Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Seminario de Tesis:

La Lógica Modal como Herramienta de Ingeniería de Software

Autores:

Areces, Carlos Eduardo
LU: 66/90
postmast@logia.uba.ar

Hirsch, Dan Francisco
LU: 267/90
dhirsch@compsc.uba.ar

Directores:

Felder, Miguel
Dr. en Cs. de la Computación

Yankelevich, Daniel
Dr. en Cs. de la Computación

1996

Indice

La Lógica Modal como Herramienta de Ingeniería de Software

Abstract

Se presentará una metodología para la representación de diseños de sistemas de software y para la verificación de propiedades sobre éstos. Los grafos de diseño son considerados modelos de una lógica modal extendida. Se enuncian las herramientas lógicas para derivar el modelo modal asociado a todo diseño, el algoritmo de chequeo de propiedades, la técnica de definición de nuevas relaciones y el método de filtración de modelos. La aplicación de estos métodos es demostrada en tres ejemplos prácticos de diverso tipo.

La metodología se compone de una lógica modal más un conjunto de herramientas. La lógica propuesta se denomina **KPI**, polimodal con operadores inversos (que en un modelo conexo permite acceso total) y será utilizada como lenguaje de especificación de propiedades a verificar mediante un algoritmo de model checking. Las herramientas son en verdad una adaptación de métodos tradicionales de la lógica (modal y clásica) que permiten realizar las tareas que usualmente se encuentran en el proceso de Ingeniería de Software. Los métodos propuestos son efectivamente implementados en un prototipo utilizando el lenguaje funcional SML/NJ.

1 Introducción

La Ingeniería de Software es el campo de la Ciencia de la Computación que trata con el diseño y la construcción de sistemas de software tan grandes y complejos que deben en general ser construidos por un grupo de personas. Normalmente, estos sistemas son usados por varios años y durante su vida sufren cambios tales como el arreglo de defectos, la incorporación de nuevas aplicaciones, la eliminación de viejas facilidades, o la migración a nuevos ambientes.

Como toda disciplina de ingeniería, el desarrollo de sistemas de software requiere del uso de herramientas. Pero el desarrollo exitoso de un sistema requiere la elección de herramientas adecuadas para lograr un propósito específico.

Los entornos de Ingeniería de Software deben proveer de herramientas que permitan al ingeniero comprender mejor el sistema que está diseñando. Estas herramientas deben ser a la vez poderosas, simples de usar y tan formales como sea necesario (o posible).

En las décadas del 40 y 50, las aplicaciones para computadoras eran pequeñas dadas las limitaciones de las máquinas donde eran ejecutadas. Al crecer las máquinas en capacidad y poder, también lo hicieron los requerimientos de los usuarios. Y para satisfacer estos requerimientos fueron necesarias herramientas cada vez más sofisticadas de análisis e implementación de sistemas.

Una de las nuevas herramientas de programación fue el flowchart o diagrama de flujo de control, que fue considerado una herramienta útil para organizar ideas antes de comenzar la programación. Al crecer los sistemas, las dificultades de los flowcharts fueron evidentes: los flowcharts sólo lograban proveer un claro conocimiento de la organización en sistemas pequeños.

El término "Ingeniería de Software" fue inventado al final de los 60 cuando se hizo evidente que todo lo aprendido acerca de la buena programación no ayudaba a la construcción de mejores sistemas de software. Mientras que el campo de la programación había realizado grandes avances mediante el estudio sistemático de las estructuras de datos y la introducción de la programación estructurada, estas herramientas y técnicas no lograban resolver dificultades en la construcción de grandes sistemas de software.

Lo que se necesitaba en estos casos complejos era utilizar el método clásico de ingeniería: definir primero claramente el problema y luego utilizar técnicas y herramientas standard para resolverlo.

Se detectó entonces que existían varias tareas que mejoraban la comprensión de un sistema y que debían realizarse antes de comenzar a dibujar un flowchart. La etapa de implementación y testeo más el conjunto de estos pasos previos dieron origen a la idea de "Ciclo de Desarrollo de Software". Tradicionalmente este ciclo está compuesto por las distintas etapas que atraviesa el software durante su "vida". Se comienza con una especificación de requerimientos, luego se alcanza la etapa de diseño, a continuación el sistema es implementado y testeado, para

finalmente ser puesto en uso hasta su eliminación. Si se detectan errores (durante la etapa de testeo o posteriormente) o si aparecen nuevos requerimientos, el sistema deberá atravesar nuevamente algunas de las etapas del ciclo. En nuestro trabajo, le dedicaremos especial atención a la Etapa de Diseño.

Luego de obtener una especificación del problema a tratar (en un lenguaje formal en mayor o menor medida), deben enunciarse las líneas generales que definen los componentes del sistema, las diversas inter-relaciones entre ellos, las propiedades o restricciones que los componentes deben satisfacer, etc. Estas líneas generales conforman el Diseño del problema.

Diferentes métodos se han utilizado para llevar a cabo el Diseño. Los más simples entre ellos usan gráficos, anotaciones y hasta diferentes colores para sugerir una idea. Pero los diseños así obtenidos son ambiguos y demostrar sus propiedades es casi imposible. La gran variedad y la libertad de estilo involucrados los vuelven inadecuados para la demostración formal.

Una importante mejora sobre esta situación fue la definición y, principalmente, la aceptación general de un conjunto de herramientas específicamente definidas para tratar este problema, como los diagramas de Entidad-Relación, los Diagramas de Flujo de Datos, los Gráficos de Estructuras, etc., que constituyen lo que se denomina el Diseño Estructurado.

El Diseño Estructurado fue la primer técnica en reflejar y manejar la complejidad de los sistemas modernos. Su principal aporte fue el reconocer que para atacar los detalles de un problema, primero es muy importante tener un claro entendimiento del problema a resolver. Para poder llevar esto a cabo, proveyó de un pequeño conjunto de herramientas para la descripción semi-formal del problema.

A pesar de que estas herramientas eliminaron la variedad, no excluyeron la ambigüedad, y la demostración de propiedades continuó siendo imposible.

En pocas palabras, un nuevo lenguaje global para la descripción de diseños había sido establecido y se había optado por un lenguaje gráfico como la mejor opción. Pero hacía falta aún, una semántica formal para este lenguaje.

Actualmente, muchos investigadores han estado trabajando en diversos aspectos de este problema usando nuevas técnicas. El trabajo de [Cosens et al., 1992], por ejemplo, tiene por objeto proveer herramientas para manipular y obtener visiones simplificadas de los diseños, que ayuden una mejor comprensión del sistema. Otros ejemplos de trabajos en esta línea pueden ser los de [Agustí et al., 1995] que trata la especificación de alto nivel de programas lógicos utilizando operaciones sobre conjuntos mediante una interface gráfica; [Paul & Prakash, 1996] quienes utilizan un álgebra de consulta para reconstruir el diseño a partir del código de un sistema (reingeniería) y el enfoque más tradicional de [Maibaum et al., 1984] en donde a partir de una especificación de alto nivel utilizando TADs, se propone un esquema de composición y refinamiento de módulos (teoría de implementación) a través de una lógica de primer orden extendida.

Nuestra opinión sin embargo, es que debemos realizar un salto hacia un nivel de mayor abstracción. Además de una herramienta que nos provea de visiones más simplificadas del sistema queremos poder razonar sobre los gráficos de diseño en una forma más abstracta. Este nivel superior de abstracción se obtiene cuando consideramos a los diseños como estructuras modales. Pensamos que el uso de un lenguaje modal para describir tanto el diseño como sus propiedades nos dará la expresividad exacta que estamos buscando.

La elección de una lógica modal para la descripción de diseños está justificada por la similitud existente entre los modelos de ambas áreas: grafos dirigidos con ejes y nodos etiquetados. Pero además, el uso que se da a ambas estructuras también coincide, dando mayor importancia a las relaciones quienes determinan las propiedades del diseño. La utilización de una lógica modal para la descripción de los propios diseños provee de un lenguaje común para caracterizar el modelo y sus propiedades, con capacidad de demostración formal, una semántica no ambigua y la posibilidad de realizar verificación automática.

2 La Etapa de Diseño en Ingeniería de Software

2.1 Conceptos Básicos

Las siguientes secciones están dedicadas principalmente a la introducción de los conceptos fundamentales de módulo y relación entre módulos. Estos conceptos serán más adelante tratados bajo el nuevo formalismo modal y también allí tendrán un rol principal. Además de dar sus definiciones formales se comentan los criterios habituales que utiliza la Ingeniería de Software para evaluar la calidad de un diseño. Las nuevas herramientas derivadas de la lógica modal tendrán por objeto contribuir al proceso de verificación de estos criterios.

Siempre es difícil dar en el área de diseño conceptos que sean universalmente aceptados, simplemente porque no conciernen a cuestiones estrictamente formales. Hemos elegido por ello aquellos criterios sobre los que existe consenso [Parnas, 1972; Parnas 1979; Page Jones, 1980; Ghezzi et al., 1991], y que conforman los "criterios mínimos" de buen diseño.

Principios de la Ingeniería de Software

Para el correcto desarrollo de software deben seguirse ciertos principios generales que son fundamentales. Estos principios se aplican tanto al *proceso* de desarrollo del software como al *software* mismo.

- *Rigor y formalidad:* El desarrollo de software es una actividad creativa, pero el rigor es un complemento necesario para la creatividad. Sólo a través de un enfoque riguroso es posible producir productos confiables y controlar costos. Durante el análisis de un proceso, varios niveles de rigor se pueden requerir, de los cuales el más alto es el de formalización, donde se exige que el proceso de software sea evaluado y dirigido por leyes matemáticas. No siempre es necesario ser formal durante el diseño, pero el ingeniero debe saber cómo y cuándo serlo.
- *Separación de intereses y responsabilidades:* Permite trabajar en forma independiente con diferentes aspectos individuales de un problema, reduciendo la complejidad inherente del desarrollo de software.
- *Modularidad:* Un sistema complejo puede ser dividido en piezas más simples llamadas módulos. El principal beneficio de una buena modularización es que permite que el principio de separación de intereses y responsabilidades sea aplicado en dos fases: cuando se está trabajando con los detalles de cada módulo por separado (e ignorando los detalles de los otros módulos) y cuando se trabaja con las características generales de todos los módulos y sus relaciones para integrarlos en un sistema coherente.
- *Abstracción:* La abstracción es un proceso en donde se identifican los aspectos relevantes de un fenómeno y se ignoran los detalles. Es decir que éste es un caso especial de separación de intereses entre los aspectos importantes y los detalles puntuales.
- *Anticipación de cambio:* El software sufre cambios constantes. Esto puede ser debido a errores que hay que corregir, a la necesidad de dar soporte a la evolución de la aplicación a medida que nuevos requerimientos aparecen o porque los viejos requerimientos cambian. El anticipo de cambio afecta dos importantes cualidades del software: simplicidad de mantenimiento y reusabilidad.
- *Generalidad:* Cada vez que se quiere resolver un problema se debe tratar de descubrir el problema más general oculto detrás del primero. Puede suceder que el problema general no sea más complejo (y hasta puede ser más simple) que el original. Siendo más general es probable que la solución de éste tenga mayor reusabilidad. También puede suceder que al generalizar el problema exista una solución en alguna librería ya creada. Por otro lado, una solución generalizada puede resultar más costosa (en términos de velocidad de ejecución, requerimientos de memoria, o tiempo de desarrollo) que una solución especializada. Por lo tanto es necesario evaluar el costo de la generalización en el momento de decidir su aplicación.
- *Incrementalidad:* La incrementalidad caracteriza a un proceso que avanza en forma escalonada. El resultado deseado es alcanzado mediante sucesivas aproximaciones, donde cada aproximación es obtenida por una mejora del paso anterior.

A pesar de su claridad, los principios no son suficientes para dirigir el desarrollo de software. Los principios son descripciones de propiedades deseables de los procesos y productos de software, pero es necesario proveer al ingeniero de software con métodos y técnicas específicas que le permitan incorporar dichos principios a su trabajo.

Los métodos son líneas generales que gobiernan la ejecución de una actividad en forma rigurosa, sistemática y disciplinada. Las técnicas son más mecánicas que los métodos y tienen, a menudo, una aplicación más restringida.

A veces, los métodos y técnicas son agrupados juntos para formar una metodología. El propósito de una metodología es promover una manera de resolver un problema mediante la preselección de métodos y técnicas a ser usados. Las herramientas son desarrolladas para soportar la aplicación de métodos, técnicas y metodologías.

Existe entonces una relación de varias capas: los principios son la base y luego se tienen los métodos y técnicas, las metodologías y finalmente las herramientas.

Módulos y Modularización

La modularización es propuesta como un método para mejorar la flexibilidad y claridad de un sistema y al mismo tiempo acortar su tiempo de desarrollo [Parnas, 1972].

El mismo Parnas cita a [Gouthier & Pont, 1970] sobre el diseño de sistemas:

"Una segmentación bien definida del esfuerzo del proyecto asegura la modularidad del sistema. Cada tarea forma un modulo de programa separado y distintivo. En tiempo de implementación, cada módulo y sus entradas y salidas se hayan bien definidas, no hay confusión en la interface con otros módulos del sistema. En tiempo de prueba la integridad del módulo es testeada en forma independiente; también se reducen los problemas de scheduling en la sincronización de varias tareas antes de que las pruebas puedan comenzar. Finalmente, el sistema es mantenido en forma modular; errores y deficiencias del sistema pueden ser rastreados a módulos específicos, limitando de esta manera el alcance de una búsqueda detallada de errores."

Sin embargo, la efectividad de una modularización depende de los criterios usados para dividir al sistema en módulos y, usualmente, nada se dice acerca de cómo estos criterios deben ser utilizados.

Los beneficios esperados de una programación modular son:

- *Mejor Administración:* el tiempo de desarrollo es menor dado que grupos separados trabajan en cada módulo, con poca necesidad de comunicación entre ellos.
- *Más Flexibilidad:* es posible hacer cambios drásticos en un módulo sin tener que cambiar otros.
- *Mayor Comprensión:* puede estudiarse el sistema módulo por módulo, logrando de esta manera que el sistema completo este mejor diseñado gracias a un buen entendimiento de cada una de sus partes.

En los párrafos anteriores, módulo no es equivalente a subprograma sino a asiganción de responsabilidad. Las partes de un sistema se agrupan en unidades que realizan una tarea en común o que están conceptualmente relacionadas.

El proceso de modularización involucra también las decisiones de diseño que deben ser hechas antes de que pueda comenzar el trabajo en cada módulo específico. Aunque en cada paso de la modularización deberán tomarse decisiones de distinto tipo, todas ellas deben reflejar las decisiones a nivel sistema (decisiones que afectan a más de un módulo) que ya fueron tomadas.

Uno de los métodos que han sido más utilizados para llevar a cabo la modularización es asignar módulos de acuerdo a las distintas tareas que pueden observarse en el proceso de resolución del problema. Se podría decir que esta descomposición está basada en el flowchart del proceso. El criterio usado es hacer que cada paso relevante en el proceso sea un módulo.

Otra posible descomposición se obtiene utilizando *Information Hiding* (IH) como criterio. Los módulos ya no corresponden a pasos del proceso sino que cada módulo está caracterizado por su conocimiento de una determinada decisión de diseño, la cual oculta del resto de los módulos.

Hay que notar que un sistema construido de acuerdo con la primera descomposición puede ser idéntico, luego de ser ensamblado, a un sistema construido de acuerdo a la segunda descomposición. Por ejemplo, las dos

descomposiciones pueden compartir todas las representaciones de datos y métodos de acceso. Las diferencias entre las dos alternativas se encuentran en la manera en que ellas influyen en la asignación de trabajo y en la definición de las interfaces entre módulos. Los sistemas son substancialmente diferentes aunque fuesen idénticos en la representación ejecutable. Esto es debido a que la representación ejecutable es sólo una de las utilizadas; otras representaciones son usadas para cambio, documentación, comprensión, etc. Los dos sistemas no serán idénticos en estas otras representaciones.

En la actualidad, el método de IH es el usualmente elegido. La complejidad de los sistemas implican concurrencia y distribución, donde el flujo de control pierde sentido, haciendo imposible el uso del primer criterio. Otra razón en favor de IH es que el conocimiento explícito del flujo de control reduce la flexibilidad del sistema complicando posibles cambios que afecten dicho flujo.

Además del criterio general de que cada módulo esconda alguna decisión de diseño del resto del sistema, pueden mencionarse algunos ejemplos específicos de descomposiciones que pueden resultar de ayuda:

- Una estructura de datos, su representación interna, sus procedimientos de acceso y modificación deben estar en un mismo módulo.
- La secuencia de instrucciones necesarias para llamar a una rutina dada y la rutina misma deben estar en un mismo módulo. (Relevante principalmente al programar a bajo nivel).
- Códigos de caracteres, órdenes alfabéticos, y datos similares deben ser ocultos en un módulo para obtener mayor flexibilidad.
- La secuencia en la cual ciertos elementos serán procesados deben (tanto como sea posible) estar ocultos dentro de un mismo módulo.

En este trabajo se usará siempre una noción muy amplia de módulo. Nada concreto acerca de su comportamiento puede asumirse del hecho que algo es un módulo; sólo será una parte del diseño que estará caracterizada en forma unívoca por un nombre. Las que definirán el rol de cada módulo en el diseño serán las relaciones que entre ellos se establezcan. Esto justamente es lo que la lógica modal captura a través de sus operadores, agregando a la información propia del módulo (información proposicional), la información que se deriva de su interacción con el resto del diseño (información modal).

La forma elegida de representar a los módulos condiciona el tipo de propiedades de diseño que podrán ser analizadas vía el formalismo. Propiedades que no dependan de la interacción entre los módulos o cuestiones estilísticas, por ejemplo, escapan a esta formalización.

Relaciones entre Módulos

Un sistema de software puede ser visto como un conjunto de partes que dependen unas de otras. Ejemplos típicos de partes son: módulos, programas, funciones, variables, tipos de datos y constantes. Ejemplos típicos de relaciones de dependencia son: uso, llamado de funciones (invocación), acceso de variables, asignación de tipos a variables, etc.

Una caracterización más formal de algunas de estas relaciones puede encontrarse en [Parnas, 1979]:

"Si se considera a un sistema como dividido en un conjunto de programas que pueden ser invocados por el flujo normal de los mecanismos de control, por interrupciones, o por mecanismos manejadores de excepciones y asumimos que cada uno de estos programas tiene una especificación que define exactamente el efecto que debe tener su ejecución, puede decirse, dados dos programas A y B, que A usa B si la correcta ejecución de B es necesaria para que A complete la tarea descripta en su especificación".

Muchas veces las relaciones *usa* e *invoca* coinciden, sin embargo estas relaciones difieren en al menos dos formas:

- Ciertas invocaciones pueden no ser instancias de usa. Si la especificación de A solo requiere que A invoque B cuando ciertas condiciones se dan, entonces A cumple con su especificación cuando éste genera una llamada correcta a B. A es correcta a pesar que B sea incorrecta o no esté presente. Una prueba de correctitud de A sólo necesita hacer suposiciones acerca de la manera de invocar a B.

• Un programa A puede usar a B aunque nunca lo invoque. La mejor ejemplificación de esto es el manejo de interrupciones. Usa también puede ser formulada como "requiere la presencia de una versión correcta de".

Algunos ejemplos de la relación usa son:

• Un tipo de uso no estructurado ocurre cuando un módulo modifica datos locales (o hasta instrucciones) de otro módulo. Esto puede ocurrir en el caso de programas en lenguaje assembler.

• Un módulo puede usar a otro comunicándose con él a través de un área común de datos, como por ejemplo una variable estática de C o un bloque común de FORTRAN.

• En un ambiente concurrente, un módulo puede comunicarse con otro que requiere sus servicios a través de mensajes (es usual en este caso decir que el segundo módulo es cliente del primero en vez de decir que lo usa).

• Un módulo que presta servicios puede comunicarse con su cliente a través de parámetros de un subprograma. Esta es la manera tradicional y disciplinada de permitir que dos módulos secuenciales interactúen.

En discusiones sobre estructuras de sistemas es fácil confundir los beneficios de una buena modularización con los de una estructura jerárquica. Se tiene una estructura jerárquica si la relación usa puede ser definida entre los módulos de forma tal que dicha relación sea un orden parcial. La existencia de un orden parcial nos brinda principalmente dos beneficios:

Primero, partes del sistema son simplificadas porque utilizan los servicios de niveles inferiores. Y segundo, el sistema puede utilizarse aún sin los niveles superiores.

En muchos proyectos de diseño de software las decisiones acerca de qué otros componentes usar son dejadas a programadores individuales del sistema. Si un programador tiene conocimiento de un programa en otro módulo, y siente que le sería de utilidad en el suyo, puede incluir una llamada a ese programa en su texto. Los programadores son alentados a usar el trabajo de otros, dado que cuando cada programador escribe sus propias rutinas para realizar funciones habituales se termina con un sistema que es más grande y complejo de lo necesario.

Sin embargo, el empleo irrestricto de la relación usa hace que los módulos del sistema se vuelvan altamente interdependientes, provocando que no haya subconjuntos que se puedan usar antes de que todo el sistema sea completado. Si se diseña un sistema en el cual los módulos de "bajo nivel" hacen algún uso de los módulos de "alto nivel", no existe la jerarquía y será más difícil remover porciones del sistema. El resultado es un sistema en el cual nada funciona hasta que todo funciona y la palabra "nivel" pierde gran parte de su sentido.

Una dependencia acíclica entre módulos simplifica también el mantenimiento del código, pues un cambio en un módulo se propagará sólo "hacia arriba" en la jerarquía. Por último, un sistema con dependencias cíclicas es más complejo de entender, pues el sistema no tendrá elementos básicos o atómicos a partir de los cuales comenzar el análisis.

En ciertos casos la jerarquía entre módulos está obligada por factores externos. Por ejemplo, en algunos lenguajes (como Pascal) una unit sólo puede ser compilada luego de que todas las otras units de las que depende hayan sido compiladas.

Podemos ver que estructura jerárquica y buena modularización son dos propiedades deseables para la estructura de un sistema pero independientes entre sí, ya que es posible dar una modularización que no respeta los criterios de buena división en módulos, pero que mantiene la estructura jerárquica,

Un criterio para obtener una jerarquía usa que además respete los principios de buena modularización es la siguente. Permitiremos que A use B cuando todas las siguientes condiciones se cumplan:

• A es esencialmente más simple porque usa a B.

• B no es substancialmente más complejo porque no le es permitido usar A.

• Hay un subconjunto coherente del sistema (es decir, subconjunto útil) que contiene B y no A.

• No hay subconjunto coherente del sistema que contenga A pero no B.

