, analizado su funcionamiento con algunos ejemplos. El chequeador fue diseñado desde el punto de vista de la programación orientada a objetos, de manera modular y extensible para otras modalidades distintas de las vistas en el capítulo 3. Esto le da flexibilidad para que sirva de plataforma para explorar chequeadores de otras combinaciones de lógicas modales. Si bien existen otras implementaciones de chequeadores de modelos, en su mayoría fueron diseñados para un tipo especifico de lógica, generalmente para modalidades temporales o epistémicas. El hecho de hacer un chequeador para una lógica poco usual, distinta a la lógica utilizada en otras implementaciones, nos hizo pensar el problema desde cero. Esto a su vez nos permitió idear una manera de resolver el problema sin estar influenciados por sistemas similares, pero deja abiertas varias líneas de investigación para seguir desarrollando:

- Definir un lenguaje para la teoría de confianza colectiva expuesta en [SR2010] que sirva para ingresar un frame y una fórmula de *N(Does)* al chequeador. Muchos de los chequeadores analizados en el capítulo 6 proveen sus propios lenguajes adaptados (en general para lógicas temporales). Esto facilitaría la instanciación del modelo y garantizaría (o por lo menos en algunos casos) su correctitud por construcción.

- Optimizar nuestro chequeador para reducir el uso de memoria como lo hacen los chequeadores de lógicas temporales presentados y optimizar la búsqueda en el grafo de relaciones de accesibilidad. Hay bastante material para analizar sobre algoritmos enfocados en resolver estos problemas en lógicas particulares que podrían llegar a aplicarse a este chequeador. En [DVDK07] se proponen algunos métodos para reducir la complejidad, como limitar la profundidad de las fórmulas modales o restringir el número de átomos proposicionales.

- Distribuir el chequeo de una fórmula por localidad o mundo, de manera de poder combinar varias computadoras o nodos de procesamiento y así aprovechar su potencia de cómputo combinada. Por ejemplo: chequear $\Box P$ en w que implica chequear que P es verdadero en todos los w' relacionados a w puede ser paralelizdo y distribuido.

Apéndice A

Apéndice: Ejemplos de uso del chequeador de modelos

Si bien no es necesario validar la semántica, decidimos hacer casos de test unitarios donde validamos el comportamiento esperado del chequeador para las distintas modalidades de modo que sirvan en caso de hacer modificaciones para detectar errores de código que pudieran influenciar en el correcto funcionamiento del chequeador:

Test para el model checker

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

  @Before
  public void setup() {
    frame1 = FrameBuilder.withAgents("a1", "a2")
        .withWorlds("w1", "w2", "w3")
        // valuation:
        .withLiteralValid("A", "w1", "w2")
        .withLiteralValid("B", "w2", "w3")
        // agent 1 Bel relationships
```

```
        .belRelationship("a1", "w1", "w2", "w3")
        .belRelationship("a1", "w2", "w1")
        // agent 2 Bel relationships
        .belRelationship("a2", "w2", "w1")
        // int relationships
        .intRelationship("a1", "w1", "w3")
        .intRelationship("a1", "w2", "w3")
        .intRelationship("a2", "w2", "w3")
        .intRelationship("a2", "w3", "w2", "w1")
        // goal relationships
        .goalRelationship("a1", "w3", "w1")
        .goalRelationship("a1", "w2", "w1")
        // does relationships
        // create a does relationship from w1 to the
           neighbor of w2 + w3 for agent 1
        .doesRelationship("a1", "w1", "w2", "w3")
        .doesRelationship("a1", "w1", "w1", "w2")
        .doesRelationship("a1", "w2", "w2", "w3")
        // create a does relationship from w3 to the
           neighbor of w1 + w2 for agent 1
        .doesRelationship("a1", "w3", "w1", "w2")
        // create a does relationship from w1 to the
           neighbor of w2 + w3 for agent 2
        .doesRelationship("a2", "w1", "w2", "w3")
        .build();

    this.mc = new ModelChecker();
}

@Test
@Ignore // this is just a test of how visualization works
public void test_graph_with_jung() throws
    IOException{
    FrameViewer.view(frame1);
    System.in.read();
}

@Test
```

```
public void a_and_b_is_true_in_w2_in_frame1() {
  mc.setFrame(frame1);
  LOGGER.info(frame1);
  Formula aAndB = new And(new Literal("A"), new
     Literal("B"));
  List<World> validInWorlds = mc.check(aAndB);
  LOGGER.info(aAndB + " valid in: " +
     validInWorlds);
  Assert.assertEquals(1, validInWorlds.size());
  Assert.assertEquals("w2", validInWorlds.get(0).
     getName());
}