A veces nos encontramos ante la situación donde dos módulos pueden beneficiarse mutuamente del uso del otro y las condiciones de arriba no pueden ser satisfechas. En estas situaciones, se resuelve el conflicto utilizando una técnica llamada "sandwiching". El problema es posiblemente, que alguno de los módulos contenía componentes que deben ser separados, rompiendo de esta manera el ciclo.

Notar de todas formas que existen muchos otros factores que influyen en la calidad del diseño interactuando uno con otro, y que por lo tanto optimizar una medida de calidad no asegura que la calidad general del sistema sea mejorada. Por ejemplo, poner todas las partes en un mismo módulo es una forma de asegurar una relación usa acíclica, pero esto definitivamente no constituye una mejora de diseño.

El proceso de modularización que describimos en la sección anterior, no se efectúa en un sólo paso. Dependiendo del nivel de detalle en que se maneje el diseño, los componentes de uno de los módulos resultantes de la modularización deberán ser hechos explícitos y un nuevo proceso de modularización de esta parte del diseño deberá comenzar. Esta iteración del proceso define una relación entre el módulo original y sus partes componentes a la que llamaremos es_parte_de. Por la misma definición del proceso que la genera esta relación no puede ser cíclica.

Ejemplos típicos de la relación es_parte_de se dan entre un tipo abstracto de datos y las funciones que este implementa, entre un objeto y sus métodos, entre una unit de Pascal y sus procedimientos o entre una función y las variables locales que define.

Muchas veces cuando se desarrolla software, varios individuos interactúan y cooperan hacia un objetivo en común. Estos individuos crean varios productos, algunos de los cuales son elementos estrictamente personales, tales como versiones intermedias de módulos o datos de testeo usados durante el debugging y otros son parte del producto final. A menudo el objetivo no es obtener un único producto, sino generar una colección de ellos. Podría haber miles de componentes en una estas colecciones. Aquí es donde es necesaria la introducción de la *Administración de la Configuración*, la disciplina de coordinación del desarrollo de software y el control del cambio y la evolución de productos de software y sus componentes.

Una configuración de un producto no sólo se refiere a los componentes finales de este, sino también a versiones específicas de los componentes.

La principal relación que vincula a los componentes tratados por la administración de la configuración es la relación *es_versión_de* que vincula dos módulos A y B cuando A es obtenido utilizando como base el módulo B. La relación es_versión_de entre módulos puede ser refinada en las siguientes relaciones:

A *es_revisión_de* B si A es obtenida modificando B de tal forma que el módulo obtenido sobrepase a la versión anterior. Es decir, la revisión es creada en un orden lineal. Hay muchas razones por las cuales se querría crear una revisión de un módulo, y esto puede ser reflejado especializando más todavía la relación es_revisión_de. Por ejemplo, uno querría especificar que una versión es una corrección de otra, o que ha sido obtenida mediante el reemplazo de un stub interno por un subprograma.

Otra especialización de la relación es_versión_de es la relación *es_variante_de*. Esta relación puede establecerse entre dos módulos si son indistinguibles bajo cierta abstracción (podrían ser dos implementaciones de la misma especificación para distintas plataformas). Obviamente, las variantes no sobrepasan al módulo original y por ende no están organizadas como una sucesión lineal.

Las relaciones definidas hasta el momento (usa, invoca, es_parte_de, etc.) son las más utilizadas. Cada diseño específico tendrá seguramente nuevas relaciones ad-hoc. Una lista de algunas de las posibles relaciones son flujo_de_datos; implementación; en el paradigma de objetos, herencia; en bajo nivel, operación_de_entrada_salida o interrupción; en arquitecturas client-server, pedido_de_servicio; etc.

Además, a partir de ciertas relaciones que se suponen básicas se pueden obtener relaciones inducidas, mediante operaciones matemáticas como composición, inversa o unión. Los motivos para utilizar estas relaciones derivadas son muchos: como abreviaturas de las relaciones básicas, por un cambio en el nivel de detalle del diseño, para visualizar diferentes facetas del sistema, etc.

Por ejemplo, a partir de una relación usa entre módulos y una relación es_parte_de entre módulos y módulos principales, puede definirse por composición e inversa la relación *usa_modPpal*. Decimos que un módulo principal X usa_modPpal otro módulo principal Y (distinto de sí mismo), si existe una parte A que es_parte_de X y una parte B que es_parte_de Y tal que A usa B.

La existencia de relaciones primitivas, y la capacidad de obtener a partir de ellas relaciones derivadas provocará una innovación en el enfoque modal tradicional. En una lógica modal clásica, cada operador se refiere a una determinada relación que está predefinida. Si utilizáramos esta relación para la definición de una más compleja (como hicimos en el párrafo anterior), no está claro cómo el operador modal asociado a la primera contribuye a la definición del nuevo operador modal (es decir, cual es la contrapartida sintáctica de la operación semántica de definición de nuevas relaciones). La explícita incorporación de la capacidad de definición de modalidades (mediante una estructura de Kleene sobre el operador) transforma la lógica modal clásica en lógica dinámica [Pratt, 1978; Harel, 1984], sin embargo, en nuestro trabajo este tema se abordará de una manera totalmente distinta.

Problemas Clásicos de Diseño

Además del problema de ciclicidad en una jerarquía y de la violación del concepto de information hiding, existen otros problemas de diseño que son clásicos en la literatura.

• *Excesiva distribución de información:* Un sistema puede resultar difícil de modificar si sus módulos son escritos asumiendo que una determinada característica o recurso está o no presente.

Una de las conclusiones más claras de la premisa "diseño para el cambio" es que se deben anticipar posibles cambios antes de comenzar el diseño. Esto implica una especificación detallada de cuales son los componentes y recursos centrales del diseño, cuya naturaleza no será modificada por posibles cambios futuros. Dichos componentes y recursos son los únicos que pueden ser presupuestos sin riesgo por otros módulos.

• *Cadena de componentes transformadores de datos:* Muchos programas están estructurados como una cadena de componentes, cada uno recibiendo datos del componente anterior, procesándolos y cambiándoles el formato, antes de enviar los datos al siguiente programa en la cadena. Si un componente en esta cadena no es necesario, generalmente es difícil removerlo porque la salida de su predecesor no es compatible con los requerimientos de su sucesor.

La idea fundamental para solucionar este problema es lograr la independencia entre la parte del módulo que realiza el trabajo y la parte del módulo que comunica el resultado obtenido. Así, cada módulo se compondrá de una parte de procesamiento que transforma la información y una parte de interface que comunica el resultado. De esta forma sólo se necesitarán realizar cambios en la interface de los módulos que quedan, cuando uno de los módulos es eliminado.

Las partes modificables del sistema deben estar siempre aisladas, desarrollando interfaces entre esos módulos y el resto del sistema que se mantengan válidas para todas las versiones. Los pasos cruciales para esto son los siguientes.

1. Identificar los componentes que probablemente cambiarán.
2. Localizar dichos componentes en módulos separados.
3. Diseñar interfaces con los módulos que no son modificadas, de tal forma que los aspectos que pueden modificarse no sean revelados por la interface.

Esto es otra vez la utilización del concepto de information hiding, encapsulamiento, o abstracción en otro nivel. La interface entre módulos debe ser de conocimiento general, el contenido no.

• *Componentes que realizan más de una función:* Otro error común es el combinar dos funciones diferentes en un componente porque parecen demasiado simples para ser puestas en módulos separados. A pesar de la aparente simplicidad de cada uno de los módulos, estos deben permanecer separados si desempeñan tareas conceptualmente distintas.

• *Acoplamiento:* También es un error que el diseño posea un alto acoplamiento, definido como el grado de interdependencia entre módulos. El objetivo es hacer a los módulos tan independientes como sea posible. Bajo acoplamiento entre módulos indica un sistema bien particionado. Los pasos usuales para minimizar el acoplamiento son los siguientes:

1. Eliminar relaciones innecesarias.
2. Reducir el número de relaciones necesarias.
3. Disminuir la "fuerza" de la conexión entre las de relaciones necesarias.

El bajo acoplamiento es deseable porque, cuanto menor es la cantidad de conexiones entre dos módulos, menor es la posibilidad de producirse el efecto de cascada donde un error en un módulo produce un comportamiento inesperado en otro. Un bajo acoplamiento permite modificar un módulo sin riesgo de tener que modificar otro y realizar mantenimiento en un módulo, sin necesidad de tener en cuenta los detalles internos de otro.

Existen distintos tipos (niveles) de acoplamiento. El acoplamiento entre dos módulos A y B es *normal* si A llama a B, B retorna a A, y toda la información pasada entre ellos se hace a través de parámetros presentes en la llamada. Es *global* si los dos hacen referencia a la misma área global de datos y *patológico* si uno hace referencia al código interno del otro. (Notar que este último tipo va en contra del concepto de encapsulamiento, dado que fuerza a un módulo a tener conocimiento del contenido e implementación de otro.)

• *Cohesión:* La noción dual a acoplamiento es cohesión: la fuerza de la relación entre las actividades dentro de un mismo módulo. Es habitual que un criterio de calidad de diseño influya en otro; en este caso, la cohesión de un módulo muchas veces determina el nivel de acoplamiento que tendrá éste con otros módulos del sistema. Asegurar que todos los módulos tienen alta cohesión es una forma de minimizar acoplamiento. Uno de los ejemplos más usuales de baja cohesión es la violación de esta premisa de "Componentes que realizan más de una función" comentada más arriba.

2.2 Estado del Arte en Verificación de Propiedades

Los entornos de Ingeniería de Software deben proveer de herramientas que permitan al ingeniero de software comprender mejor el sistema que esta diseñando. A pesar de la existencia de múltiples herramientas que ayudan al seguimiento y la verificación del proceso de desarrollo de software, han sido pocas las que hemos encontrado que específicamente permitan la verificación formal del sistema en la etapa de diseño.

En su mayoría, las aplicaciones que permiten especificar el diseño de un sistema son herramientas CASE que utilizan métodos y técnicas del Diseño Estructurado, como los Gráficos de Estructuras, Diagramas de Entidad-Relación, etc. El problema con estas herramientas es que en general sólo permiten una descripción semi-formal del sistema y no soportan la verificación de propiedades sobre el diseño. Esto se debe a que muchas veces la verificación formal es dejada para etapas posteriores del desarrollo. Pero, de la misma forma que una especificación formal del problema evita la ambigüedad y ayuda a una mejor comprensión de los requerimientos de un sistema, la posibilidad de descripción y verificación formal en la etapa de diseño permite llegar a una implementación del sistema con una visión más clara de los componentes que lo forman. La detección temprana de errores disminuye el costo final del sistema, y en particular, los errores de diseño son los que mayor impacto poseen pues producen una cascada de modificaciones en todas las etapas posteriores del desarrollo.

Algunos de los trabajos que se han realizado (GraphLog [Cosens et al., 1992], GraSp [Agustí et al., 1995]) para tratar con este problema utilizan lenguajes gráficos como herramienta principal.

Cosens et al. proponen una técnica de visualización gráfica que brinda ayuda en la interpretación y corrección de una cierta clase de problemas comunes en el área de Ingeniería de Software.

Si consideramos a un sistema de software como un gran conjunto de hechos interrelacionados, podemos pensar en almacenar esta información en forma de base de datos. Posteriormente, un lenguaje de consultas permitirá al ingeniero extraer de la base subconjuntos de datos de especial interés, correlacionar información dentro de la base,

proveer distintas visiones de los datos y también obtener información derivada mediante nuevas definiciones de hechos y relaciones inferidas.

Sin embargo la tecnología convencional utilizada en bases de datos no es totalmente apropiada para los requerimientos de la Ingeniería de Software. En particular, los lenguajes de consulta basados en lógica de primer orden como SQL no tienen el poder expresivo necesario para capturar conceptos habituales en el análisis de diseños como pueden ser la clausura transitiva de una relación, o distintas propiedades de caminos en un grafo.

La propuesta de los autores se plantea en la definición de un lenguaje de consultas denominado GraphLog. La idea es considerar al sistema de software como un grafo dirigido. Las consultas se construyen mediante la definición de un grafo que actúa de patrón. El proceso de evaluación de la consulta consiste en encontrar en la base de datos todas las instancias del patrón definido y sobre cada una de estas instancias ejecutar una acción determinada que puede ser, por ejemplo, la definición de una nueva relación en la base o la extracción de la instancia del patrón hallada. GraphLog trabaja sobre una base de datos escrita en PROLOG.

La principal ventaja que provee GraphLog es que reduce el gap que existe entre el lenguaje en que usualmente se formulan las consultas (lenguaje simbólico) y el que se utiliza para mostrar el resultado (lenguaje gráfico). Pero además, GraphLog posee un mayor poder expresivo que SQL ya que permite, entre otras cosas, la definición de clausuras transitivas sin utilizar recursión.

En GraphLog, las entidades se representan mediante cajas, y las relaciones como arcos dirigidos entre las cajas. Las consultas en GraphLog son gráficos cuyos nodos están etiquetados por una secuencia de variables y constantes, y cuyos arcos están etiquetados por expresiones regulares compuestas por nombres de predicados.

Otra herramienta que utiliza un lenguaje gráfico para la descripción de sistemas es GraSp (A Graphical Specification Language for the Preliminary Specification of Logic Programs). GraSp es presentado como un lenguaje gráfico de especificación de alto nivel, que utiliza la teoría de conjuntos como base. En él, las diferentes entidades del sistema pueden ser representados como conjuntos o funciones entre conjuntos.

GraSp también permite la traducción automática de los diagramas a cláusulas de Horn, y se está desarrollando un sistema automático de deducción (basado en los modelos de teorías de inclusión) para ayudar a la construcción y verificación de dichos diagramas.

Un enfoque similar al de Cosens et al. pero que no utiliza un lenguaje gráfico para el diseño de las consultas es el trabajo de [Paul & Prakash, 1996]. En este trabajo también se considera el modelo del programa como un conjunto de hechos, en este caso representados mediante una base de datos de *objetos*. El programa es visto como una red o un grafo donde los nodos son objetos fuertemente tipados y los ejes son las relaciones entre los objetos. La principal diferencia en el enfoque de Paul y Prakash es que su modelo esta muy orientado a la reingeniería de programas (la reconstrucción del diseño de un sistema a partir de su código), lo que motiva su cambio de un lenguaje gráfico para la formulación de consultas (cercano al modelo de diseño) por un lenguaje algebraico (cercano al modelo de implementación). El álgebra de consultas SCA propuesto en el trabajo es una extensión del álgebra de relaciones usual en base de datos. Esta extensión también esta motivada por una necesidad de un mayor poder expresivo en el lenguaje (operadores de clausura, aplicación y cuantificación sobre objetos).

Se contraponen a los enfoques anteriores, metodologías más tradicionales como las propuestas en [Maibaum et al., 1984] en donde a partir de una especificación de alto nivel utilizando TADs se propone un esquema de composición y refinamiento articulado como una teoría de implementación de módulos basada en una lógica de primer orden infinitaria. Esta teoría ecuacional define perfectamente la noción de implementación de un TAD en base a TADs de menor complejidad, permitiendo la demostración de la correctitud de dicha implementación por conmutación de diagramas de interpretación. Este enfoque tiene una visión totalmente constructiva y descuida en parte la importancia de las distintas relaciones que pueden establecerse entre los elementos de un diseño.

El formalismo propuesto en esta tesis está más próximo al espíritu del enfoque que utiliza lenguajes gráficos. Dado que nuestros métodos fueron pensados para aplicarse al análisis y verificación del diseño de un sistema, tomaremos el grafo de diseño como instrumento básico. Proponemos sin embargo la utilización de un lenguaje modal lógico que funcione como la contrapartida simbólica formal de los gráficos de consulta propuestos por

[Cosens et al., 1992]. La elección de un lenguaje lógico está motivada por la intensión de testear propiedades sobre el diseño, es decir, verificar formalmente las características que intuitivamente se asignan a un diseño. La forma más natural de llevar a cabo esta verificación, es proponer un lenguaje para expresar dichas propiedades y definir un método que permita determinar si son o no validadas por el diseño. Es totalmente distinto el caso al que se enfrentan [Paul & Prakash, 1996] en donde se debe analizar el código de un programa desconocido. En este caso, es más útil un lenguaje de consulta que *extraiga* información del objeto (como el álgebra relacional propuesta en dicho artículo), puesto que no poseemos presuposiciones a *testear*.

Aunque el prototipo presentado en el Apéndice de esta tesis, sólo implementa una interface sintáctica, no vemos como impensable una herramienta que utilice una interface gráfica para el diseño de la consulta (un lenguaje más simple que el de las fórmulas modales) que será traducida a una propiedad modal a chequear sobre el modelo correspondiente al sistema. Creemos que esta traducción puede ser en algunos aspectos más apropiada que la traducción a consulta PROLOG, al permitir la manipulación explícita de las relaciones del diseño vía los operadores modales.

3 Lógica Modal

3.1 Conceptos Básicos

Daremos a continuación las nociones básicas de Lógica Proposicional y Lógica Modal que se utilizarán en el trabajo. Las referencias clásicas son [Chellas, 1980; Hughes & Cresswell, 1984]. Aquellos que estén familiarizados con el tema pueden saltear las siguientes secciones y remitirse a "3.2 Extensiones: Lógica Polimodal con Operadores Inversos", donde se realizan las extensiones necesarias para adaptar el formalismo clásico al nuevo enfoque.

El Cálculo Proposicional

El Cálculo Proposicional Clásico (CP) se construye a partir del siguiente conjunto de símbolos primitivos o no definidos que constituyen su *alfabeto básico* (usualmente notado como **A**):

1. Un conjunto infinito (numerable) de *símbolos de proposición*, que notaremos como p, q, r, ... o con subíndices p_1, p_2, p_3, ... Representaremos a este conjunto mediante el símbolo $\mathbb{P}$.
2. Los *símbolos* u *operadores básicos* $\top$, $\neg$, $\vee$, (y).

Ciertas secuencias de símbolos constituyen lo que se denominan las *fórmulas bien formadas* (fbf) del Cálculo Proposicional. Las reglas de formación son las siguientes:

BF1. $\top$ es una fbf.
BF2. Todo símbolo de proposición es una fbf.
BF3. Si φ es una fbf, entonces $\neg\varphi$ es una fbf.
BF4. Si φ y ψ son fbf, entonces $(\varphi \vee \psi)$ es una fbf.

Notaremos con las letras φ, ψ, ... fórmulas cualesquiera y con Γ, Σ, ... conjuntos de fórmulas cualesquiera. Cuando sea posible evitaremos el uso de paréntesis según es habitual. Representaremos al conjunto de todas las fórmulas bien formadas (lenguaje) sobre el alfabeto **A** mediante el símbolo $\mathcal{L}_{\mathbf{A}}$.

Los símbolos de proposición son variables que toman proposiciones como valores. Cada proposición es verdadera o falsa, pero no ambos. Usaremos los valores '1' para el valor verdadero y '0' para el valor falso.

Interpretaremos $\neg$ y $\vee$ como sigue: Para toda fórmula φ, si φ es verdadera entonces $\neg\varphi$ es falsa, y si φ es falsa entonces $\neg\varphi$ es verdadera; y sea φ y ψ fórmulas cualquiera, $(\varphi \vee \psi)$ es verdadera si al menos una de φ y ψ es verdadera, y falsa si ambas φ y ψ son falsas.

Estos conceptos pueden expresarse más correctamente utilizando la noción de *valuación* o *asignación*. Decimos que $V: \mathcal{L}_{\mathbf{A}} \rightarrow \{0, 1\}$ es una valuación del Cálculo Proposicional si y sólo si satisface las siguientes condiciones:

V1. $V(\top) = 1$.
V2. Para todo símbolo de proposición p, $V(p) = 0$ o $V(p) = 1$, pero no ambos.
V3. Para toda fbf φ, $V(\neg\varphi) = 1$ si $V(\varphi) = 0$, en otro caso $V(\neg\varphi) = 0$.
V4. Para todo par de fbfs φ y ψ, $V(\varphi \vee \psi) = 1$ si $V(\varphi) = 1$ o $V(\psi) = 1$, en otro caso $V(\varphi \vee \psi) = 0$.

Una fbf φ que recibe el valor 1 bajo toda posible valuación V es llamada *CP-válida* o *tautología del CP*. El símbolo $\top$ es también utilizado para representar una tautología cualquiera. Esto es posible debido a que en toda fórmula cuyo significado sea función de verdad (como cualquier fórmula de CP) una tautología puede ser intercambiada por otra sin que el valor de verdad de la fórmula sea modificado. La noción opuesta a CP-tautología es la de *CP-contradicción*, una fórmula que recibe el valor 0 bajo toda posible valuación V. El símbolo $\bot$ se utiliza para representar una contradicción cualquiera.

Notaremos como *Tau* al conjunto de todas las tautologías y *Con* al conjunto de todas las contradicciones.

$\neg$ y $\vee$ se denominan operadores básicos pues a partir de ellos pueden definirse *operadores derivados*. Los más usuales son los siguientes:

Def $\bot$. $\bot =_{def} \neg\top$.

Def $\wedge$. $(\varphi \wedge \psi) =_{def} \neg(\neg\varphi \vee \neg\psi)$.

Def $\rightarrow$. $(\varphi \rightarrow \psi) =_{def} (\neg\varphi \vee \psi)$.

Def $\leftrightarrow$. $(\varphi \leftrightarrow \psi) =_{def} ((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi))$.

Las interpretaciones de $\bot$, $\wedge$, $\rightarrow$, $\leftrightarrow$ están determinadas por su definición, más la definición de las interpretaciones de los operadores básicos.

Alfred Tarski [Tarski, 1956] introdujo la noción de operador de consecuencia. Un *operador de consecuencia* es una función Cn definida entre subconjuntos de fórmulas (Cn: $\mathcal{P}(\mathcal{L}_A) \rightarrow \mathcal{P}(\mathcal{L}_A)$) que posee las siguientes propiedades, para Γ y Σ conjuntos de fórmulas cualesquiera:

Cn1. $\Gamma \subseteq Cn(\Gamma)$. (Inclusión)

Cn2. $Cn(\Gamma) = Cn(Cn(\Gamma))$. (Idempotencia)

Cn3. $\Gamma \subseteq \Sigma$ implica $Cn(\Gamma) \subseteq Cn(\Sigma)$. (Monotonía)

Existen muchas funciones que satisfacen estas propiedades, por ejemplo, la función identidad en $\mathcal{L}_A$.

El significado esperado de la función Cn es que dado un conjunto de sentencias Γ nos retorne el conjunto de las sentencias que son consecuencias lógicas de Γ. Es decir que $\varphi \in Cn(\Gamma)$ si y sólo si φ se sigue lógicamente de las sentencias de Γ. Es por eso usual agregar a las condiciones mínimas impuestas por Tarski las siguientes:

Cn4. Si φ puede ser obtenido a partir de Γ por tablas de verdad, entonces $\varphi \in Cn(\Gamma)$. (Supraclasicalidad)

Cn5. Si $\varphi \rightarrow \psi \in Cn(\Gamma)$ entonces $\psi \in Cn(\Gamma \cup \{\varphi\})$. (Deducción)

Cn6. Si $\varphi \in Cn(\Gamma)$ entonces existe Γ' finito, $\Gamma' \subseteq \Gamma$ tal que $\varphi \in Cn(\Gamma')$. (Compacidad)

Aún con estos nuevos postulados, la función Cn no queda determinada unívocamente, pero ahora algunas propiedades interesantes pueden derivarse, como por ejemplo que si φ es una CP-tautología entonces $\varphi \in Cn(\Gamma)$ para todo Γ. En particular, si Cn es la relación de consecuencia mínima que satisface los postulados antes mencionados, *Tau* = $Cn(\varnothing)$.

El Cálculo Proposicional Modal

Todos los operadores que definimos hasta ahora (básicos y derivados) comparten la característica de ser *veritativos funcionales*. Esto significa que cuando construimos una proposición utilizando proposiciones más simples y alguno de estos operadores, el valor de verdad de la nueva proposición queda unívocamente determinado por el valor de verdad de las proposiciones que utilizamos en la construcción.

El Cálculo Modal Clásico (CM) se obtiene como una extensión del Cálculo Proposicional Clásico, introduciendo el operador [] y obteniendo de esta manera el alfabeto modal **AM** = $\{(,), \neg, \vee, []\} \cup \{p_i \mid i \in \mathbb{N}\}$. La principal característica del operador [] que lo hace diferente de los operadores anteriores es que no es veritativo funcional. Veremos con detalle esta característica más adelante.

Dado que hemos extendido nuestro alfabeto agregando más operadores primitivos, debemos también realizar extensiones a las definiciones que hemos dado hasta el momento.

El conjunto $\mathcal{L}_{AM}$ de las fórmulas bien formadas sobre **AM** se define mediante las reglas BF1 a BF4 más:

BF5. Si φ es una fbf, entonces $[]\varphi$ es una fbf.

Definimos también un nuevo operador derivado <> como:

Def <>. $<>\varphi =_{def} \neg[]\neg\varphi$.

Los nuevos operadores [] y <> pueden ser interpretados de distintas formas. Clásicamente, [] es el operador de necesidad y <> su dual, posibilidad; sin embargo, sus características exactas pueden variar dependiendo de diferentes parámetros.

Sistemas Modales Normales

Un *sistema de lógica modal* o más simplemente una *lógica modal* no es más que un subconjunto del lenguaje $\mathcal{L}_{AM}$. Si S es una lógica modal, los elementos de S son *teoremas* de la lógica y notaremos habitualmente $\vdash_S \varphi$ si $\varphi \in S$. La lógica modal que comprende todo el lenguaje $\mathcal{L}_{AM}$ es el único *sistema inconsistente*, todos los otros sistemas son *consistentes*.

Dada una fórmula φ cualquiera en $\mathcal{L}_{AM}$, una *instancia de sustitución* de φ es la fórmula que se obtiene al reemplazar toda aparición de un símbolo de proposición p_i en φ, uniformemente, por otra fórmula φ' de $\mathcal{L}_{AM}$.

Un sistema de lógica modal S es *normal* si cumple con las siguientes reglas:

1. Si φ es una CP-tautología o una instancia de sustitución de CP-tautología entonces φ está en S.
2. S contiene todas las instancias de sustitución de la fórmula **K**. $[](p \rightarrow q) \rightarrow ([]p \rightarrow []q)$.
3. S está clausurada por las siguientes *reglas de derivación*

MP. Si φ está en S y $\varphi \rightarrow \psi$ está en S, entonces ψ está en S. (Modus Ponens)

N. Si φ está en S entonces $[]\varphi$ está en S. (Necesitación)

Es habitual definir sistemas normales especificando efectivamente un conjunto de fórmulas (usualmente finito) que se denominan *axiomas esquemas* y utilizar el nombre de dichos esquemas o una referencia a cierta característica destacada para nombrar a la lógica. Por ejemplo, la lógica normal mínima que dimos recién se llama lógica **K**. Los axiomas esquemas se utilizan como plantillas, de forma tal que cualquier instancia de sustitución de una de estas fórmulas es considerada como perteneciente al sistema, es decir que dar un axioma esquema es equivalente a dar el conjunto de fórmulas que se obtienen por sustitución a partir de él. El sistema lógico será entonces, el conjunto que fórmulas que se obtienen de los axiomas esquemas por sustitución y clausura por **MP** y **N**, además de las instancias de sustitución de CP-tautologías. Notar que para que el sistema sea normal, debe poder inferirse toda instancia de **K** a partir de estas reglas.

Cuando se presenta un sistema en esta forma, se dice que se ha dado una *base axiomática* para el sistema, o que el sistema ha sido *axiomatizado*. Si el conjunto de axiomas esquema que utilizamos para especificar el sistema es finito, entonces tenemos una *base axiomática finita*.

Una *demostración* de una fórmula φ en un sistema axiomático S, es una secuencia finita de fórmulas $\varphi_1, \ldots, \varphi_n$ tal que $\varphi_n = \varphi$ y donde cada φ_i es o bien una instancia de un axioma esquema de S, o bien el resultado de la aplicación de una instancia de una regla de S con premisas en $\varphi_1, \ldots, \varphi_{i-1}$.

Modelos

Un *modelo modal* o *modelo de Kripke* (por Samuel A. Kripke quien en 1963 introdujo esta noción para dar semántica a lógicas modales [Kripke, 1963a; Kripke, 1963b]) es una estructura matemática que permite interpretar las fórmulas de CM de la misma forma que las valuaciones permiten interpretar las fórmulas de CP.

Un modelo de Kripke se define como una upla $M = \langle W, \longrightarrow, V \rangle$, donde:

W es un conjunto no vacío, los elementos de W se llamarán *mundos*, o *mundos posibles*;

$\longrightarrow$ es una relación diádica sobre W (i.e. $\longrightarrow \subseteq W \times W$), usualmente denominada *relación de accesibilidad*. Si $m \longrightarrow m'$ decimos que m' *es accesible desde* m o que m *puede ver a* m';

V es una función total del conjunto de pares de símbolos de proposición y mundos en los valores de verdad '0' y '1' (i.e. $V: \mathbb{P} \times W \rightarrow \{0, 1\}$). Esta función es una *valuación modal* que indica si una variable proposicional es verdadera o no en un determinado mundo.

Decimos que un modelo *M* es un *modelo finito*, si su conjunto de mundos tiene sólo un número finito de elementos (notar que entonces la relación de accesibilidad también es finita); de otra forma el modelo es infinito.

Escribiremos $M \models_m \varphi$ para decir que la fórmula φ es *válida en un mundo* posible m en el modelo *M*.

Dado un modelo $M = \langle W, \longrightarrow, V \rangle$ la noción de validez en mundos posibles se define recursivamente como:

1. $M \models_m \top$.
2. $M \models_m p_i$ sii $V(p_i, m) = 1$, con p_i un símbolo proposicional.
3. $M \models_m \neg\varphi$ sii no es el caso que $M \models_m \varphi$.
4. $M \models_m (\varphi \vee \psi)$ sii $M \models_m \varphi$ o $M \models_m \psi$.
5. $M \models_m []\varphi$ sii para todo mundo m' en W, $m \longrightarrow m'$ implica $M \models_{m'} \varphi$.

Con esta definición podemos ahora comentar por qué [] no es un operador veritativo funcional. Supongamos que estamos evaluando el valor de []p en un mundo m_1, ahora bien, el valor de []p estará determinado por el valor de p en los mundos accesibles desde m_1 que pueden dar a p un valor verdadero o falso independientemente del valor que m_1 le asigne a esa variable proposicional o sea que, si el mismo m_1 no es accesible desde sí mismo, el valor de p en m_1 podría ser modificado sin alterar el valor de []p. Para comprender mejor este ejemplo, pensemos que []p representa Creo_que p. El valor de verdad de p puede no influir en el valor de verdad de Creo_que p. Bien podría ser que se hayan descubierto recientemente hechos que implican la falsedad de p y que por mala información o simplemente por desconfianza de la fuente yo siguiera sosteniendo Creo_que p.

A partir de la relación de validez para mundos pueden derivarse las siguientes nociones:

Decimos que una fórmula φ es *válida en un modelo* $M = \langle W, \longrightarrow, V \rangle$ (Not.: $M \models \varphi$) sii para todo mundo $m \in W$, $M \models_m \varphi$.

Decimos que una fórmula φ es *válida en una clase de modelos* C (Not.: $\models_C \varphi$) sii para todo modelo $M \in C$, $M \models \varphi$.

Correctitud, Completitud y Decidibilidad

En las secciones anteriores definimos los conceptos de teorema de un sistema axiomático ($\vdash$) y de fórmula válida ($\models$). Estas nociones no son independientes.

Dado un sistema axiomático S y una clase de modelos C, decimos que:

S es un *sistema correcto* para C sii para toda fórmula φ, $\vdash_S \varphi$ implica $\models_C \varphi$, es decir todo lo demostrable en el sistema es válido en la clase.

S es un *sistema completo* para C sii para toda fórmula φ, $\models_C \varphi$ implica $\vdash_S \varphi$, es decir todo lo válido en la clase es demostrable en el sistema.

(Existen en la literatura otras definiciones de correctitud y completitud, pero las dadas más arriba son las más habituales. En algunos casos es particularmente importante dejar en claro que definición de completitud se está manejando).

Cuando puede demostrarse para una cierta clase de modelos C que un sistema axiomático S es tanto correcto como completo, se está mostrando que existe una perfecta concordancia entre sintaxis y semántica. Una lógica que es a la vez completa y correcta se denomina *adecuada*.

Las demostraciones de correctitud son habitualmente simples. Cuando el sistema S viene presentado como un conjunto de esquemas axiomas y de reglas de inferencia, basta mostrar que toda instancia de axioma es correcta (mostrando que cualquier mundo en cualquier modelo la valida) y que las reglas de inferencias preservan correctitud (llevándonos de premisas válidas a conclusiones válidas).

Las demostraciones de completitud en cambio, son en general mucho más difíciles, pues no tenemos a priori una especificación efectiva del conjunto de las fórmulas válidas en todos los modelos de la clase. Sin embargo, el *método del modelo canónico* puede usualmente ser utilizado para obtener resultados de completitud. (Este método deriva del trabajo de [Lemmon & Scott, 1977]).

Supongamos que S es un sistema modal normal, y que tenemos una clase de modelos C respecto de la cual queremos demostrar completitud. Llamaremos a los modelos en C, *C-modelos* y a una fórmula *C-válida* si es válida en todos los mundos de todos los modelos de C. Entonces, decir que S es completo respecto de C es decir que toda

fórmula C-válida es un teorema de S, o equivalentemente, que si una fórmula no es un teorema de S entonces no es C-válida (por contrarrecíproco), o más formalmente si y sólo si:

• Para toda fórmula φ, si $\nvdash_S \varphi$ entonces existe un C-modelo $M = \langle W, \longrightarrow, V\rangle$ y un mundo m en W tal que $M \nvDash_m \varphi$.

Por otro lado, digamos que una fórmula φ es *S-inconsistente* si su negación es un teorema ($\vdash_S \neg\varphi$), y *S-consistente* en otro caso ($\nvdash_S \neg\varphi$). Usando estas nuevas nociones y la definición de validez dada anteriormente es fácil demostrar que la proposición anterior es equivalente a:

• Para toda fórmula φ, si φ es *S-consistente* entonces existe un C-modelo $M = \langle W, \longrightarrow, V\rangle$ y un mundo m en W tal que $M \vDash_m \varphi$.

El método del modelo canónico implica la construcción de un modelo *M* que valida en algún mundo toda fórmula S-consistente, y la demostración de que *M* pertenece a la clase C de modelos, probando de esta manera que S es completo para C.

Nos queda por analizar un último tema en relación a la lógica modal clásica, quizás el de mayor interés desde un punto de vista computacional.

En el área de Computabilidad es conocido el resultado que establece que algunos conjuntos no son *decidibles* o lo que es lo mismo, que existen conjuntos cuya función característica no es *recursiva*. Estos conjuntos poseen la propiedad que, a pesar de estar bien definidos, la cuestión de si un elemento φ pertenece o no al conjunto no puede decidirse efectivamente.

Como ya dijimos anteriormente que una lógica no es más que un conjunto de fórmulas, podría ser que fuera justamente un conjunto no decidible y que, por lo tanto, la cuestión de decidir si $\varphi \in S$ ($\vdash_S \varphi$) fuera no computable.

Puesto que trabajaremos con lógicas que son presentadas como sistemas axiomáticos, la "mitad" del problema está solucionada. Existe un método para determinar que una fórmula φ sí pertenece al sistema S: basta comenzar a generar sistemáticamente todas las posibles demostraciones realizables en S. Es cierto que el número de demostraciones posibles es infinito, pero si φ es un teorema de la lógica en algún momento aparecerá al final de alguna de las demostraciones y el proceso podrá acabar. El problema es que si está abierta la pregunta de si φ es o no un teorema, mientras φ no aparezca al final de una demostración, no sabremos si φ **todavía** no apareció a pesar de ser teorema, o si **nunca** aparecerá puesto que no es teorema.

Lo que necesitamos es un método efectivo para determinar que una fórmula no es teorema.

Una forma de proveer este método es demostrar que una lógica modal posee la *propiedad de modelo finito*, que se demuestra utilizando la técnica de filtración. Informalmente, un modelo M^* es el producto de una Γ-filtración de otro modelo *M*, si en algún sentido, las fórmulas en Γ son "preservadas" en M^*, o en otras palabras, M^* es una simplificación de *M* que sin embargo respeta el valor de verdad de las fórmulas en Γ. La filtración se utilizará para hallar un modelo finito equivalente a *M* respecto de Γ.

Dado un sistema axiomático S y una clase de modelos C, decimos que S tiene la propiedad de modelo finito para una clase C sii toda fórmula S-consistente es válida en un mundo en un modelo *M* de C, y *M* es finito.

La demostración de la propiedad anterior no es otra cosa que la demostración de que S es completa respecto de la subclase de los modelos finitos de C.

Ahora bien, si tenemos una fórmula φ que no pertenece a un sistema S correcto y completo para una clase C, es porque existe un modelo $M = \langle W, \longrightarrow, V\rangle$ en C que es *contraejemplo* para φ, es decir que existe $m \in W$ tal que $M \nvDash_m \varphi$ sii por definición de valuación $M \vDash_m \neg\varphi$.

Es decir que para testear que $\nvdash_S \varphi$ hay que mostrar un modelo contraejemplo, pero si φ no está en S, $\neg\varphi$ es S-consistente y si S tiene la propiedad del modelo finito el modelo buscado es finito y todos los modelos finitos (de cualquier tamaño) pueden ser sistemáticamente explorados. Otra vez, si no sabemos si φ está o no en S, este proceso puede no detenerse nunca, pues no sabremos si todavía no encontramos el modelo contraejemplo buscado o si nunca lo encontraremos. Pero ahora, teniendo procesos de decisión tanto para pertenencia al sistema como para la no pertenencia, podemos armar el proceso de decisión para S, basta "correr ambos métodos en paralelo" hasta que alguno de ellos responda con éxito.

3.2 Extensiones: Lógica Polimodal con Operadores Inversos

Hasta aquí, hemos trabajado con lo que se denomina lógica modal clásica pues las únicas modalidades permitidas en los sistemas eran [] y <>. Para nuestro trabajo sin embargo, los recursos que ofrece esta lógica no son suficientes y deberemos utilizar *extensiones* a la lógica modal clásica.

La primera extensión que realizaremos está motivada por el tipo de modelos en los que basaremos nuestro trabajo. Los modelos que utilizaremos no tienen una sino muchas relaciones de accesibilidad entre mundos. Esto motiva que existan diferentes modalidades [a], vinculada cada una a las diferentes relaciones $-a\rightarrow$ del modelo.

Nuestros modelos serán entonces uplas $M = <W, \{-a\rightarrow \mid a \in L\}, V>$, donde la relación $\longrightarrow$ fue reemplazada por un conjunto de relaciones. L es llamado el *sort* de la lógica y es un conjunto no vacío de etiquetas o labels que nos permiten identificar a las distintas relaciones de accesibilidad. En nuestro caso además L será siempre finito.

Puede verse fácilmente que esta extensión agrega poder expresivo al lenguaje. Todos los modelos anteriores están admitidos en el formalismo (basta usar una única relación de accesibilidad) y además toda fórmula que incorpore modalidades de distintos sorts no podrá ser equivalente a ninguna fórmula en el formalismo anterior.

Denominaremos a la mínima lógica normal extendida por múltiples modalidades **KP** (**K** polimodal).

Las lógicas polimodales nacieron en el marco de las lógicas de conocimiento y creencia. Un ejemplo clásico puede encontrarse en [Hintikka, 1962] donde $[a]\varphi$ se interpreta como a *sabe o cree* φ, o semánticamente, en un sistema de transición que representa creencia o conocimiento $s -a\rightarrow t$ simboliza el hecho que a en el estado s sabe o cree las cosas en t. Las lógicas polimodales pueden ser aplicadas naturalmente a la comprensión de actividad y cambio.

La otra extensión es también tradicional y está relacionada con el área de lógica temporal [Burgess, 1984]. En esta área la relación de accesibilidad $\longrightarrow$ es interpretada como la relación temporal que ordena los instantes de tiempo: $t_1 \longrightarrow t_2$ sii t_1 es un instante anterior (temporalmente) a t_2.

Si $\longrightarrow$ es interpretada de esta forma, el operador <> se torna un operador de *futuro*: $<>\varphi$ es cierta en un instante t_1 sii existe un instante t_2 posterior a t_1 en donde φ es cierta. Es fácil verificar que el operador dual [] no es el operador de pasado, como podría esperarse. [] representa *siempre en el futuro*: $[]\varphi$ sii $\neg<>\neg\varphi$ sii no existe momento posterior en donde vale $\neg\varphi$ o, en otras palabras, todo momento posterior (si es que hay alguno) es un momento en donde φ es cierta.

Bajo esta interpretación y sin modificaciones, la lógica modal clásica constituye una *lógica temporal de futuro puro*, las expresiones acerca del pasado no pueden ser expresadas en el lenguaje.

La extensión en este caso no se consigue mediante la redefinición del concepto de modelo sino mediante el agregado de las modalidades inversas $<>_i$ y su dual $[]_i$.

$M \models_m <>_i\varphi$ sii existe un mundo m' en W tal que $m' \longrightarrow m$ y $M \models_{m'} \varphi$ (notar que las posiciones de m' y m están intercambiadas respecto de la definición de $M \models_m <>\varphi$).

De esta forma $<>_i\varphi$ es cierta en un instante t_1 sii existe un instante t_2 anterior a t_1 en donde φ es cierta.

En nuestro trabajo, no usaremos los operadores inversos como operadores temporales sino por el mayor poder expresivo que brindan a la lógica. Es por eso que no siempre haremos las suposiciones de transitividad y antisimetría sobre la relación de accesibilidad que se hacen habitualmente en la interpretación temporal.

Notar que en este caso, los operadores <> y $<>_i$ están íntimamente relacionados (a diferencia de <a> y <b> con a, $b \in L$, en la extensión anterior). En particular, verifican la propiedad que dado que φ es cierta ahora, será siempre cierto que φ ha sucedido (y simétricamente hacia el pasado, ha sido siempre el caso que φ sucederá). Esta relación se verá reflejada en la axiomatización del sistema.

Denominaremos a la mínima lógica normal extendida por modalidades inversas **KI** (**K** Inversa).

Presentamos a continuación entonces la lógica polimodal inversa **KPI** que incorpora ambas extensiones.

Definición: Dado un sort L de labels (siempre finito), el alfabeto de la lógica $\mathbf{KPI}_L$ se define como $\mathbf{API}_L = \{\vee, \neg, (,), \top\} \cup \{[a] \mid a \in L\} \cup \{[a]_i \mid a \in L\} \cup \{p_i \mid i \in \mathbb{N}\}$. La restricción a lenguaje finito de n símbolos de proposición ($n \in \mathbb{N}$) es el alfabeto $\mathbf{API}_L^n = \{\vee, \neg, (,), \top\} \cup \{[a] \mid a \in L\} \cup \{[a]_i \mid a \in L\} \cup \{p_i \mid i < n\}$.

Definición: Dado un sort L de labels, las fórmulas bien formadas de la lógica $\mathbf{KPI}_L$ se definen por recursión como:

BF1. $\top$ es una fbf.
BF2. Todo símbolo de proposición es una fbf.
BF3. Si φ es una fbf, entonces $\neg\varphi$ es una fbf.
BF4. Si φ y ψ son fbf, entonces $(\varphi \vee \psi)$ es una fbf.
BF5. Si φ es una fbf y $a \in L$, entonces $[a]\varphi$ es una fbf.
BF6. Si φ es una fbf y $a \in L$, entonces $[a]_i\varphi$ es una fbf.

Definición: Dada una fórmula φ en $\mathbf{KPI}_L$ decimos que ψ es subfórmula de φ si ψ pertenece al conjunto $Sub(\varphi)$ definido recursivamente como:

1. Para $\varphi = \top$, $Sub(\varphi) = \{\top\}$.
2. Para $\varphi = p_i$, $Sub(\varphi) = \{p_i\}$.
3. Para $\varphi = \neg\psi$, $Sub(\varphi) = \{\neg\psi\} \cup Sub(\psi)$.
4. Para $\varphi = (\psi \vee \theta)$, $Sub(\varphi) = \{(\psi \vee \theta)\} \cup Sub(\psi) \cup Sub(\theta)$.
5. Para $\varphi = [a]\psi$, $Sub(\varphi) = \{[a]\psi\} \cup Sub(\psi)$.
6. Para $\varphi = [a]_i\psi$, $Sub(\varphi) = \{[a]_i\psi\} \cup Sub(\psi)$.

Decimos que una fórmula tiene tamaño n sii $n = \#Sub(\varphi)$.