@Test
public void a_or_B_is_true_in_all_worlds_in_frame1
   () {
  mc.setFrame(frame1);
  LOGGER.info(frame1);
  Formula aOrB = new Or(new Literal("A"), new
     Literal("B"));
  List<World> validInWorlds = mc.check(aOrB);
  LOGGER.info(aOrB + " valid in: " + validInWorlds
     );
  Assert.assertEquals(frame1.getWorlds().size(),
     validInWorlds.size());
}

@Test
public void
   a_implies_B_is_true_in_w2_and_w3_in_frame1() {
  mc.setFrame(frame1);
  LOGGER.info(frame1);
  Formula aImpliesB = new Implies(new Literal("A")
     , new Literal("B"));
  List<World> validInWorlds = mc.check(aImpliesB);
  LOGGER.info(aImpliesB + " valid in: " +
     validInWorlds);
  Assert.assertEquals(2, validInWorlds.size());
```

```
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w2")));
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w3")));
  }

  @Test
  public void not_a_is_true_in_w3_in_frame1() {
    mc.setFrame(frame1);
    LOGGER.info(frame1);
    Formula notA = new Not(new Literal("A"));
    List<World> validInWorlds = mc.check(notA);
    LOGGER.info(notA + " valid in: " + validInWorlds
        );
    Assert.assertEquals(1, validInWorlds.size());
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w3")));
  }

  @Test
  public void a_iff_B_is_true_in_w2_in_frame1() {
    mc.setFrame(frame1);
    LOGGER.info(frame1);
    Formula aIffB = new Iff(new Literal("A"), new
        Literal("B"));
    List<World> validInWorlds = mc.check(aIffB);
    LOGGER.info(aIffB + " valid in: " +
        validInWorlds);
    Assert.assertEquals(1, validInWorlds.size());
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w2")));
  }

  @Test
  public void bel_A_is_true_in_w2_for_a1_in_frame1()
      {
    mc.setFrame(frame1);
    LOGGER.info(frame1);
```

```
    Formula belA1A = new ModalFormula(frame1.
        getModalty(Bel.class), "a1", new Literal("A")
        );
    List<World> validInWorlds = mc.check(belA1A);
    LOGGER.info(belA1A + " valid in: " +
        validInWorlds);
    Assert.assertEquals(2, validInWorlds.size());
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w2")));
    /* it's also valid in w3 because w3 has no
        adjacent for Bel_a1  */
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w3")));
}

@Test
public void int_A_is_true_at_w3_for_a1_in_frame1()
    {
    mc.setFrame(frame1);
    LOGGER.info(frame1);
    Formula intA1A = new ModalFormula(frame1.
        getModalty(Int.class), "a1", new Literal("A")
        );
    List<World> validInWorlds = mc.check(intA1A);
    LOGGER.info(intA1A + " valid in: " +
        validInWorlds);
    Assert.assertEquals(1, validInWorlds.size());
    /* it's valid in w3 because w3 has no adjacent
        for Int_a1 */
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w3")));
}

@Test
public void
    goal_A_is_true_in_w2_and_w3_for_a1_in_frame1()
    {
    mc.setFrame(frame1);
```

```
    LOGGER.info(frame1);
    Formula goalA1A = new ModalFormula(frame1.
        getModalty(Goal.class), "a1", new Literal("A"
        ));
    List<World> validInWorlds = mc.check(goalA1A);
    LOGGER.info(goalA1A + " valid in: " +
        validInWorlds);
    Assert.assertEquals(3, validInWorlds.size());
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w2")));
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w3")));
    /* it's also valid in w1 because w1 has no
        adjacent for Goal_a1 */
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w1")));
}

@Test
public void
    does_a1_A_is_true_in_w1_and_w3_in_frame1() {
    mc.setFrame(frame1);
    LOGGER.info(frame1);
    Formula doesA1A = new ModalFormula(frame1.
        getModalty(Does.class), "a1", new Literal("A"
        ));
    List<World> validInWorlds = mc.check(doesA1A);
    LOGGER.info(doesA1A + " valid in: " +
        validInWorlds);
    Assert.assertEquals(2, validInWorlds.size());
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w1")));
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w3")));
}