Definición: En $\mathbf{KPI}_L$ utilizaremos los siguientes operadores derivados:

Def $\bot$. $\bot =_{def} \neg\top$.
Def $\wedge$. $(\varphi \wedge \psi) =_{def} \neg(\neg\varphi \vee \neg\psi)$.
Def $\rightarrow$. $(\varphi \rightarrow \psi) =_{def} (\neg\varphi \vee \psi)$.
Def $\leftrightarrow$. $(\varphi \leftrightarrow \psi) =_{def} ((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi))$.
Def <a>. Sea a cualquiera en L, $\langle a\rangle\varphi =_{def} \neg[a]\neg\varphi$.
Def $\langle a\rangle_i$. Sea a cualquiera en L, $\langle a\rangle_i\varphi =_{def} \neg[a]_i\neg\varphi$.

Definición: Un sistema de lógica polimodal con operadores inversos S de sort L es normal si puede ser especificado mediante las siguientes reglas:

1. Si φ es CP-tautología o una instancia de sustitución de una CP-tautología entonces φ está en S.
2. S contiene todas las instancias de sustitución de las fórmulas
   Ka. $[a](\varphi \rightarrow \psi) \rightarrow ([a]\varphi \rightarrow [a]\psi)$, para toda $a \in L$.
   Ka$_i$. $[a]_i(\varphi \rightarrow \psi) \rightarrow ([a]_i\varphi \rightarrow [a]_i\psi)$, para toda $a \in L$.
   It1. $\varphi \rightarrow [a]\langle a\rangle_i\psi$, para toda $a \in L$.
   It2. $\varphi \rightarrow [a]_i\langle a\rangle\psi$, para toda $a \in L$.
3. S está clausurada por las siguientes reglas de derivación
   MP. Si φ está en S y $\varphi \rightarrow \psi$ está en S, entonces ψ está en S. (Modus Ponens)
   Na. Si φ está en S entonces $[a]\varphi$ está en S, para toda $a \in L$. (Necesitación)
   Na$_i$. Si φ está en S entonces $[a]_i\varphi$ está en S, para toda $a \in L$. (Nec. Inversa)

Definición: Un modelo de Kripke para $\mathbf{KPI}_L$ es una upla $M = \langle W, \{-a\rightarrow \mid a \in L\}, V\rangle$ donde:

1. W es un conjunto no vacío.
2. L es un conjunto (finito).
3. Cada $-a\rightarrow$, con $a \in L$ es una relación en $W \times W$.
4. V es una valuación modal ($V: \mathbb{P} \times W \rightarrow \{0, 1\}$).

Definición: Dado un modelo $M = \langle W, \{-a\rightarrow \mid a \in L\}, V\rangle$ la noción de validez en mundos posibles se define recursivamente como:

1. $M \models_m \top$.
2. $M \models_m p_i$ sii $V(p_i, m) = 1$, con p_i un símbolo proposicional.
4. $M \models_m \neg\varphi$ sii no es el caso que $M \models_m \varphi$.
4. $M \models_m (\varphi \vee \psi)$ sii $M \models_m \varphi$ o $M \models_m \psi$.
5. $M \models_m [a]\varphi$ sii para todo mundo m' en W, $m -a\rightarrow m'$ implica $M \models_{m'} \varphi$.
6. $M \models_m [a]_i\varphi$ sii para todo mundo m' en W, $m' -a\rightarrow m$ implica $M \models_{m'} \varphi$.

Decimos que una fórmula φ es válida en un modelo $M = \langle W, \{-a\rightarrow \mid a \in L\}, V\rangle$ (Not.: $M \models \varphi$) sii para todo mundo $m \in W$, $M \models_m \varphi$.

Decimos que una fórmula φ es válida en una clase de modelos C (Not.: $\models_C \varphi$) sii para todo modelo $M \in C$, $M \models \varphi$.

Definición: Sean $M_1 = \langle W_1, \{-a\rightarrow \mid a \in L_1\}, V_1\rangle$ y $M_2 = \langle W_2, \{-a\rightarrow \mid a \in L_2\}, V_2\rangle$ dos modelos, y L un conjunto de labels, f: $W_1 \rightarrow W_2$ es un L-morfismo de M_1 en M_2 si para todo label $a \in L$, para todo $m_i, m_j \in W_1$ vale que $m_i -a\rightarrow m_j$ implica $f(m_i) -a\rightarrow f(m_j)$.

Definición: Sea Γ un conjunto de fórmulas definimos como $L(\Gamma)$ al conjunto de labels de las fórmulas de Γ.

Definición: Sean $M_1 = \langle W_1, \{-a\rightarrow \mid a \in L_1\}, V_1\rangle$ y $M_2 = \langle W_2, \{-a\rightarrow \mid a \in L_2\}, V_2\rangle$ dos modelos y Γ un conjunto de fórmulas del lenguaje de M_1 ($L(\Gamma) \subseteq L_1$).

1. Un Γ-morfismo de M_1 en M_2 es un $L(\Gamma)$-morfismo f: $W_1 \rightarrow W_2$ tal que para toda variable proposicional $p_i \in \Gamma$ vale que $V_1(p_i, m_j) = 1$ implica $V_2(p_i, f(m_j)) = 1$.
2. Una Γ-filtración de M_1 en M_2 es un Γ-morfismo f tal que:

(Sur) f es suryectiva.

(Var) Para toda variable proposicional $p_i \in \Gamma$ y para todo mundo $m_j \in W_1$, $V_1(p_i, m_j) = 1$ si y sólo $V_2(p_i, f(m_j)) = 1$.

(Fil) Para todo label a y toda fórmula φ tal que $[a]\varphi \in \Gamma$, vale que $M_1 \models_{mj} [a]\varphi$ implica $M_1 \models_{mk} \varphi$ para todo par de mundos m_j, m_k en W_1 tal que $f(m_j) -a\rightarrow f(m_k)$ vale en el modelo M_2.

La importancia de la noción de Γ-filtración está dada por el siguiente resultado (que detallamos en forma exhaustiva dado que las filtraciones desempeñarán un papel central en nuestro trabajo):

Teorema: Sea Γ un conjunto de fórmulas cerrado por subfórmulas, y sea f una Γ-filtración de $M_1 = \langle W_1, \{-a\rightarrow \mid a \in L_1\}, V_1\rangle$ en $M_2 = \langle W_2, \{-a\rightarrow \mid a \in L_2\}, V_2\rangle$. Para toda fórmula $\varphi \in \Gamma$, para todo mundo m_i en W_1 vale que $M_1 \models_{mi} \varphi$ si y sólo si $M_2 \models_{f(mi)} \varphi$.

dem:

La demostración procede por inducción en la complejidad de la fórmula $\varphi \in \Gamma$.

El caso $\varphi = \top$ es trivial, todo modelo valida $\top$.

Si $\varphi = p_j$ un símbolo proposicional entonces dado que φ está en Γ la condición (Var) nos garantiza $M_1 \models_{mi} p_j$ si y sólo si $M_2 \models_{f(mi)} p_j$.

Hipótesis Inductiva: Para φ de complejidad n, vale $M_1 \models_{mi} \varphi$ si y sólo si $M_2 \models_{f(mi)} \varphi$.

Sea φ una fórmula de complejidad n+1 entonces:

$\varphi = \neg\psi$ y tenemos que $M_1 \models_{mi} \neg\psi$ sii no es el caso que $M_1 \models_{mi} \psi$ sii (por HI, que puede aplicarse puesto que Γ es cerrado por subfórmulas y por lo tanto $\psi \in \Gamma$) no es el caso que $M_2 \models_{f(mi)} \psi$ sii $M_2 \models_{f(mi)} \neg\psi$.

$\varphi = (\psi \vee \theta)$ y tenemos que $M_1 \models_{mi} (\psi \vee \theta)$ sii $M_1 \models_{mi} \psi$ o $M_1 \models_{mi} \theta$ sii (por HI, puesto que $\psi, \theta \in \Gamma$) $M_2 \models_{f(mi)} \psi$ o $M_2 \models_{f(mi)} \theta$ sii $M_2 \models_{f(mi)} (\psi \vee \theta)$.

$\varphi = [a]\psi$ para algún label a.

Este es el caso interesante. Demostremos la doble implicación.

$\Rightarrow$) Sea m_i cualquiera, supongamos que $M_1 \models_{mi} [a]\psi$.

Sea m' cualquiera en W_2 tal que $f(m_i) \text{–a}\rightarrow m'$. Como f es suryectiva, tenemos que existe $m_j \in W_1$ tal que $f(m_j) = m'$. Pero entonces (Fil) garantiza que $M_1 \models_{mj} \psi$, ahora por HI $M_2 \models_{f(mj)} \psi$. O sea $M_2 \models_{m'} \psi$ y como m' era cualquiera tal que $f(m_i) \text{–a}\rightarrow m'$ tenemos $M_2 \models_{f(mi)} [a]\psi$.

$\Leftarrow$) Supongamos ahora que $M_2 \models_{f(mi)} [a]\psi$ y sea m' cualquiera, $m' \in W_1$ tal que $m_i \text{–a}\rightarrow m'$. Como $\varphi \in \Gamma$ tenemos $a \in L(\Gamma)$. Por ser f un $L(\Gamma)$-morfismo tenemos que $f(m_i) \text{–a}\rightarrow f(m')$ y por lo tanto $M_2 \models_{f(m')} \psi$. Por HI ahora, $M_1 \models_{m'} \psi$ y entonces $M_1 \models_{mi} [a]\psi$. □

El resultado anterior nos muestra como la existencia de una Γ-filtración entre dos modelos asegura que las fórmulas de Γ son "tratadas igual" por ambos modelos, a pesar de las diferencias que pudiera haber entre ellos. Es decir que las Γ-filtraciones son la herramienta que nos permite decir cuándo dos modelos son equivalentes respecto de un conjunto de fórmulas. Aparte de su interés teórico en la demostración de la decidibilidad de la lógica **KPI**, veremos más adelante la gran importancia que este método posee desde un punto de vista práctico.

El resultado que afirma que el sistema **KPI** es adecuado y decidible en la clase de todos los modelos se sigue de los resultados de adecuación y decidibilidad de la lógica polimodal mínima y la lógica temporal mínima, más resultados de preservación (estos resultados pueden verse por ejemplo en [Popkorn, 1994]).

Teorema: El sistema **KPI** es adecuado y decidible en la clase de todos los modelos.

3.3 Estado del Arte en Verificación de Propiedades

El uso de lógicas modales para la verificación de propiedades no es una idea original en el área de Computación. Diversas lógicas se han propuesto para trabajar en la verificación de algoritmos, especialmente, algoritmos concurrentes (PTL, Propositional Temporal Logic [Pnueli, 1977; Manna y Pnueli, 1981]; TLA, Temporal Logic of Actions [Lamport, 1994]; CTL, Computational Tree Logic [Clarke et al., 1986], etc.). Todas ellas están basadas en la premisa de que mostrar que un programa satisface su especificación expresada como una fórmula temporal o modal es equivalente a probar que la fórmula es verdadera en un estado (o conjunto de estados) o en un conjunto de corridas en el sistema de transición que modela el programa.

Cada lógica fue diseñada para atacar un determinado problema, pero en general, pueden agruparse en dos grandes familias de acuerdo al uso que de ellas se hace.

PTL y TLA, por ejemplo, apuntan a caracterizar sintácticamente un algoritmo mediante un conjunto de fórmulas para luego demostrar que de esa caracterización se implican las propiedades que se desean verificar.

La propuesta de Manna y Pnueli para PTL utiliza técnicas de teoría de prueba. Ellos proponen que el programa, vía su modelo, sea codificado como una teoría (un conjunto de fórmulas temporales) Γ. La propiedad requerida debe entonces seguirse de, o ser una consecuencia temporal lógica de Γ. Una axiomatización correcta y completa de las fórmulas que son teoremas de la lógica provee el marco en el cual estas deducciones pueden ser presentadas. Este método ve al sistema de transición de un programa como secundario, que será dejado atrás una vez que la teoría Γ halla sido obtenida.

La idea básica de Lamport para TLA es similar (métodos de teoría de prueba) pero en este caso las modalidades se definen sobre una lógica de acciones (en vez de una lógica proposicional), lo que brinda una mayor riqueza expresiva en el lenguaje objeto. De esta forma pueden escribirse, por ejemplo, directamente en el lenguaje lógico,

fórmulas que representan asignaciones a variables (utilizando la relación de identidad) simplificando en gran medida la tarea de caracterización.

CTL en cambio, permite sólo la enunciación de propiedades que son luego verificadas sobre un determinado modelo del algoritmo (representado mediante su sistema de transición de estados finito). Esto constituye un método más directo de establecer que un conjunto de estados o corridas tenga una dada propiedad, y esta basado en técnicas de tableaux. Este método es el soporte de los *model checkers*, algoritmos que establecen automáticamente si una propiedad dada es cierta en un sistema de estados finito.

La principal diferencia entre estos dos enfoques es que el primero se apoya en una caracterización sintáctica del programa (mediante fórmulas) mientras que el trabajo en el segundo caso es principalmente semántico, basándose en el modelo específico del problema a tratar y utilizando la definición semántica de los operadores.

Si bien el enfoque axiomático parece más abstracto y general, debe tratar con la complejidad que caracteriza la demostración simbólica de teoremas. Los demostradores automáticos existentes hasta ahora, deben en general recurrir a la ayuda del especialista para decidir en pasos cruciales de una demostración y usualmente es necesario un alto grado de ingenio para organizar la prueba en forma manejable.

El enfoque semántico necesita sólo chequear que cada fórmula determinada sea válida en el modelo, utilizando algoritmos específicos para esta tarea (los model checkers). El problema de validez en un modelo en las lógicas modales utilizadas en este área es siempre decidible y de baja complejidad, lo que permite el desarrollo de algoritmos de chequeo eficientes.

Los sistemas lógicos nombrados arriba son los ejemplos más clásicos que se hallan en cada área. En los últimos años se han presentado toda una serie de modificaciones, restricciones y nuevos lenguajes que pueden de una forma u otra relacionarse con los que acabamos de describir, entre ellos podemos nombrar CTL* (CTL con cuantificación arbitraria sobre fórmulas lineales), PTLA (la restricción proposicional de TLA), RPTL (PTL para Tiempo Real) y RTTL (lógica de tiempo real con restricción por intervalos de tiempo). Sin embargo, los últimos avances en verificación de propiedades utilizando lógicas modales se han dado empleando técnicas semánticas, correspondiendo así al predominio del enfoque semántico en toda la Lógica moderna.

El principal problema al que se enfrenta el área de model checking es la explosión combinatoria de estados que se produce al analizar programas concurrentes. Las técnicas de reducción de estados en los modelos y la optimización de los algoritmos de chequeo permiten en la actualidad verificar propiedades sobre modelos de millones de estados y más, y esto en nuestro caso es más que suficiente ya que un diseño nunca superará tal magnitud.

La lógica **KPI** fue definida teniendo este predominio en mente, y por eso responde adecuadamente a las técnicas semánticas de validación, siendo menos útil como una herramienta puramente sintáctica.

4 La Lógica Modal como herramienta de Ingeniería de Software

4.1 Motivación: Por qué usar Lógica Modal en Diseño

Ya comentamos brevemente en secciones anteriores la necesidad de introducir un mayor nivel de formalización en el lenguaje utilizado en el Diseño de Software y también dimos indicios de las razones que guiaron la elección de la Lógica Modal como el método para alcanzar este objetivo.

Daremos como introducción a las nuevas propuestas de Diseño una presentación más detallada de nuestra motivación:

El primer gran punto de contacto que encontramos entre ambas áreas fue la similitud entre los modelos utilizados en una y otra. Lo que en Diseño era representado mediante un módulo con ciertas características, como un nombre X, un número de versión Y, etc., bien podía ser un mundo posible en el cual las propiedades "mi nombre es X" y "mi versión es Y" fueran validadas, mientras que las relaciones de interacción entre los módulos no eran más que relaciones de accesibilidad.

Sin embargo, una mera similitud en las estructuras utilizadas no era suficiente para justificar la introducción de todo el aparato modal. El uso que se daba a ambas estructuras debía también ser similar. Nos preguntamos entonces cual era el concepto fundamental en ambas estructuras y la respuesta fue: las relaciones. Eran éstas las que en ambas jugaban el papel principal determinando las propiedades del modelo y la mayor parte de la información que del mismo podía derivarse.

Si la lógica modal pudiera ser adaptada en una forma simple y natural para manipular a los propios diseños, obtendríamos una herramienta que unificaría el lenguaje utilizado para caracterizar los modelos y sus propiedades, con capacidad de demostración formal, una semántica no ambigua y la posibilidad de realizar verificación automática.

La lógica **KPI** obtenida a partir de la lógica mínima **K** mediante extensiones tradicionales, posee todas estas características.

4.2 Nuevas Propuestas en Diseño

Como dijimos anteriormente, la principal herramienta utilizada en la actualidad para la verificación de propiedades es el model checking. Es por esto que nuestro trabajo tiene por principal meta estudiar la aplicación de este concepto a la verificación de propiedades de diseño. En relación con esta tarea se desarrollarán también algunas herramientas que permiten la manipulación de los modelos a chequear.

Finalmente discutiremos brevemente la posibilidad de un enfoque sintáctico y los problemas que este intento trae aparejado.

Durante toda esta sección trabajaremos sobre el problema del KWIC (Key Word in Context) para mostrar cómo las nuevas herramientas propuestas son usadas. Este problema fue introducido en [Parnas, 1972] y es muy utilizado como instrumento de enseñanza en Ingeniería de Software.

El sistema de indexado KWIC acepta un conjunto ordenado de líneas, donde cada línea es un conjunto ordenado de palabras, y cada palabra es un conjunto ordenado de caracteres. A cualquier línea se le puede aplicar en forma repetida un "shift circular" removiendo la primer palabra de la línea y colocándola al final de ésta. El sistema de indexado KWIC retorna como resultado una lista ordenada alfabéticamente de todos los shift circulares de todas las líneas.

Se han propuesto varias descomposiciones en módulos para este problema, [Garlan & Shaw, 1993]; en esta sección utilizaremos una descomposición basada en tipos abstractos de datos.

La descomposición consta de un componente principal de control, y otros cinco componentes principales que a su vez están divididos en submódulos a través de los cuales se accederán a los tipos de datos:

Entrada. Este módulo lee las líneas del medio de entrada y las almacena en forma interna.

Caracteres. Provee funciones para acceder a los caracteres de las palabras de las líneas y para contar la cantidad de palabras en una línea.

Shift_Circular. Provee funciones similares a las del módulo Caracteres pero en este caso trabaja no sólo sobre las líneas de entrada, sino sobre todo posible shift de las mismas.

Shift_Alfabético. Este módulo consiste principalmente de dos funciones. Alfabetizador realiza la alfabetización de las líneas y I-ésimo retorna el índice del shift circular que está en la i-ésima posición en el orden alfabético.

Salida. Este módulo dará la salida deseada del conjunto de shift circulares de las líneas.

En esta descomposición se tienen tres relaciones básicas entre los componentes: es_parte_de, invoca y entrada/salida.

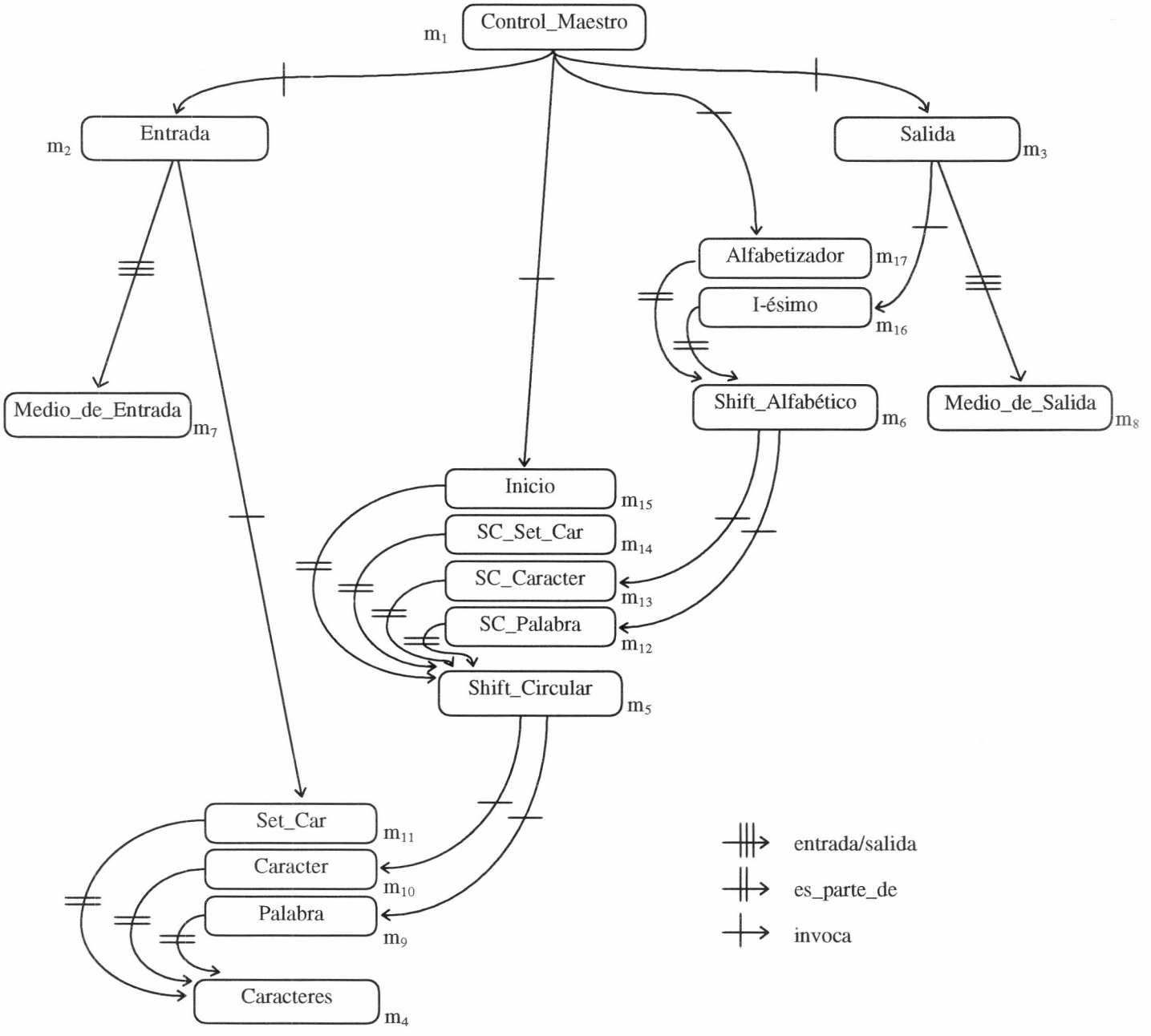

Figura N° 1: Diseño del KWIC

Ya hemos dicho que la forma más natural de describir un diseño es realizar un gráfico en el cual usamos un símbolo (generalmente una caja o un rectángulo) para representar los componentes, y líneas que los conectan para representar las relaciones entre ellos, como en la figura N° 1.

Grafo de Diseño

Podemos también dar una definición matemática de este dibujo:

Definición: Un esquema general de diseño se define como un grafo etiquetado dirigido <N, E, LN, LE, R_{LN}, R_{LE}>, donde:

N es un conjunto de Nodos (finito, no vacío).

E es un conjunto de Ejes ($E \subseteq N \times N$).

LN y LE son conjuntos de etiquetas (labels) (disjuntos y finitos).

R_{LN} es la función total en LN que asigna un label a cada nodo (R_{LN}: $N \rightarrow LN$).

R_{LE} es la relación que asigna por lo menos un label a cada eje ($R_{LE} \subseteq E \times LE$, $\Pi_1(R_{LE}) = E$).

En otras palabras: las cajas son *nodos*, las flechas son *ejes*, los nombres de las cajas son *labels de nodos*, los nombres de las flechas son *labels de ejes*, dar un nombre a una caja es *agregar un par en* R_{LN} y dar un nombre a una flecha es *agregar un par en* R_{LE}.

Ejemplo: El grafo asociado al diseño de la figura N° 1 será G = <N, E, LN, LE, R_{LN}, R_{LE}> con:

N = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}

E = {(1, 2), (1, 3), (1, 15), (1, 17), (2, 7), (2, 11), (3, 8), (3, 16), (5, 9), (5, 10), (6, 12), (6, 13), (9, 4), (10, 4), (11, 4), (12, 5), (13, 5), (14, 5), (15, 5), (16, 6), (17, 6)}

LN = {Control_Maestro, Entrada, Salida, Caracteres, Shift_Circular, Shift_Alfabético, Medio_de_Entrada, Medio_de_Salida, Palabra, Caracter, Set_Car, SC_Palabra, SC_Caracter, SC_Set_Car, Inicio, I-ésimo, Alfabetizador}

LE = {/, //, ///}

R_{LN} = {(1, Control_Maestro), (2, Entrada), (3, Salida), (4, Caracteres), (5, Shift_Circular), (6, Shift_Alfabético), (7, Medio_de_Entrada), (8, Medio_de_Salida), (9, Palabra), (10, Caracter), (11, Set_Car), (12, SC_Palabra), (13, SC_Caracter), (14, SC_Set_Car), (15, Inicio), (16, I-ésimo), (17, Alfabetizador)}

R_{LE} = {((1, 2), /), ((1, 3), /), ((1, 15), /), ((1, 17), /), ((2, 7), ///), ((2, 11), /), ((3, 8), ///), ((3, 16), /), ((5, 9), /), ((5, 10), /), ((6, 12), /), ((6, 13), /), ((9, 4), //), ((10, 4), //), ((11, 4), //), ((12, 5), //), ((13, 5), //), ((14, 5), //), ((15, 5), //), ((16, 6), //), ((17, 6), //)}

En la construcción anterior la relación $-/\rightarrow$ representa invoca, $-//\rightarrow$ representa es_parte_de y $-///\rightarrow$ representa entrada/salida.

Comenzaremos ahora a desarrollar las distintas herramientas que utilizaremos para la verificación de propiedades de diseño.

De Grafo de Diseño a Modelo Modal

Dado que trabajaremos con lógicas modales, debemos transformar el grafo que representa un determinado diseño al nuevo formalismo.

Definición: Dado un grafo <N, E, LN, LE, R_{LN}, R_{LE}> correspondiente a un diseño, según la definición introducida en la sección anterior, el modelo de Kripke asociado es M = <W, {$-a\rightarrow$ | a ∈ L}, V> donde:

$W = \{m_i \mid i \in N\}$.

L = LE.

$-a\rightarrow = \{(m_k, m_l) \in W \times W \mid ((k, l), a) \in R_{LE}\}$.

Para $p_i \in \mathbb{P}$, $m_j \in W$, $V(p_i, m_j) = 1$ si $(j, li) \in R_{LN}$, $V(p_i, m_j) = 0$ si no.

Proposición : Si G es un grafo de diseño, *M* obtenido a partir de G es un modelo de Kripke.

dem:

Tenemos que mostrar que $M = <W, \{-a\rightarrow \mid a \in L\}, V>$ obtenido a partir de G, cumple con la definición de modelo de Kripke:

1. $W \neq \varnothing$ sii $N \neq \varnothing$, asegurado por definición de G.
2. Sólo se pide que L sea un conjunto finito y LE es finito por definición de G.
3. Sea a cualquiera en L, $-a\rightarrow \subseteq W \times W$ por construcción de *M*.
4. V es una relación en $((\mathbb{P} \times W) \times \{0, 1\})$ por construcción, falta mostrar que V es función total, pero esto está asegurado por ser R_{LN} función total. □

De la definición de grafo de diseño surge que el número de variables proposicionales que son validadas en algún mundo del modelo es finito y coincide con la cantidad de labels que se asignaron a nodos en el grafo. Podemos interpretar la variable de proposición p_{ln} como "El nombre del módulo es ln".

Ejemplo: Partiendo del grafo de diseño de la figura N° 1 se obtiene el modelo $M = <W, \{-/\rightarrow, -//\rightarrow, -///\rightarrow\}, V>$ con:

$W = \{m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9, m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, m_{15}, m_{16}, m_{17}\}$

$-/\rightarrow = \{(m_1, m_2), (m_1, m_3), (m_1, m_{15}), (m_1, m_{17}), (m_2, m_{11}), (m_3, m_{16}), (m_5, m_9), (m_5, m_{10}), (m_6, m_{12}), (m_6, m_{13})\}$

$-//\rightarrow = \{m_9, m_4), (m_{10}, m_4), (m_{11}, m_4), (m_{12}, m_5), (m_{13}, m_5), (m_{14}, m_5), (m_{15}, m_5), (m_{16}, m_6), (m_{17}, m_6)\}$

$-///\rightarrow = \{(m_2, m_7), (m_3, m_8)\}$

V(Control_Maestro, m_1) = 1,	V(Entrada, m_2) = 1,	V(Salida, m_3) = 1,
V(Caracteres, m_4) = 1,	V(Shift_Circular, m_5) = 1,	V(Shift_Alfabético, m_6) = 1,
V(Medio_de_Entrada, m_7) = 1,	V(Medio_de_Salida, m_8) = 1,	V(Palabra, m_9) = 1,
V(Caracter, m_{10}) = 1,	V(Set_Car, m_{11}) = 1,	V(SC_Palabra, m_{12}) = 1,
V(SC_Caracter, m_{13}) = 1,	V(SC_Set_Car, m_{14}) = 1,	V(Inicio, m_{15}) = 1,
V(I-ésimo, m_{16}) = 1,	V(Alfabetizador, m_{17}) = 1,	$V(p_i, m_j) = 0$ para todo otro par (p_i, m_j)

Con $-/\rightarrow$ la relación correspondiente a invoca, $-//\rightarrow$ a es_parte_de y $-///\rightarrow$ entrada/salida.

Demostración de Propiedades en el Modelo

Daremos a continuación el algoritmo que permite chequear la validez de una fórmula de **KPI** en un determinado modelo. Este algoritmo no es más que una adaptación del algoritmo en [Clarke et al., 1986] a los operadores de **KPI**.

Supóngase que quiere determinarse la validez de una fórmula φ en un modelo *M*. Nuestro algoritmo procederá en etapas trabajando recursivamente en el tamaño de las subfórmulas de φ. Al final de cada etapa, la validez de cada subfórmula de tamaño n habrá sido determinada y la subfórmula podrá interpretarse como un nuevo símbolo proposicional p_i. El algoritmo construirá para cada mundo m_j, un conjunto C_j formado por las subfórmulas de φ validadas en dicho mundo. Puesto que φ es subfórmula de sí misma, al terminar el algoritmo, m_j validará φ si y sólo si $\varphi \in C_j$.

Definición: Dada una fórmula φ de **KPI** y un modelo $M = \{W, \{-a\rightarrow \mid a \in L\}, V\}$ finito, el siguiente algoritmo determina para todo mundo m_j los conjuntos C_j de subfórmulas de φ válidas en m_j.

Algoritmo:

1. Para todo m_j, hacer
   $C_j = \varnothing$
2. Para todo $\psi \in Sub(\varphi)$ & $Tam(\psi) = 1$
   Para todo $m_j \in W$
   Si $(\psi = \top)$ entonces $C_j := C_j \cup \{\top\}$
   Si $(\psi = p_i$ & $V(p_i, m_j) = 1)$ entonces $C_j := C_j \cup \{p_i\}$

3. Para n = 2 hasta Tam(φ)
   Para todo ψ ∈ Sub(φ) & Tam(ψ) = n
   Para todo $m_j \in W$
   Si ($\psi = \neg\theta$ & $\theta \notin C_j$) entonces
   $C_j := C_j \cup \{\neg\theta\}$
   Si ($\psi = (\theta \vee \xi)$ & ($\theta \in C_j$ o $\xi \in C_j$)) entonces
   $C_j := C_j \cup \{(\theta \vee \xi)\}$
   Si ($\psi = [a]\theta$ & $((\forall m_k)(m_j -a\rightarrow m_k \Rightarrow \theta \in C_k)$) entonces
   $C_j := C_j \cup \{[a]\theta\}$
// de donde Si ($\psi = \langle a\rangle\theta$ & $((\exists m_k)(m_j -a\rightarrow m_k$ & $\theta \in C_k)$) entonces
// $C_j := C_j \cup \{\langle a\rangle\theta\}$
   Si ($\psi = [a]_i\theta$ & $((\forall m_k)(m_k -a\rightarrow m_j \Rightarrow \theta \in C_k)$) entonces
   $C_j := C_j \cup \{[a]_i\theta\}$
// de donde Si ($\psi = \langle a\rangle_i\theta$ & $((\exists m_k)(m_k -a\rightarrow m_j$ & $\theta \in C_k)$) entonces
// $C_j := C_j \cup \{\langle a\rangle_i\theta\}$
//
// El algoritmo es implementado como la función *check* en el Apéndice.

El siguiente teorema demuestra la correctitud del algoritmo anterior.

Teorema: Para toda fórmula φ de **KPI** el algoritmo anterior termina, y al terminar tenemos que para toda subfórmula ψ ∈ Sub(φ), $M \models_{mj} \psi$ sii ψ ∈ C_j.

dem:

Dado que φ es una fórmula de **KPI**, tiene tamaño finito. Por lo que el conjunto Sub(φ) de sus subfórmulas es finito. Además, el modelo *M* es finito. Estos dos hechos aseguran terminación.

Podemos demostrar que el algoritmo obtiene un resultado correcto por inducción en la complejidad de la fórmula ψ ∈ Sub(φ).

Si ψ = ⊤ entonces $M \models_{mj} \top$ (por definición de $\models$) y ⊤ ∈ C_j (por el paso 2).

Si $\psi = p_i$ entonces $M \models_{mj} p_i$ sii $V(p_i, m_j) = 1$ sii $p_i \in C_j$ (por el paso 2).

Hipótesis Inductiva: Para ψ de complejidad n, vale $M \models_{mj} \psi$ sii ψ ∈ C_j.

Sea ψ una fórmula de complejidad n+1 entonces:

$\psi = \neg\theta$, entonces $M \models_{mj} \neg\theta$ sii no es el caso que $M \models_{mj} \theta$ sii (por HI y dado que θ ∈ Sub(φ)) tenemos $\theta \notin C_j$. Y por el paso 3, $\neg\theta \in C_j$.

$\psi = (\theta \vee \xi)$, entonces $M \models_{mj} (\theta \vee \xi)$ sii ($M \models_{mj} \theta$ o $M \models_{mj} \xi$) sii (por HI y θ, ξ ∈ Sub(φ)) tenemos ($\theta \in C_j$ o $\xi \in C_j$). Y por el paso 3, $(\theta \vee \xi) \in C_j$.

$\psi = [a]\theta$, entonces $M \models_{mj} [a]\theta$ sii para todo m_k tal que $m_j -a\rightarrow m_k$ pasa que $M \models_{mk} \theta$ sii (por HI y θ ∈ Sub(φ)) tenemos que para todo m_k tal que $m_j -a\rightarrow m_k$ pasa que θ ∈ C_k. Y por el paso 3, $[a]\theta \in C_j$.

$\psi = [a]_i\theta$, entonces $M \models_{mj} [a]_i\theta$ sii para todo m_k tal que $m_k -a\rightarrow m_i$ pasa que $M \models_{mk} \theta$ sii (por HI y θ ∈ Sub(φ)) tenemos que para todo m_k tal que $m_k -a\rightarrow m_i$ pasa que θ ∈ C_k. Y por el paso 3, $[a]_i\theta \in C_j$. □

Aunque el objetivo de nuestro trabajo no es obtener algoritmos óptimos sino presentar una metodología formal, puede determinarse fácilmente que la complejidad de este algoritmo es del O(t.Max(n, n.e)) donde t es el tamaño de la fórmula φ, n es la cantidad de mundos en *M* y e es la cantidad de ejes en *M*. Notemos que el valor de verdad que *M* asigna a cada subfórmula de φ (t en total) debe ser hallado, y que para cada subfórmula se utilizarán n pasos analizando lo que sucede en cada mundo (si es un símbolo proposicional, o un operador veritativo funcional) o n.e pasos analizando en cada mundo todos los ejes que a él se refieren (si es un operador modal).

La complejidad de nuestro algoritmo es menor que la del algoritmo presentado por [Clarke et al., 1986] debido al cambio en los operadores modales utilizados y principalmente a que no es preciso clausurar nuestro modelo

transitivamente para determinar el valor de las fórmulas. Si se presupone alguna propiedad sobre las relaciones de accesibilidad (transitividad, reflexividad, simetría, etc.), el algoritmo básico puede ser adaptado para tenerla en consideración. Si esta propiedad fuera transitividad, nuestro algoritmo sería la transcripción directa del de Clarke a nuestros operadores.

Ejemplo: Partiendo una vez más del modelo *M* correspondiente al diseño del KWIC, verifiquemos la siguiente propiedad φ = "Un TAD utiliza servicios sólo de otros TADs". En nuestro modelo podemos caracterizar un TAD como aquellos módulos que definen funciones, entonces si un mundo corresponde a un TAD debe valer en él $<//>_i\top$. Decir que sólo utiliza servicios de TADs equivale a decir que todo módulo invocado es parte de un TAD, o formalmente $[/]<//>\top$. Debemos entonces verificar en nuestro modelo que $\varphi = <//>_i\top \rightarrow [/]<//>\top$ es válida en todo mundo. (Notar como esta propiedad sólo involucra a las relaciones del modelo y no a los propios módulos que pueden ser abstraídos mediante la utilización del símbolo $\top$ de tautología).

Daremos a continuación, las construcciones realizadas por nuestro algoritmo. Eliminando las abreviaturas, la fórmula a chequear es $[//]_i\neg\top \vee [/]\neg[//]\neg\top$:

Tam: 1, Sub: $\top$, $C_j = \{\top\}$ para todo m_j en W.

Tam: 2, Sub: $\neg\top$, $C_j = \{\top\}$ para todo m_j en W.

Tam: 3, Sub: $[//]_i\neg\top$, $C_j = \{\top\}$ para j = 4 al 6,
$C_j = \{\top, [//]_i\neg\top\}$ sino.

Sub: $[//]\neg\top$, $C_j = \{\top, [//]_i\neg\top, [//]\neg\top\}$ para j = 1, 2, 3, 7 y 8,
$C_j = \{\top, [//]\neg\top\}$ para j = 4 al 6,
$C_j = \{\top, [//]_i\neg\top\}$ sino.

Tam: 4, Sub: $\neg[//]\neg\top$, $C_j = \{\top, [//]_i\neg\top, [//]\neg\top\}$ para j = 1, 2, 3, 7 y 8,
$C_j = \{\top, [//]\neg\top\}$ para j = 4 al 6,
$C_j = \{\top, [//]_i\neg\top, \neg[//]\neg\top\}$ sino.

Tam: 5, Sub: $[/]\neg[//]\neg\top$, $C_j = \{\top, [//]_i\neg\top, [//]\neg\top\}$ para j = 1,
$C_j = \{\top, [//]_i\neg\top, [//]\neg\top, [/]\neg[//]\neg\top\}$ para j = 2, 3, 7 y 8,
$C_j = \{\top, [//]\neg\top, [/]\neg[//]\neg\top\}$ para j = 4 al 6,
$C_j = \{\top, [//]_i\neg\top, \neg[//]\neg\top, [/]\neg[//]\neg\top\}$ sino.

La fórmula φ, de tamaño 6, es validada en todo mundo ya que al menos uno de sus disyuntos es validado en todo mundo. Para validar φ no es necesario computar todos los pasos explicitados más arriba, por ejemplo al analizar las fórmulas de tamaño 3, ya determinamos que m_i distinto de 4, 5 o 6 valida $[//]_i\neg\top$ y por lo tanto la disyunción que queremos chequear es validada por dichos mundos y podríamos seguir trabajando sólo con m_4, m_5 y m_6.

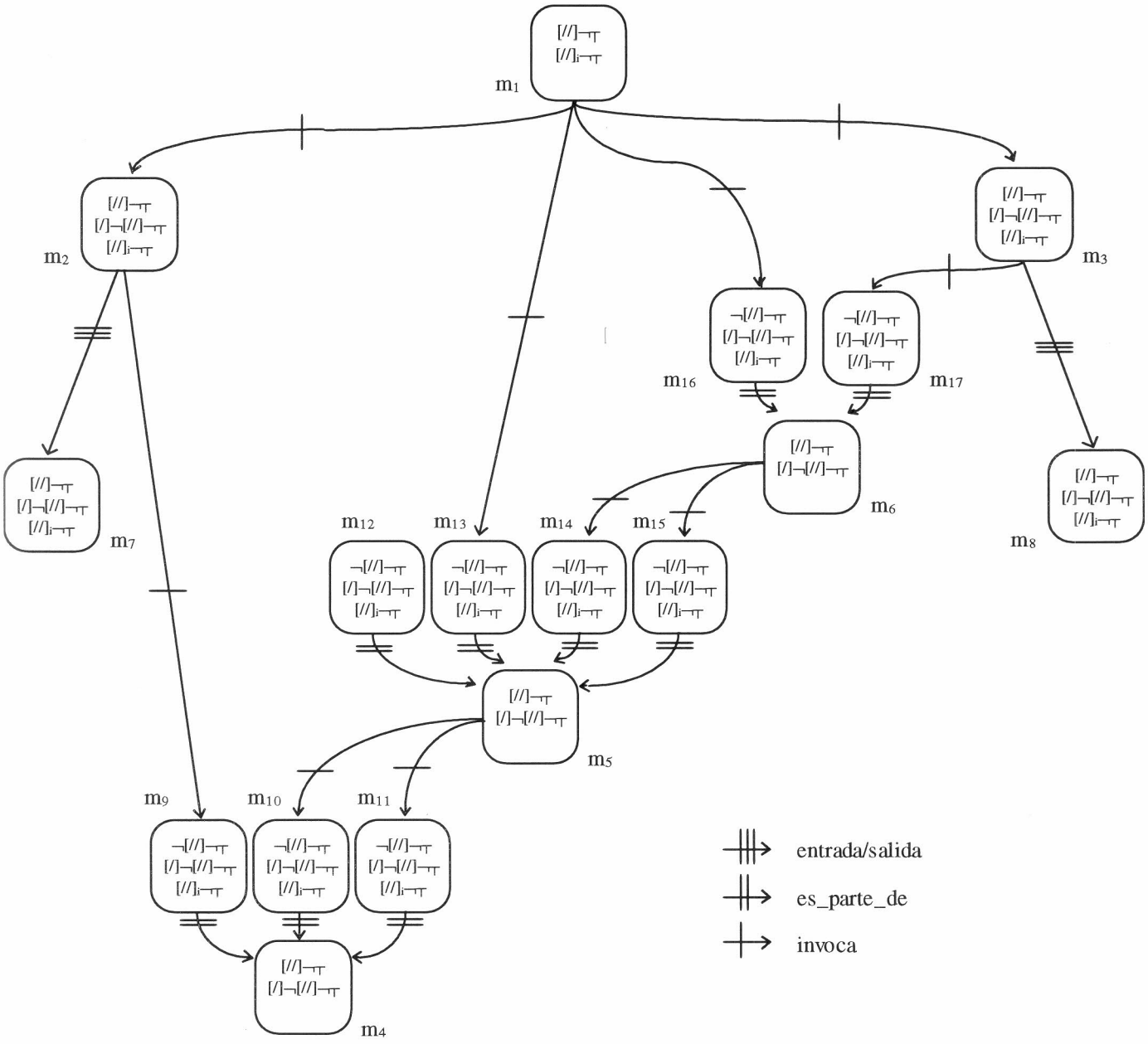

Figura N° 2: Resultado del algoritmo de model checking para la fórmula $\langle // \rangle_i \top \rightarrow [/] \langle // \rangle \top$

Expansión de Modelos

La verificación de un diseño está usualmente asociada a la demostración de propiedades acerca de las relaciones involucradas. Como dijimos anteriormente, dada la complejidad habitual de los diseños, es normal tratar de obtener una visión más global definiendo nuevas relaciones sobre las originales o prestando atención sólo a un cierto subconjunto de las mismas.

En un lenguaje modal, las relaciones están vinculadas a los operadores modales de la lógica. Por eso, el equivalente a crear nuevas relaciones es la definición de nuevas modalidades.

Desde el punto de vista estrictamente lógico y modal, es extraña la idea de definición de nuevas modalidades en la misma lógica. En general, la incorporación de una nueva modalidad, como por ejemplo los operadores P y F de pasado y futuro comentada en "3.2 Extensiones: Lógica Polimodal con Operadores Inversos", da origen a una nueva lógica que se propone modelarla (sintáctica y semánticamente) y está motivada por la búsqueda de una mayor

expresividad o versatilidad en el lenguaje. No es este sin embargo el caso que nos interesa, las nuevas modalidades que intentamos definir están gobernadas por las nuevas relaciones introducidas en el diseño que deben su comportamiento a las relaciones básicas que participaron en su creación. Es decir que procuraremos introducir modalidades a manera de *abreviaturas* del lenguaje y que por otro lado tengan como contrapartida semántica una relación en el modelo.

Las únicas referencias a un propósito similar se encuentran en la Teoría de la Definición Explícita de Beth en lógica clásica de primer orden [Beth, 1953]. No conocemos ningún trabajo realizado en este área en el campo de la lógica modal.