@Test
```

```
public void
    bel_a1_A_is_true_in_w2_by_belief_generalization_in_frame1
    () {
  mc.setFrame(frame1);
  LOGGER.info(frame1);
  Literal a = new Literal("A");
  World w2 = frame1.getWorld("w2");
  Assert.assertTrue("A should be true in w2", mc.
     check(a, w2));

  Formula belA1A = new ModalFormula(frame1.
     getModalty(Bel.class), "a1", a);
  List<World> validInWorlds = mc.check(belA1A);
  LOGGER.info(belA1A + " valid in: " +
     validInWorlds);
  Assert.assertTrue("Bel(a1, A) should be true in
     w2", validInWorlds.contains(w2));
}

@Test
public void
    bel_a2_A_is_true_in_w2_by_belief_generalization_in_frame1
    () {
  mc.setFrame(frame1);
  LOGGER.info(frame1);
  Literal a = new Literal("A");
  World w1 = frame1.getWorld("w1");
  World w2 = frame1.getWorld("w2");
  Assert.assertTrue("A should be true in w1", mc.
     check(a, w1));
  Assert.assertTrue("A should be true in w2", mc.
     check(a, w2));

  Formula belA2A = new ModalFormula(frame1.
     getModalty(Bel.class), "a2", a);
  List<World> validInWorlds = mc.check(belA2A);
  LOGGER.info(belA2A + " valid in: " +
     validInWorlds);
```


```
    Assert.assertTrue("Bel(a2, A) should be true in
        w2", validInWorlds.contains(w2));
  }

  @Test
  public void
     int_a2_does_a1_A_is_true_in_w2_and_w1_in_frame1
     () {
    mc.setFrame(frame1);
    LOGGER.info(frame1);
    Formula belA1DoesA2A = new ModalFormula(frame1.
        getModalty(Int.class), "a2", new ModalFormula
        (frame1.getModalty(Does.class), "a1", new
        Literal("A")));
    List<World> validInWorlds = mc.check(
        belA1DoesA2A);
    LOGGER.info(belA1DoesA2A + " valid in: " +
        validInWorlds);
    Assert.assertEquals(2, validInWorlds.size());
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w2")));
    /* it's also true in w1 since it has no int-
        related world for a2 */
    Assert.assertTrue(validInWorlds.contains(frame1.
        getWorld("w1")));
  }
}
```

Con algunos de los tests, fuimos capaces de detectar una inconsistencia entre la semántica procedural que analizamos y la semántica dada por las reglas de derivación en el frame brindado como ejemplo. En principio esto nos dice que el frame no pertenece a la clase de frames válidos para $N(Does)$, ya que de pertenecer debería existir una correspondencia entre ambas.

Bibliografía

[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.

[Akka] http://akka.io/.

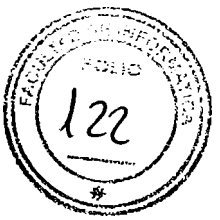
[BdRV01] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic.* Cambridge University Press, Cambridge, England, 2001.

[Blackburn] Patrick Blackburn, Johan F. A. K. van Benthem, and Frank Wolter. *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning).* Elsevier Science Inc., New York, NY, USA, 2006.

[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.

[GR04b] Guido Governatori and Antonino Rotolo. On the axiomatization of elgesem's logic of agency and ability. In *Advances in Modal Logic*, pages 130–144. University of Manchester, Department of Computer Science, 2004.

[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.

[HM92] Joseph Y Halpern and Yoram Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial intelligence*, 54(3):319–379, 1992.

[Hazelcast] http://hazelcast.com.

[JAVA] http://www.java.com.

[LO2009] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. Mcmas: A model checker for the verification of multi-agent systems. In *Computer Aided Verification*, pages 682–688. Springer, 2009.

[LO2014] Alessio R Lomuscio. A model checker for strategy logic. 2014.

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

[Mocha] Rajeev Alur, Thomas A Henzinger, Freddy YC Mang, Shaz Qadeer, Sriram K Rajamani, and Serdar Tasiran. Mocha: Modularity in model checking. In *Computer Aided Verification*, pages 521–525. Springer, 1998.

[NG2000] Linh Anh Nguyen. Constructing the least models for positive modal logic programs. *Fundamenta Informaticae*, 42(1):29–60, 2000.

[Rum91] J Rumbaugh. Object-oriented modeling and design, 1991.

[SP08] Lutz Schröder and Dirk Pattinson. The craft of model making: Pspace bounds for non-iterative modal logics. *arXiv preprint arXiv:0802.0116*, 2008.

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

[Scala] http://www.scala-lang.org/.

[TIOBE] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

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