La teoría de Beth describe en qué forma pueden agregarse a la signatura S de un lenguaje de primer orden nuevos símbolos de relación, función y constante formando así una signatura extendida S^+, *sin ampliar el poder expresivo del lenguaje*. Estos nuevos símbolos se definen como equivalentes a fórmulas en el lenguaje original, y Beth demuestra que es posible dar para todo modelo M correspondiente a la signatura original S, un modelo M^+ correspondiente a la signatura extendida S^+ tal que toda fórmula del lenguaje original preserve su validez y las fórmulas que utilicen los nuevos símbolos preserven el significado que se asignó a estas abreviaturas vía su definición.

Este resultado se apoya en los siguientes lemas:

Lema de Expansión: Sean M (de signatura S) y M^+ (de signatura S^+) estructuras de primer orden, tales que M^+ es una S^+-expansión de M, entonces para toda fórmula $\varphi \in \mathcal{L}_S$ vale que, $M \models \varphi$ sii $M^+ \models \varphi$.

Lema de Definición [Beth, 1953]: Sea Γ un conjunto de S-sentencias y $S \subset S^+$. Sea Δ un conjunto de S-definiciones en Γ, una para cada símbolo en $S^+ \setminus S$. Bajo estas condiciones, para toda S-estructura M tal que $M \models \Gamma$ existe una única S^+-expansión M^+ tal que $M^+ \models \Delta$.

El lenguaje correspondiente a la signatura S^+ posee todas las fórmulas del lenguaje de la signatura S más fórmulas que utilizan los nuevos símbols de S^+. El lema de expansión nos dice entonces que el valor de verdad de las fórmulas en $\mathcal{L}_S$ será preservado en la expansión, mientras que el lema de definición más la propiedad de la lógica de primer orden que nos asegura invarianza por sustitución de equivalentes nos asegura que toda fórmula en $\mathcal{L}_{S+}\setminus\mathcal{L}_S$ es equivalente a una fórmula en $\mathcal{L}_S$.

La trascendencia del lema de definición se muestra principalmente en su *generalidad* (basta que las sentencias en Δ sean definiciones según la especificación dada por Beth) y en el resultado de *unicidad* de la extensión.

Parece ser que no podemos aspirar a ninguna de estas dos cualidades en una Teoría de la Definición para la lógica modal. Por un lado, no podremos obtener unicidad, ya que es sabido que debido a la limitación en la expresividad de los lenguajes modales existen siempre modelos estructuralmente distintos a uno dado y que son sin embargo lógicamente equivalentes. Pero por otro lado, nos va a resultar díficil hallar una especificación de qué constituye una *definición* modal correcta: mientras que en lógica de primer orden toda fórmula $\varphi(x_1, ..., x_n)$ por ejemplo, define unívocamente una relación n-aria $R(x_1, ..., x_n)$ (justamente la relación formada por las n-uplas $(a_1, ..., a_n)$ que tales que $\varphi(x_1, ..., x_n)[a_1, ..., a_n]$ es verdadera), no es cierto que toda fórmula modal $\varphi(p_1, ..., p_n)$ defina una modalidad $M(p_1, ..., p_n)$ (o al menos no una modalidad clásica como las utilizadas en la lógica **KPI**).

De todas formas, para la presente tesis necesitamos un resultado mucho menos general que una Teoría de Definición Modal. Dado que nuestro interés es caracterizar sintácticamente la construcción de nuevas relaciones a partir de relaciones básicas, nos basta dar definiciones para los operadores que utilizaremos (composición, unión, etc.) sobre relaciones.

Comencemos entonces por mostrar un resultado similar al Lema de Expansión sobre modelos modales.

Definición: Sea $M = \langle W, \{-a\rightarrow \mid a \in L\}, V\rangle$ un modelo modal, decimos que $M^+ = \langle W^+, \{-a\rightarrow \mid a \in L^+\}, V^+\rangle$ es una L^+-expansión de M sii

1. $W = W^+$
2. $L \subseteq L^+$ y para todo $a \in L$ vale $-a\rightarrow_L = -a\rightarrow_{L+}$
3. $V = V^+$

(Es decir, que M^+ es idéntico en todo a M, salvo en que puede definir nuevas relaciones de accesibilidad.)

Lema de Expansión Modal: Sean $M = \langle W, \{-a\rightarrow \mid a \in L\}, V\rangle$ y $M^+ = \langle W^+, \{-a\rightarrow \mid a \in L^+\}, V^+\rangle$ modelos modales, tales que M^+ es una L^+-expansión de M, entonces para toda fórmula $\varphi \in \mathcal{L}_L$ vale que para todo $m \in W$, $M \models_m \varphi$ sii $M^+ \models_m \varphi$.

dem:

Comencemos por notar que por 1 en la definición de L^+-expansión $W = W^+$ y que por lo tanto tiene sentido afirmar el si y sólo si del lema.

La demostración es simple y procede por inducción en la complejidad de φ. El caso proposicional se resuelve utilizando que $V = V^+$ por 3 en la definición. Por inducción se pasa sobre los operadores booleanos. Para el caso modal basta notar que dado que $\varphi \in \mathcal{L}_L$ sus modalidades deben usar labels en L y para dichos labels las relaciones $-a\rightarrow_L$ y $-a\rightarrow_{L+}$ coinciden por 2 en la definición. □

Ahora bien, el lema anterior nos asegura que las fórmulas que no utilizan nuevos símbolos preservan su valor de verdad de M a M^+, pero que pasa con las fórmulas que utilizan los nuevos operadores? Para las fórmulas en el lenguaje extendido, usaremos también el resultado de invarianza por sustitución del cálculo modal.

Los siguientes resultados nos permiten ahora por ejemplo, caracterizar las operaciones de composición y unión.

Proposición (Composición, $-c\rightarrow = -a\rightarrow \circ -b\rightarrow$): Sea $M = \langle W, \{-a\rightarrow \mid a \in L\}, V\rangle$ un modelo modal cualquiera tal que $a, b \in L$, entonces $M^+ = \langle W, \{-a\rightarrow \mid a \in L\} \cup \{-c\rightarrow \mid c \notin L$ y $m-c\rightarrow m'$ sii existe m'' tal que $m-a\rightarrow m''$ y $m''-b\rightarrow m'\}, V\rangle$ es una $L\cup\{c\}$-expansión de M que satisface que para toda $\varphi \in L_{L\cup\{c\}}$, para todo $m \in W$, $M^+ \models_m \langle c\rangle\varphi \leftrightarrow \langle a\rangle\langle b\rangle\varphi$.

dem:

Por construcción M^+ es una $L\cup\{c\}$-expansión de M.

Mostremos entonces que para toda $\varphi \in L_{L\cup\{c\}}$, para todo $m \in W$, $M^+ \models_m \langle c\rangle\varphi \leftrightarrow \langle a\rangle\langle b\rangle\varphi$.

Sea φ cualquiera en $L_{L\cup\{c\}}$, m cualquiera en W,

supongamos que $M^+ \models_m \langle a\rangle\langle b\rangle\varphi$

sii existe m'' tal que $m-a\rightarrow m''$ y $M^+ \models_{m''} \langle b\rangle\varphi$

sii existe m'' tal que $m-a\rightarrow m''$ y existe m' tal que $m''-a\rightarrow m'$ y $M^+ \models_{m'} \varphi$

sii existe m' tal que (existe m'' tal que $m-a\rightarrow m''-a\rightarrow m'$ y $M^+ \models_{m'} \varphi$)

sii existe m' tal que $m-c\rightarrow m'$ y $M^+ \models_{m'} \varphi$

sii $M^+ \models_m \langle c\rangle\varphi$ □

Proposición (Unión, $-c\rightarrow = -a\rightarrow \cup -b\rightarrow$): Sea $M = \{W, \{-a\rightarrow \mid a \in L\}, V\}$ un modelo modal cualquiera tal que $a, b \in L$ entonces $M^+ = \{W, \{-a\rightarrow \mid a \in L\} \cup \{-c\rightarrow \mid c \notin L$ y $m-c\rightarrow m'$ sii $m-a\rightarrow m'$ o $m-b\rightarrow m'\}\}$ es una $L\cup\{c\}$-expansión de M que satisface que para toda $\varphi \in L_{L\cup\{c\}}$, para todo $m \in W$, $M^+ \models_m \langle c\rangle\varphi \leftrightarrow (\langle a\rangle\varphi \vee \langle b\rangle\varphi)$.

dem:

Por construcción M^+ es una $L\cup\{c\}$-expansión de M.

Mostremos entonces que para toda $\varphi \in L_{L\cup\{c\}}$, para todo $m \in W$, $M^+ \models_m \langle c\rangle\varphi \leftrightarrow (\langle a\rangle\varphi \vee \langle b\rangle\varphi)$.

Sea φ cualquiera en $L_{L\cup\{c\}}$, m cualquiera en W,

supongamos que $M^+ \models_m (\langle a\rangle\varphi \vee \langle b\rangle\varphi)$

sii $M^+ \models_m \langle a\rangle\varphi$ o $M^+ \models_m \langle b\rangle\varphi$

sii existe m' tal que $m-a\rightarrow m'$ y $M^+ \models_{m'} \varphi$ o existe m'' tal que $m-b\rightarrow m''$ y $M^+ \models_{m''} \varphi$

sii existe m' tal que m–a→m' o m–b→m' y $M^+ \models_{m'} \varphi$
sii existe m' tal que m–c→m' y $M^+ \models_{m'} \varphi$
sii $M^+ \models_m \langle c\rangle\varphi$ □

Los resultados de caracterización de operaciones entre relaciones están obviamente limitados por la expresividad de la lógica modal. Por ejemplo, dado que la lógica **KPI** posee el operador de accesibilidad inversa, la inversa de una relación puede ser caracterizada. Esta operación no podría ser caracterizada en lógica modal clásica en vista del resultado que establece la mayor expresividad de lenguaje con operadores de pasado. Esto mismo nos hará imposible caracterizar en **KPI** la operación de complemento, pues la lógica con operador de inaccesibilidad (equivalente a accesibilidad sobre el complemento de la relación) es más expresiva que una lógica con operadores sólo de accesibilidad.

Proposición (Inversa, $-b\rightarrow = (-a\rightarrow)^{-1}$*)*: Sea $M = \langle W, \{-a\rightarrow \mid a \in L\}, V\rangle$ un modelo modal cualquiera tal que $a \in L$, entonces $M^+ = \langle W, \{-a\rightarrow \mid a \in L\} \cup \{-b\rightarrow \mid b \notin L$ y m–b→m' sii m'–a→m $\}, V\rangle$ es una $L\cup\{b\}$-expansión de M que satisface que para toda $\varphi \in L_{L\cup\{b\}}$, para todo $m \in W$, $M^+ \models_m \langle b\rangle\varphi \leftrightarrow \langle a\rangle_i\varphi$.

dem:

Por construcción M^+ es una $L\cup\{b\}$-expansión de M.

Mostremos entonces que para toda $\varphi \in L_{L\cup\{b\}}$, para todo $m \in W$, $M^+ \models_m \langle b\rangle\varphi \leftrightarrow \langle a\rangle_i\varphi$.

Sea φ cualquiera en $L_{L\cup\{b\}}$, m cualquiera en W,

supongamos que $M^+ \models_m \langle a\rangle_i\varphi$

sii existe m' tal que m'–a→m y $M^+ \models_{m'} \varphi$
sii existe m' tal que m–b→m' y $M^+ \models_{m'} \varphi$
sii $M^+ \models_m \langle b\rangle\varphi$ □

Otras operaciones sobre relaciones (como intersección y resta) no puden ser caracterizadas sobre la clase de todos los modelos (otra vez debido a restricciones de expresividad en el lenguaje modal), pero si pueden ser capturadas parcialmente en la clase de los modelos que corresponden a diseños.

Proposición (Intersección, $-c\rightarrow = -a\rightarrow \cap -b\rightarrow$*)*: Sea $M = \{W, \{-a\rightarrow \mid a \in L\}\}$ un modelo que corresponda a un grafo de diseño tal que a, b ∈ L entonces $M^+ = \{W, \{-a\rightarrow \mid a \in L\} \cup \{-c\rightarrow \mid c \notin L$ y m–c→m' sii m–a→m' y m–b→m'$\}\}$ es una $L\cup\{c\}$-expansión de M que satisface que para toda $\varphi \in L_{L\cup\{c\}}$, $p \in \mathbb{P}$, $M^+ \models_m \langle c\rangle(\varphi \wedge p) \leftrightarrow (\langle a\rangle(\varphi \wedge p) \wedge \langle b\rangle(\varphi \wedge p))$.

dem:

Por construcción M^+ es una $L\cup\{c\}$-expansión de M.

Mostremos entonces que para toda $\varphi \in L_{L\cup\{c\}}$, para todo $p \in \mathbb{P}$, para todo $m \in W$, $M^+ \models_m \langle c\rangle(\varphi \wedge p) \leftrightarrow (\langle a\rangle(\varphi \wedge p) \wedge \langle b\rangle(\varphi \wedge p))$.

Sea φ cualquiera en $L_{L\cup\{c\}}$, m cualquiera en W, p cualquiera en $\mathbb{P}$,

supongamos que $M^+ \models_m (\langle a\rangle(\varphi \wedge p) \wedge \langle b\rangle(\varphi \wedge p))$

sii $M^+ \models_m \langle a\rangle(\varphi \wedge p)$ y $M \models_m \langle b\rangle(\varphi \wedge p)$

sii existe m' tal que m–a→m' y $M^+ \models_{m'} (\varphi \wedge p)$ y existe m'' tal que m–b→m'' y $M^+ \models_{m''} (\varphi \wedge p)$

(De la existencia de los mundos m' y m'' que satisfacen las correspondientes fórmulas no se sigue necesariamente que exista un único mundo que las satisfaga. Pero si podemos afirmar esto en nuestro caso, pues M es un modelo correspondiente a un grafo de diseño y por lo tanto si en ambos m' y m'' vale p, podemos afirmar m' = m''.)

sii existe m' tal que m–a→m' y m–b→m' y $M^+ \models_{m'} (\varphi \wedge p)$
sii existe m' tal que m–c→m' y $M^+ \models_{m'} (\varphi \wedge p)$
sii $M^+ \models_m \langle c\rangle(\varphi \wedge p)$ □

Proposición (Resta, $-c\rightarrow = -a\rightarrow \setminus -b\rightarrow$): Sea $M = \{W, \{-a\rightarrow \mid a \in L\}\}$ un modelo que corresponda a un grafo de diseño tal que a, b ∈ L entonces $M^+ = \{W, \{-a\rightarrow \mid a \in L\} \cup \{-c\rightarrow \mid c \notin L$ y $m-c\rightarrow m'$ sii $m-a\rightarrow m'$ y no $m-b\rightarrow m'\}\}$ es una $L\cup\{c\}$-expansión de M que satisface que para toda $\varphi \in L_{L\cup\{c\}}$, $p \in \mathbb{P}$, $M^+ \models_m <c>(\varphi \wedge p) \leftrightarrow (<a>(\varphi \wedge p) \wedge \neg<b>(\varphi \wedge p))$.

dem:

Por construcción M^+ es una $L\cup\{c\}$-expansión de M.

Mostremos entonces que para toda $\varphi \in L_{L\cup\{c\}}$, para todo $p \in \mathbb{P}$, para todo $m \in W$, $M^+ \models_m <c>(\varphi \wedge p) \leftrightarrow (<a>(\varphi \wedge p) \wedge \neg<b>(\varphi \wedge p))$.

Sea φ cualquiera en $L_{L\cup\{c\}}$, m cualquiera en W, p cualquiera en $\mathbb{P}$,

supongamos que $M^+ \models_m (<a>(\varphi \wedge p) \wedge \neg<b>(\varphi \wedge p))$

sii $M^+ \models_m <a>(\varphi \wedge p)$ y $M \models_m \neg<b>(\varphi \wedge p)$

sii existe m' tal que $m-a\rightarrow m'$ y $M^+ \models_{m'} (\varphi \wedge p)$ y no existe m'' tal que $m-b\rightarrow m''$ y $M^+ \models_{m''} (\varphi \wedge p)$

(En un caso similar a la demostración de la proposición anterior, la implicación ⇒ es válida para cualquier modelo pero no así ⇐. Sólo el hecho que existe un único mundo satisfaciendo la proposición p, nos permite pasar de $m-a\rightarrow m'$ y no $m-b\rightarrow m'$ a no existe m'' tal que $m-b\rightarrow m''$)

sii existe m' tal que $m-a\rightarrow m'$ y no $m-b\rightarrow m'$ y $M^+ \models_{m'} (\varphi \wedge p)$

sii existe m' tal que $m-c\rightarrow m'$ y $M^+ \models_{m'} (\varphi \wedge p)$

sii $M^+ \models_m <c>(\varphi \wedge p)$ □

Notar que en estos dos últimos casos la caracterización es mucho más débil que en los casos anteriores. Por un lado, trabajamos sólo sobre la clase de modelos correspondiente a los grafos de diseño, pero además no hemos conseguido una fórmula equivalente para *toda* fórmula en el lenguaje ampliado. Una fórmula $<c>\psi$ donde $\psi \neq (\varphi \wedge p)$ no posee equivalente en el lenguaje original. Deberá entonces tenerse cuidado en las fórmulas a escribir en el lenguaje extendido si se desea mantener el significado de la nueva modalidad como el de una mera abreviatura. O en otras palabras, el lenguaje completo $\mathcal{L}_{L\cup\{c\}}$, es en estos casos potencialmente más expresivo que $\mathcal{L}_L$.

Ejemplo: Como ejemplo de aplicación de esta técnica definamos sobre el modelo del KWIK, a partir de las relaciones básicas invoca y es_parte_de la relación usa_TAD como usa_TAD = invoca ° es_parte_de. Recordemos la definición del modelo original $M = <W, \{-/\rightarrow, -//\rightarrow, -///\rightarrow\}, V>$ con:

$W = \{m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9, m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, m_{15}, m_{16}, m_{17}\}$

$-/\rightarrow = \{(m_1, m_2), (m_1, m_3), (m_1, m_{15}), (m_1, m_{17}), (m_2, m_{11}), (m_3, m_{16}), (m_5, m_9), (m_5, m_{10}), (m_6, m_{12}), (m_6, m_{13})\}$

$-//\rightarrow = \{(m_9, m_4), (m_{10}, m_4), (m_{11}, m_4), (m_{12}, m_5), (m_{13}, m_5), (m_{14}, m_5), (m_{15}, m_5), (m_{16}, m_6), (m_{17}, m_6)\}$

$-///\rightarrow = \{(m_2, m_7), (m_3, m_8)\}$

$V(Control\_Maestro, m_1) = 1$,	$V(Entrada, m_2) = 1$,	$V(Salida, m_3) = 1$,
$V(Caracteres, m_4) = 1$,	$V(Shift\_Circular, m_5) = 1$,	$V(Shift\_Alfabético, m_6) = 1$,
$V(Medio\_de\_Entrada, m_7) = 1$,	$V(Medio\_de\_Salida, m_8) = 1$,	$V(Palabra, m_9) = 1$,
$V(Caracter, m_{10}) = 1$,	$V(Set\_Car, m_{11}) = 1$,	$V(SC\_Palabra, m_{12}) = 1$,
$V(SC\_Caracter, m_{13}) = 1$,	$V(SC\_Set\_Car, m_{14}) = 1$,	$V(Inicio, m_{15}) = 1$,
$V(I\text{-ésimo}, m_{16}) = 1$,	$V(Alfabetizador, m_{17}) = 1$,	$V(p_i, m_j) = 0$ para todo otro par (p_i, m_j)

Con $-/\rightarrow$ la relación correspondiente a invoca, $-//\rightarrow$ a es_parte_de y $-///\rightarrow$ entrada/salida.

El modelo $M^+ = <W, \{\longrightarrow, -/\rightarrow, -//\rightarrow, -///\rightarrow\}, V>$ donde

$\longrightarrow = \{(m_1, m_5), (m_1, m_6), (m_2, m_4), (m_3, m_6), (m_5, m_4), (m_6, m_5)\}$

Con $\longrightarrow$ correspondiente a usa_TAD.

Por la Proposición anterior entonces, el nuevo modelo M^+ usa <> como abreviatura de </><//> proveyendo además de la relación explícita correspondiente a <>.

Abstracción de Modelos

Si bien el método de filtración fue diseñado para la demostración de decidibilidad de una lógica, su resultado concreto es permitir la detección de un modelo más simple que el modelo original y que sin embargo preserva la validez de las fórmulas que tomaron parte de la filtración. Por ello, aparte del valor teórico del método, podemos utilizarlo como un recurso netamente práctico, aplicado a modelos particulares.

El método de filtración nos permite definir un nuevo modelo que actúa como una abstracción del modelo original. Este nuevo modelo nos muestra una visión simplificada, en la cual los detalles sin interés han sido eliminados mientras que nuevos conceptos más generales y abstractos (que eran ocultados por estos mismos detalles) han sido resaltados.

Debemos dejar en claro desde el principio, que la obtención de modelos más simples mediante el método de filtración no tiene como objetivo disminuir la complejidad del método de verificación de propiedades. Esto será obvio un poco más adelante, cuando se vea que el mismo algoritmo de filtración requiere primero establecer la validez de todas las fórmulas del conjunto de filtración.

El método de filtración aspira a un resultado de mayor valor conceptual. El modelo filtrado muestra al diseño tal cual éste se "ve" desde el punto de vista de las propiedades que fueron en un principio elegidas como relevantes para la tarea que se está llevando a cabo y que fueron incluidas en el conjunto Γ de filtración. Esta nueva visión provee información conceptual de mayor nivel que la provista por un simple resultado de validez. Los ejemplos que mostraremos reflejarán el potencial de este método.

Llama la atención la existencia de un método automático que realice esta tarea de abstracción, la cual parece requerir de una buena dosis de ingenio y habilidad, aún si el modelo filtrado no siempre concuerde con el resultado que un diseñador propondría.

El algoritmo que damos a continuación permite a partir de cualquier modelo de diseño obtener esta visión abstracta con la garantía de que una filtración podrá ser definida para el conjunto de fórmulas Γ de interés.

Definición: Sea un modelo $M = \langle W, \{-a\rightarrow \mid a \in L\}, V\rangle$ y un conjunto finito Γ de fórmulas cerrado por subfórmulas, definimos $M^* = \langle W^*, \{-a\rightarrow^* \mid a \in L^*\}, V^*\rangle$ como el modelo filtrado por Γ:

Algoritmo:

1. Correr en M el algoritmo de model checking con la fórmula $\wedge\Gamma$
2. $W' := W$
   $W^* := \varnothing$
   Mientras $W' \neq \varnothing$
   Tomar $m_i \in W'$
   $[m_i] := \{m_i\}$
   $W' := W' \setminus \{m_i\}$
   Para todo $m_j \in W'$
   Si $(C_i = C_j)$ entonces
   $W' := W' \setminus \{m_j\}$
   $[m_i] := [m_i] \cup \{m_j\}$
   $W^* := W^* \cup \{[m_i]\}$
3. $L^* := L(\Gamma)$
4. Para todo eje $m_i -a\rightarrow m_j$ & $a \in L^*$
   $[m_i] -a\rightarrow^* [m_j]$

5. Para todo $p_i \in \Gamma$

Para todo $m_j \in W$

Si ($V(p_i, m_j) = 1$) entonces

$V^*(p_i, [m_j]) := 1$

// Asumiremos $V^*(p_i, [m_j]) := 0$ para todo $m_j \in W$ y para todo $p_i \notin \Gamma$.

//

// El algoritmo es implementado como la función *filtrar* en el Apéndice.

Debemos verificar que el resultado del algoritmo anterior es en realidad un modelo.

Proposición: El algoritmo anterior termina y al terminar el resultado $M^* = <W^*, \{-a\rightarrow^* \mid a \in L^*\}, V^*>$ es un modelo modal.

dem:

Como Γ es finito, $\wedge\Gamma$ es una fórmula de **KPI** y en este caso ya mostramos que el algoritmo de model checking termina, por lo que el paso 1 termina. El paso 2 es un ciclo sobre la cantidad de elementos en W', un número finito de mundos. El paso 4 es un ciclo sobre la cantidad de ejes de *M*, finito. El paso 5 es un doble ciclo en el número de proposiciones en Γ y la cantidad de mundos en W, ambos finitos. Con esto mostramos que el algoritmo termina.

Si *M* es un modelo entonces W es no vacío. Sea entonces $m_i \in W$, tendremos por el paso 2 que $[m_i] \in W^*$.

El paso 3 define L^* como $L(\Gamma)$ el conjunto de labels de Γ.

El paso 4 define los elementos de $-a\rightarrow^*$ para a en L^* como pares de $W^* \times W^*$ por lo que $-a\rightarrow^*$ son relaciones binarias sobre el conjunto de mundos.

El paso 5 define V^* como $V^*(p_i, [m_j]) = V(p_i, m_j)$ si $p_i \in \Gamma$ y $V^*(p_i, [m_j]) = 0$ sino. Resta demostrar que no puede suceder $V^*(p_i, [m_j]) \neq V^*(p_i, [m_k])$ si $[m_j] = [m_k]$.

Si p_i no pertenece a Γ, entonces $V^*(p_i, [m_l]) = 0$ para todo m_l.

Si p_i pertenece a Γ, y $[m_j] = [m_k]$ entonces el paso 2 asegura $C_j = C_k$. Luego m_j y m_k coinciden en el valor asignado a p_i (p_i pertenece a ambos conjuntos o a ninguno). Es decir $V(p_i, m_j) = V(p_i, m_k)$ y por lo tanto $V^*(p_i, [m_j]) = V^*(p_i, [m_k])$.

□

El algoritmo que dimos recién construye a partir de un modelo *M* y un conjunto Γ de fórmulas, un modelo M^* tal que una Γ-filtración puede definirse de *M* en M^*. Veamos esto:

Teorema: Sea $M = <W, \{-a\rightarrow \mid a \in L\}, V>$ y $M^* = <W^*, \{-a\rightarrow^* \mid a \in L^*\}, V^*>$ el obtenido de *M* a partir de un conjunto de fórmulas Γ por el algoritmo anterior. Entonces $f: W \rightarrow W^*$ tal que $f(m_i) = [m_i]$ es una Γ-filtración de W en W^*.

dem:

Para que f sea Γ-filtración, f debe ser un morfismo suryectivo que satisface las condiciones (Var) y (Fil).

Veamos primero algunas propiedades de los objetos construidos por el algoritmo:

El paso 2 construye el conjunto $W^* = \{[m_i] \in \mathcal{P}(W) \mid m_i \in W\}$, con la propiedad que $m_i \in [m_i]$ y $m_j, m_k \in [m_i]$ sii $C_j = C_k$ es decir sii para todo $\varphi \in \Gamma$, $M \models_{mj} \varphi$ sii $M \models_{mk} \varphi$. Podemos definir $m_j \sim m_k$ sii $m_j, m_k \in [m_i]$ para algún i. Esta relación es de equivalencia sobre W y $W^* = W/\sim$.

Pero entonces f es la función canónica que asigna a un elemento de W su clase en W^*, por lo que f es suryectiva.

Para mostrar que f es un $L(\Gamma)$-morfismo, tomemos m_i, m_j en W tal que $m_i -a\rightarrow m_j$ para algún label $a \in L(\Gamma)$. Tenemos que mostrar que $f(m_i) -a\rightarrow^* f(m_j)$. Pero $f(m_i) = [m_i]$ y $f(m_j) = [m_j]$, y el paso 4 asegura que si $m_i -a\rightarrow m_j$ y $a \in L(\Gamma)$ tenemos que $[m_i] -a\rightarrow^* [m_j]$.

El paso 5 del algoritmo asegura que para todo símbolo proposicional p_i en Γ, $V^*(p_i, [m_j]) = 1$ sii $V(p_i, m_j) = 1$. Es decir, para todo $p_i \in \Gamma$, para todo $m_j \in W$, $V(p_i, m_j) = V^*(p_i, f(m_j))$ satisfaciendo la condición (Var).

Para verificar la última condición (Fil) tomemos dos elementos m_i, $m_j \in W$ tal que $f(m_i) -a\rightarrow^* f(m_j)$ para algún a en $L(\Gamma)$. Por el paso 4, $[m_i] -a\rightarrow^* [m_j]$ sii existe $m_k \in [m_i]$ y $m_l \in [m_j]$ tal que $m_k -a\rightarrow m_l$.

Entonces, si tenemos $[a]\varphi \in \Gamma$, $M \models_{mi} [a]\varphi$ implica $M \models_{mk} [a]\varphi$ pues son de la misma clase. Pero entonces $M \models_{ml} \varphi$ pues m_l es accesible desde m_k. Y entonces, otra vez por pertenecer a la misma clase, $M \models_{mj} \varphi$. □

Podemos terminar el análisis del algoritmo, calculando su orden de complejidad. La complejidad del paso 1 es la complejidad del algoritmo de model checking corrido sobre $\Lambda\Gamma$, o sea O(t.Max(n, n.e)), donde t es el tamaño de $\Lambda\Gamma$, n la cantidad de nodos del modelo y e la cantidad total de ejes. El paso 2 calcula las clases de equivalencia y tiene por ello complejidad n^2. El paso 3 copia todos los labels de $\Lambda\Gamma$ (a lo sumo t) al conjunto L^*. El paso 4 copia los ejes en las relaciones de M al modelo M^* y por lo tanto tiene a lo sumo e pasos. Finalmente el paso 5 debe definir V^* para todo mundo en W^* y para todo símbolo de proposición en $\Lambda\Gamma$, lo que insume un máximo de t.n pasos.

Un simple análisis muestra entonces que el algoritmo tiene $O(t.Max(n, n.e)+n^2)$.

Notar que el modelo obtenido dependerá de las fórmulas contenidas en Γ, como también sucede en la definición formal. Es por eso que debemos analizar la composición de Γ si queremos comprender la forma en que el método realiza la abstracción. Notaremos que un Γ bien definido, provocará la abstracción esperada. Veamos algunos ejemplos:

Ejemplo: Comencemos primero con un caso simple. Trabajemos sobre el modelo M^+ del ejemplo de la sección de Expansión de Modelos y supongamos que nos interesa tratar sólo con las propiedades que involucran a la nueva relación usa_TAD definida como la composición de invoca con es_parte_de. Tenemos entonces que lo único que nos interesa son los módulos del modelo (que sintácticamente están representados por los símbolos proposicionales), más la relación mencionada (que podemos abstraer como $[usa\_TAD]_\top$).

Proponemos entonces Γ = {Control_Maestro, Caracteres, ..., Shift_Alfabético, $[usa\_TAD]_\top$}.

Al agregar a Γ los símbolos proposicionales y dado que en nuestro ejemplo los mundos poseen la característica de validar un único símbolo de predicado (su nombre), tendremos $[m_i] = \{m_i\}$ para todo i, con lo que los mundos no serán colapsados en el paso 2 del algoritmo. Como $[usa\_TAD]_\top$ está en Γ el paso 4, copiará todos los ejes $-usa\_TAD\rightarrow$ de M^+ a $(M^+)^*$ mientras que las demás relaciones serán ignoradas por no tener representantes en Γ.

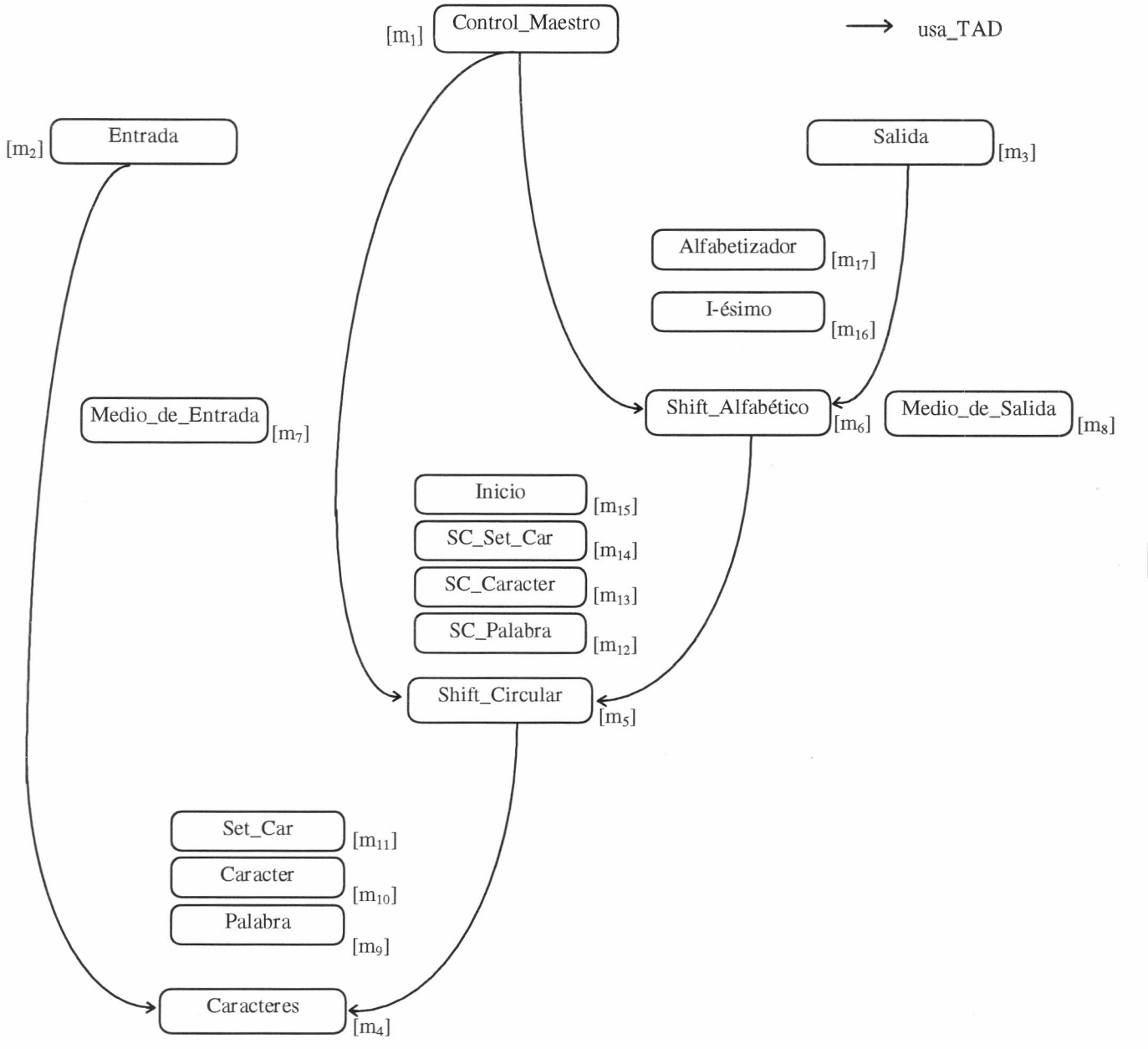

Figura N° 3: Filtración del modelo M^+ del KWIC preservando la relación usa_TAD

Como muestra la figura N° 3, el modelo filtrado preserva la relación que nos interesa analizar, y gracias a que el método de Definición permite introducir la nueva relación en forma explícita y hacerla independiente de las demás, todas las otras relaciones pueden ser eliminadas como irrelevantes. Notar que podemos generalizar este ejemplo para varias relaciones $-a_i\rightarrow$ simplemente agregando a Γ las fórmulas $[a_i]_\top$ correspondientes.

Ejemplo: Observando el ejemplo anterior nos puede llamar la atención aquellos mundos que quedaron totalmente aislados. Estos mundos no son de relevancia para el análisis de la relación usa_TAD y sin embargo no fueron eliminados en la abstracción. Su aparición está justificada por la incorporación de sus nombres en Γ; puesto que eran mencionados, la filtración los preservó. Podemos redefinir Γ con la información obtenida de la filtración anterior como Γ = {Entrada, Salida, Control_Maestro, Caracteres, Shift_Circular, Shift_Alfabético, $[usa\_TAD]_\top$} y ahora sí, todos los mundos aislados serán colapsados en una única clase que los representará en la nueva abstracción.

Este breve ejemplo nos muestra cómo la definición correcta de Γ puede alcanzarse en etapas, utilizando la información obtenida en abstracciones sucesivas.

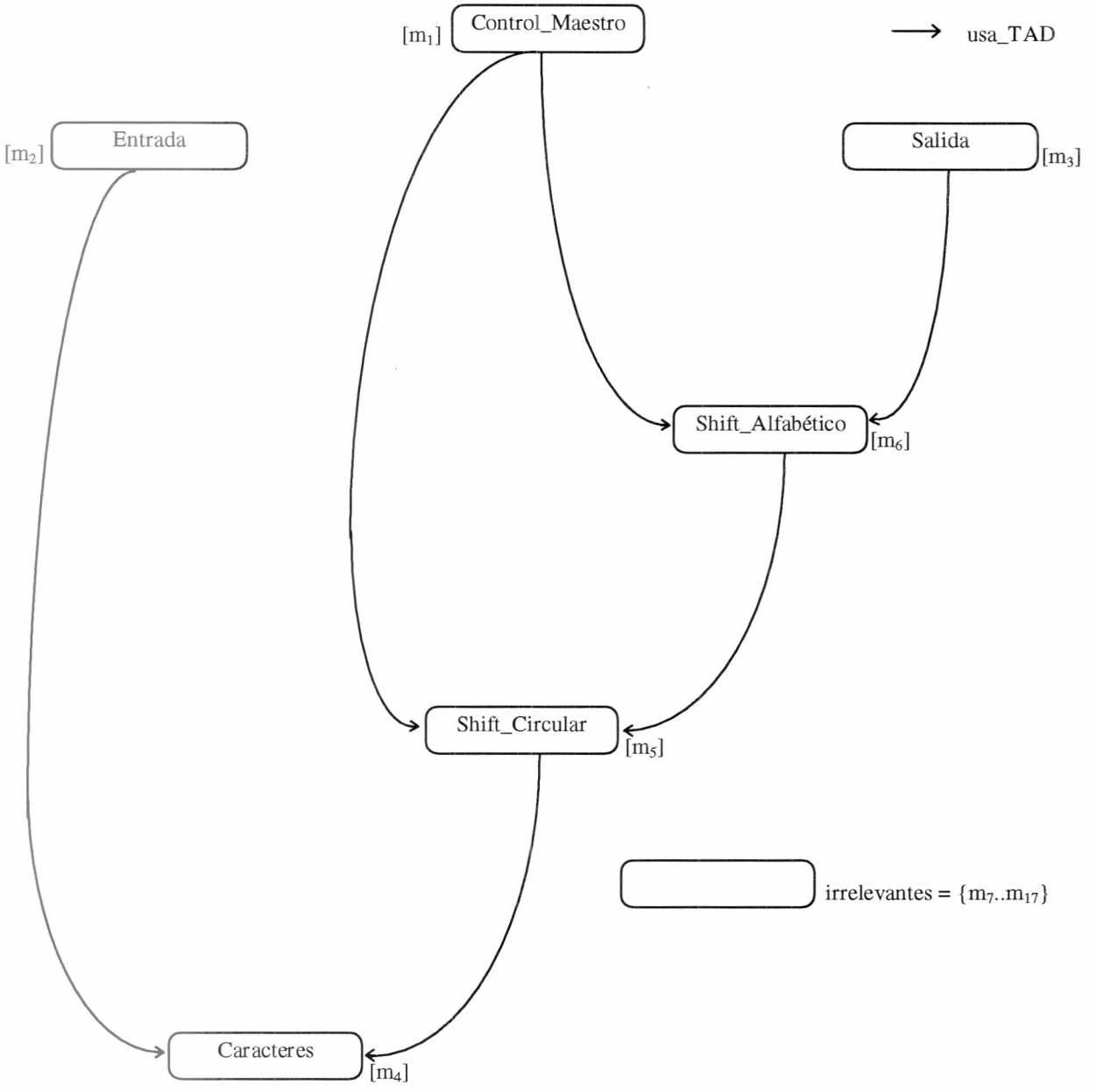

Figura N° 4: Filtración de módulos irrelevantes en el ejemplo anterior

Ejemplo: Los ejemplos anteriores presentan aplicaciones bastantes simples del método de filtración. En cambio, consideramos al siguiente ejemplo como una real muestra del potencial de este método, permitiendo apreciar sus cualidades como una verdadera herramienta para la abstracción de diseños.

Tomemos otra vez el modelo original del KWIC junto con la fórmula que utilizamos en el ejemplo de model checking y hagamos $\Gamma = Sub(<//>_{i\top} \rightarrow [/]<//>_{\top})$. Es decir, queremos obtener una visión del modelo cuando lo que tenemos en mente es la propiedad que ya sabemos válida de "Un TAD utiliza servicios sólo de otros TADs".

El algoritmo nos dará como resultado el modelo de la figura N° 5.

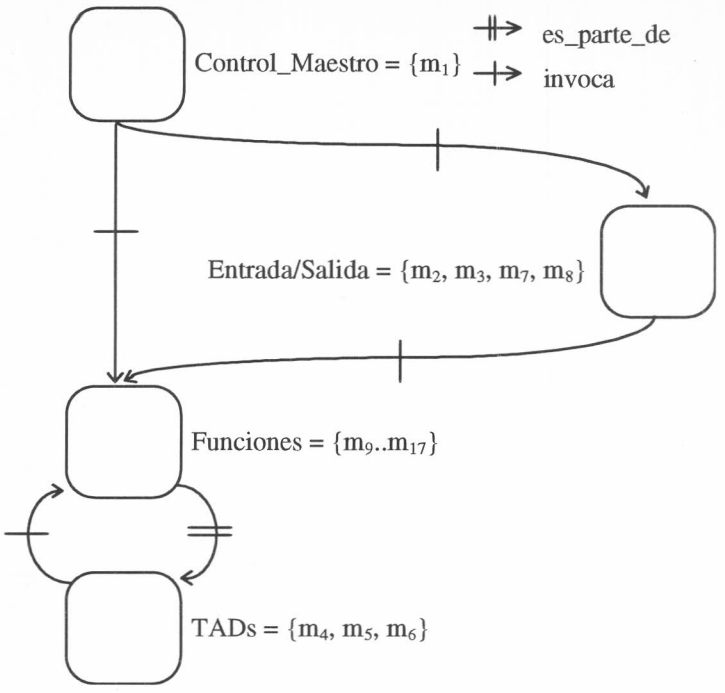

Figura N° 5: Filtración del modelo KWIC por la propiedad $\langle // \rangle_{i\top} \rightarrow [/]\langle // \rangle_{\top}$

Puesto que sólo la propiedad $\langle // \rangle_{i\top} \rightarrow [/]\langle // \rangle_{\top}$ y sus partes estaban en Γ, los módulos del modelo original fueron agrupados de acuerdo al papel que estos cumplían respecto de las mismas. Las subfórmulas de $\langle // \rangle_{i\top} \rightarrow [/]\langle // \rangle_{\top}$ son las que caracterizan las funciones de las distintas entidades del diseño. Es así que obtenemos las clases TADs, Funciones, Entrada/Salida y Control_Maestro que podemos ahora, gracias a la abstracción, distinguir claramente en el modelo original. Además de que en este modelo es visible que los TADs sólo utilizan funciones que son parte de otros TADs el resultado obtenido va mucho más allá: la abstracción nos muestra la misma *arquitectura* del diseño, reflejando la primacía del uso de funciones de TADs y otras peculiaridades, como por ejemplo la administración centralizada de Entrada/Salida a través de Control_Maestro.

Demostración Sintáctica de Propiedades

Hasta el momento, hemos utilizado exclusivamente el método denominado de model checking o método semántico para la verificación de propiedades sobre un diseño pero, como comentamos al principio de esta sección, es posible también un enfoque puramente sintáctico donde las propiedades se demuestran a partir de los axiomas de la lógica en cuestión y un conjunto de fórmulas que caracterizan unívocamente al modelo, utilizando las reglas de inferencia.

Para que el análisis del problema que encara esta tesis sea completo, dedicaremos esta sección justamente a mostrar los inconvenientes que presenta el intento de utilizar el método sintáctico aplicado a la lógica **KPI**.

El método sintáctico se basa en la obtención de un conjunto de fórmulas que caracterizan unívocamente al modelo que nos interesa. Aquí debemos entender que "unívocamente" se interpreta respecto a la expresividad del lenguaje que estamos utilizando. Por ejemplo, dado que el lenguaje modal clásico no nos permite hablar de inaccesibilidad, las fórmulas válidas en un mundo en un modelo dado y las válidas en el modelo que se obtiene al agregar un nuevo mundo aislado son las mismas, y por lo tanto no hay conjunto de fórmulas que nos permitan distinguir entre ambos. Esto sin embargo no constituye un problema que nos ocupe; sabíamos de antemano las limitaciones del lenguaje modal y nos interesa solamente trabajar con aquellas propiedades que sí puedan ser expresadas en la lógica.

Sí tendremos dificultades si a pesar de que dos modelos tienen propiedades diferentes (expresables en la lógica), no podemos construir una descripción que distinga a uno de otro. Si éste es el caso, sólo conseguiremos demostrar

sintácticamente las propiedades que ambos modelos tienen en común y por lo tanto las propiedades que diferenciaban a uno del otro no podrán ser obtenidas.

El caso general de describir cualquier modelo mediante un conjunto de fórmulas es complejo, pero de todas formas no es esto lo que se necesita hacer, ya que los modelos que representan diseños forman una clase muy particular y sólo para ellos interesa obtener descripciones. Lo que hace particular a los modelos de diseño es la restricción impuesta desde un principio sobre las valuaciones. Dado que las valuaciones deben sólo representar el nombre de los módulos del diseño y que no se permiten módulos repetidos, cada mundo hace válido uno y sólo un símbolo proposicional que lo distingue. Además, y puesto que sabemos que la conexidad no puede ser expresada en el lenguaje modal, podemos restringirnos a trabajar con una componente conexa del diseño a la vez y así todos los modelos a analizar son conexos.

Pensemos ahora cómo se describiría en castellano un grafo que tuviera las características de nuestros modelos. La forma obvia es comenzar por un nodo cualquiera diciendo "partimos desde X" y luego describir los posibles caminos que podemos seguir como "desde X por un eje de tipo a se puede ir a Y y por un eje de tipo b a Z" para luego seguir desde estos nodos "a su vez, desde Y puede irse a ...". Lo que queremos decir es que al parecer basta incluir un nodo que marque el origen, y una fórmula que describa cada eje del modelo a partir de este nodo inicial (lo que puede hacerse dado que el modelo es conexo) para obtener una descripción completa del modelo. Hagamos esto más formalmente en un ejemplo.

Ejemplo: Dado el modelo de la figura N° 6 obtenemos a partir del nodo m_1 la siguiente descripción.

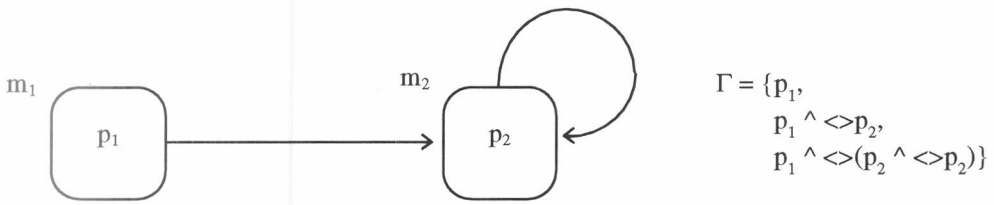

Figura N° 6

Dado que cada nodo m_i está identificado por el símbolo proposicional p_i, la fórmula p_1 marca el nodo inicial y luego se da una fórmula por cada eje del modelo: $p_1 \wedge \Diamond p_2$, dice que partiendo de p_1 se puede llegar hasta p_2 por un eje, mientras que la tercera fórmula dice que desde p_1 se llega a p_2, desde el cual se puede seguir caminando por un eje para llegar nuevamente a p_2.

Notar que esta descripción no es única, y que pueden obtenerse otras dependiendo del criterio que se utilice para recorrer el grafo del modelo (por ejemplo otra descripción para el ejemplo puede lograrse reemplazando la tercer fórmula por $(p_1 \wedge \Diamond(p_2 \wedge \Diamond_i p_2)))$. A pesar de las múltiples descripciones, el modelo mínimo (en mundos y ejes) que respeta la restricción impuesta sobre las valuaciones y valida la descripción es para todas el mismo: el modelo a partir del cual las descripciones fueron construidas.

Lo interesante de este método es que podemos dar un algoritmo que nos retorne una descripción, siguiendo las indicaciones que dimos recién [Hirsch & Areces, 1995].

Con esto, al parecer se solucionan todos los problemas, pues una vez obtenida la descripción del modelo (en una forma automática además) se podría verificar una propiedad φ sobre el diseño demostrando que ésta se deduce del conjunto de fórmulas Γ de la descripción en el sistema axiomático **KPI** ($\Gamma \vdash_{\mathbf{KPI}} \varphi$).

Sin embargo, si bien dadas las restricciones que indicamos hemos construido una descripción que sólo es satisfecha por el modelo del que partimos, no es así para la lógica **KPI**, *pues no le hemos dicho a la lógica que respete nuestras restricciones.*

Ejemplo: Veamos los siguientes modelos que también validan la descripción de la figura N° 6 a partir del nodo m_1.

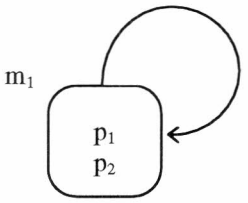

Figura N° 7

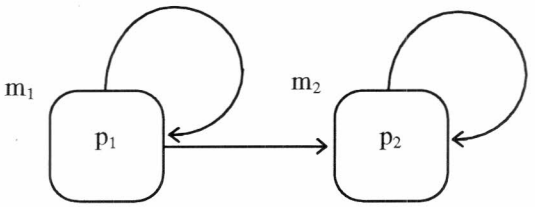

Figura N° 8

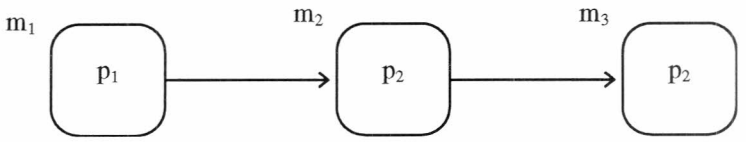

Figura N° 9

En la figura N° 7 tenemos un modelo en donde la valuación de un mundo hace verdadera a más de un símbolo de proposición. En la figura N° 8 del mundo m_1 del modelo sale un eje más que no es descripto por las fórmulas de Γ; y el modelo de la figura N° 9 aunque posea un mundo más, igual valida las fórmulas ya que para ellas no hay diferencia entre el loop del mundo m_2 del modelo de la figura N° 6 y el eje que une los mundos m_2 y m_3 del modelo de la figura N° 9, los cuales validan la misma proposición.

Por lo tanto, el conjunto de fórmulas al que habíamos llamado descripción no le alcanza a la lógica **KPI** para caracterizar en forma unívoca al modelo del diseño que queríamos verificar. Y de esta forma por ejemplo, la fórmula $\neg< >p_1$ que era validada por el modelo original en m_1 no podrá ser deducida sintácticamente pues los modelos de las figuras N° 7 y 8 la invalidan.

Pero como dijimos antes, los modelos con los que queremos trabajar poseen características que los modelos de las figuras N° 7, 8 y 9 no respetan. El modelo de la figura N° 7 no cumple con la condición que se impuso sobre los nombres de los módulos. Veremos más adelante cómo podemos indicar a la lógica que estos modelos no deben ser tenidos en cuenta.

Los modelos de las figuras N° 8 y 9, en cambio, nos muestran un problema diferente. En las descripciones que estuvimos realizando (tanto en castellano como en el lenguaje de la lógica modal) siempre hemos realizado una suposición más en forma implícita: que toda la información relevante estaba siendo mencionada. Si no decíamos

explícitamente que un determinado eje unía dos nodos, debía sobreentenderse que dicho eje no existía, pero como vemos en las figuras N° 8 y 9, la lógica **KPI** no realiza tal suposición. Será este problema de minimalidad el que impedirá la utilización del método sintáctico en la verificación de propiedades.

Veamos para terminar, la solución al problema de valuación y un poco más formalmente las complicaciones del problema de minimalidad.

Valuación

Debemos entonces forzar en el sistema lógico la condición que dice que en cada mundo un único símbolo proposicional es validado, para que de esta forma, modelos como el de la figura N° 7 no sean tenidos en cuenta.

Esta condición no puede introducirse axiomáticamente a una lógica modal clásica en el caso general cuando el lenguaje posee infinitos símbolos de proposición, pero sí cuando el lenguaje es finito. Notar que aunque la restricción a lenguaje finito no es problemática desde el punto de vista de la aplicación dado que todos nuestros modelos son finitos, sí posee un impacto considerable desde el punto de vista lógico.

(Si nos permitimos salir de los operadores modales más clásicos, podemos hallar lógicas que incorporan símbolos *nominales*. Estos nuevos símbolos atómicos poseen la propiedad de ser verdaderos exactamente en un punto del modelo y fueron introducidos por [Blackburn, 1993]. En su trabajo se demuestra que la lógica modal de <> es mucho menos expresiva que la extensión obtenida agregando nominales. Esta referencia fue obtenida del libro Extending Modal Logic [de Rijke, 1993]).

Una vez que tenemos finitos símbolos de proposición, podemos dar una axiomatización de esta condición para cada lenguaje $\mathcal{L}_n$ (donde $\mathcal{L}_n$ es el lenguaje original $\mathcal{L}$ restringido a sólo n símbolos de proposición).

El axioma que necesitamos es Vn. $\vee_{(i \leq n)} (p_i \wedge_{(j \leq n, j \neq i)} \neg p_j)$.

Este axioma indica justamente lo que queríamos: en todos los casos donde demostremos p_i para algún símbolo de proposición, podremos entonces derivar $\neg p_j$ para todo $j \neq i$. Y así (presuponiendo correctitud) los modelos del tipo de la figura N° 7 no son modelos válidos, pues, por ejemplo, en el mundo m_1 del modelo valen al mismo tiempo p_1 y p_2 de los cuales, por el axioma Vn, se derivan $\neg p_1$ y $\neg p_2$ dándonos una valuación inconsistente.

Cabe destacar algunas características respecto del axioma Vn. Primero, Vn es un axioma y no un axioma esquema, es decir, no consideramos toda instancia de sustitución de Vn, sino únicamente la fórmula Vn tal cual está escrita. Una regla de sustitución nos llevaría rápidamente a una lógica inconsistente, pues es fácil ver que, por ejemplo, la regla Sust. nos permite obtener $\{Vn\} \vdash (p_1 \wedge \neg p_1)$. Por otro lado, quizás sea esclarecedor mencionar la contrapartida semántica del axioma Vn. Desde la semántica, la incorporación de este axioma equivale a la extensión del conjunto de tautologías, o en otras palabras a la definición de una nueva relación de consecuencia que incluye a la noción clásica. Más exactamente, la nueva noción Cn' puede definirse como la mínima extensión a la noción clásica de consecuencia que verifica además la regla $Cn'(\varnothing) = Cn(\vee_{(i \leq n)} (p_i \wedge_{(j \leq n, j \neq i)} \neg p_j))$.

La extensión de la noción de consecuencia clásica a una noción más fuerte es habitual en el ámbito de las lógicas para computación. Por ejemplo, de esta forma se pueden reflejar las leyes físicas de gravedad en sistemas expertos que deben manipular objetos (como el sistema de control de un brazo robot).

Resumiendo, la nueva lógica propuesta que soluciona el problema de valuaciones es el sistema **KPIVn** axiomatizado por:

PL. φ, donde φ es una instancia de CP-tautología.
Vn. $\vee_{(i \leq n)} (p_i \wedge_{(j \leq n, j \neq i)} \neg p_j)$.
Ka. Toda instancia de $[a](\varphi \rightarrow \psi) \rightarrow ([a]\varphi \rightarrow [a]\psi)$, para toda $a \in L$.
Kai. Toda instancia de $[a]_i(\varphi \rightarrow \psi) \rightarrow ([a]_i\varphi \rightarrow [a]_i\psi)$, para toda $a \in L$.

It1. Toda instancia de $\varphi \rightarrow [a]\langle a\rangle_i\varphi$, para toda $a \in L$.

It2. Toda instancia de $\varphi \rightarrow [a]_i\langle a\rangle\varphi$, para toda $a \in L$.

MP. De $\vdash_{KPIVn} \varphi \rightarrow \psi$ y $\vdash_{KPIVn} \varphi$, obtener $\vdash_{KPIVn} \psi$.

Na. De $\vdash_{KPIVn} \varphi$, obtener $\vdash_{KPIVn} [a]\varphi$, para toda $a \in L$.

Nai. De $\vdash_{KPIVn} \varphi$, obtener $\vdash_{KPIVn} [a]_i\varphi$, para toda $a \in L$.

Vamos a demostrar que este sistema es correcto y completo respecto de la clase de modelos que nos interesan.

Teorema: La lógica **KPIVn** es correcta y completa respecto de la clase de modelos $CVn = \{\langle W, \{-a\rightarrow \mid a \in L\}, V\rangle \mid W \neq \varnothing$ y para todo $m \in W$, existe p_i tal que $V(p_i, m) = 1$ y para todo p_j, con $i \neq j$, $V(p_j, m) = 0\}$.

dem:

Correctitud:

Para completar la demostración de correctitud, y dado que demostramos la correctitud de **KPI** en la clase de todos los modelos (y por lo tanto en la clase menor de todos los modelos de CVn) basta demostrar que Vn es validada en todo modelo de CVn.

Sea M un modelo cualquiera de CVn, sea m cualquiera en W, queremos ver que $M \models_m \bigvee_{(i \leq n)} (p_i \bigwedge_{(j \leq n, j \neq i)} \neg p_j)$.

Supongamos que no, es decir (por definición de $\models$)

$M \not\models_m \bigvee_{(i \leq n)} (p_i \bigwedge_{(j \leq n, j \neq i)} \neg p_j)$ sii

$M \models_m \neg\bigvee_{(i \leq n)} (p_i \bigwedge_{(j \leq n, j \neq i)} \neg p_j)$ sii

$M \models_m \bigvee_{(i \leq n, j \leq n, i \neq j)} (p_i \wedge p_j) \vee \bigwedge_{(i \leq n)} \neg p_i$.

Es decir, existen i, j distintos tal que $M \models_m (p_i \wedge p_j)$ o para todo i $M \models_m \neg p_i$ sii

$(V(p_i, m) = 1$ y $V(p_j, m) = 1)$ o $V(p_i, m) = 0$ para todo p_i, con lo que M no esta en CVn. *Absurdo.*

Completitud:

Usando el método del modelo canónico, Si Σ es el conjunto de todos los axiomas de la lógica, basta demostrar que el modelo canónico M_Σ construido a partir de Σ esta en la clase CVn.

Para que M_Σ esté en CVn, ya que por construcción $W_\Sigma \neq \varnothing$, basta mostrar que para todo $m \in W_\Sigma$, existe p_i, tal que $V_\Sigma(p_i, m) = 1$ y para todo p_j con $i \neq j$ tenemos $V_\Sigma(p_j, m) = 0$.

Supongamos que no, es decir, existe m tal que o bien 1) existen $i \neq j$ tal que $V_\Sigma(p_i, m) = 1$ $V_\Sigma(p_j, m) = 1$ o bien pasa que 2) $V_\Sigma(p_i, m) = 0)$ para todo i.

Caso 1)

Luego por definición de V_Σ, p_i está en m y p_j está en m.

Pero m es un conjunto maximal consistente respecto de Σ, luego $p_i \wedge p_j$ está en m.

Pero Vn está en m por ser un axioma y $\{Vn\} \vdash p_i \rightarrow \neg p_j$ y $\{Vn\} \vdash p_j \rightarrow \neg p_i$.

Luego en m tenemos $(p_i \wedge p_j) \wedge (\neg p_i \wedge \neg p_j)$, que es una instancia de contradicción, por lo que m no sería un conjunto maximal consistente. *Absurdo.*

Caso 2)

Por definición de V_Σ, $V_\Sigma(p_i, m) = 0$ sii $\neg p_i \in m$, luego $\bigwedge_{(i \leq n)}\neg p_i \in m$ (pues m es conjunto maximal consistente).

Pero es fácil ver que $\{\bigwedge_{(i \leq n)}\neg p_i\} \vdash \neg Vn$, por lo que $\neg Vn \in m$ y al mismo tiempo $Vn \in m$. *Absurdo.* □

De esta forma logramos introducir al sistema axiomático la restricción sobre las valuaciones. Pasemos ahora al problema de minimalidad.

Minimalidad

Trabajemos una vez más sobre el ejemplo para ver que características presenta el problema de minimalidad. Podríamos pensar que el verdadero problema está en la forma en que construimos la descripción. Al fin y al cabo no dimos ninguna justificación de que ésta era la forma en que una descripción debía realizarse, más que hacer desde el lenguaje lógico lo que hubiéramos hecho al dar una descripción del grafo en castellano. Sin embargo, es fácil

demostrar que la característica de minimalidad no puede ser capturada agregando más información (más fórmulas modales) a la descripción.

Probemos por ejemplo tomar a Γ como el conjunto que describe todos los posibles caminos del grafo (sin repetición de ejes). Este conjunto incorporará en general mucha mas información que el conjunto obtenido por el criterio de descripción original.

Ejemplo: Si tomamos el modelo de la figura N° 6, siguiendo el criterio de tomar todos los caminos sin repetición de ejes, obtenemos una nueva descripción.

$$\Gamma = \{ \quad p_1,$$
$$p_1 \wedge \Diamond p_2,$$
$$p_1 \wedge \Diamond(p_2 \wedge \Diamond p_2),$$
$$p_1 \wedge \Diamond(p_2 \wedge \Diamond ip_2)\}$$

Ahora el modelo de la figura N° 9 ya no responde a esta descripción (pues no satisface la última fórmula), pero el modelo de la figura N° 8 sigue validando todas las fórmulas de Γ. No debemos sin embargo engañarnos con esta "mejora". En el caso general, dada cualquier descripción (conjunto consistente finito de fórmulas), puede siempre construirse un modelo no minimal que la satisfaga. Si Γ es finito, describe el grafo tal como este se ve desde el mundo inicial hasta una distancia d fija del mundo inicial. Todo modelo que respete el modelo original dentro del "radio" d satisfacerá Γ, sin importar lo que suceda a más de d pasos. Notar que es importante, para la argumentación anterior que el conjunto Γ sea finito, pero no hay forma de abandonar esta restricción pues manejar conjuntos infinitos hace intratable el método.

La última pregunta que queda abierta en este trabajo es si no hay una forma de utilizar el tipo de descripciones que inicialmente propusimos y encontrar alguna forma de incorporar la condición de minimalidad a la lógica. Parece bastante seguro que ningún axioma puede obtener este resultado (a diferencia del problema de valuaciones) y que deben utilizarse otros métodos, como por ejemplo utilizar una relación de demostración que no sea la standard (o semánticamente, una noción de consecuencia diferente pero no simplemente con un conjunto de tautologías más amplio). Las pistas para la solución quizás puedan derivarse del hecho que el modelo descripto cumple justamente con la propiedad de ser el mínimo modelo que satisface la descripción y por lo tanto es *inicial* en su clase. El problema es justamente como traducir la noción categórica de inicialidad a la lógica modal. Existen trabajos que vinculan teoría de categorías y lógica temporal como el de [Fiadeiro et al., 1991].

5 Ejemplos de Aplicación

Esta sección estará dedicada a la aplicación de los nuevos métodos formales de análisis de diseño discutidos en las secciones anteriores. Se desarrollarán tres ejemplos ordenados según su complejidad. El primer ejemplo muestra en forma genérica cómo detectar el uso incorrecto de la interface de un módulo. El segundo ejemplo muestra un diseño real de un módulo que calcula el CRC de un archivo en el cuál se detectan dos serios problemas de diseño. Por último, en el tercer ejemplo tomado del área de Diseño Orientado a Objetos se muestra como pueden manipularse diseños con mayor información en los módulos sin modificar el formalismo propuesto en esta tesis.

5.1 Uso Correcto de Interface

Como se discutió ampliamente en la sección 2, una de las nociones centrales del buen diseño es el concepto de Information Hiding para la obtención de una correcta modularización, que en muchos casos se ve planteado como el uso estricto de interfaces para el acceso a módulos. Las interfaces permitirán independizar la implementación de la funcionalidad de un módulo de su utilización, asegurando un menor acoplamiento, una mayor reusabilidad y principalmente facilitando la modificación posterior del módulo ante cambios del sistema.

Podemos pensar entonces a una librería dada, como un conjunto de submódulos que son "administrados" mediante un módulo de interface que atiende los requerimientos externos y los traduce a requerimientos para los diferentes submódulos internos. La figura N° 10 muestra como podríamos representar el uso correcto de la interface de un módulo.

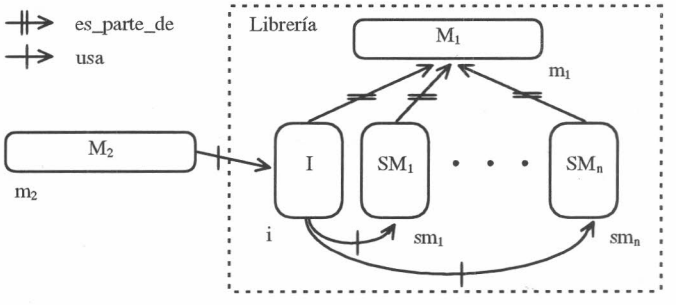

Figura N° 10: m_2 usa correctamente la interface del módulo m_1

Supongamos ahora un error de diseño, donde no se respeta la estructura de la figura N° 10, es decir, el módulo m_2 accede directamente al submódulo sm_1.

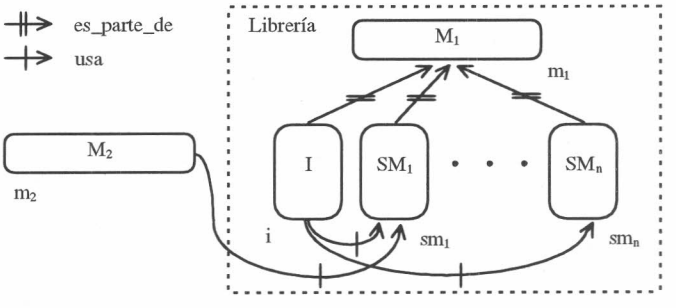

Figura N° 11: m_2 accede directamente al submódulo sm_1 sin respetar la interface del módulo m_1

Comparando las figuras N° 10 y 11, es fácil enunciar una fórmula que represente justamente el uso correcto de la interface. Parémonos en m_2 y veamos que m_2 hace uso incorrecto de m_1 si accede a una parte de m_1 que no es su interface, es decir que es obligación que cada vez que se accede a una parte de m_1 ésta sea la interface:

$[/](<//>M_1 \rightarrow I)$ (que puede leerse como M_2 *usa bien la librería* M_1). Es decir que probar (por ejemplo, utilizando el algoritmo de model checking) $M \models_{m2} [/](<//>M_1 \rightarrow I)$ demuestra que m_2 hace uso correcto de m_1.

Esto mismo puede ser verificado desde m_1. El módulo m_1 es usado correctamente si vale en todos sus submódulos que, o bien son la interface o bien son invocados sólo por ella: $[//]_i(I \vee [/]_i I)$ (que puede leerse como *la librería* M_1 *esta siendo bien usada*). Por supuesto, esta fórmula debe ser validada por m_1, no por m_2.

Observemos que la existencia de dos fórmulas para chequear la propiedad concuerda con el método usual de chequear el sistema cuando se incluye una librería y rechequear cada vez que un nuevo módulo hace uso de la misma.

En esta caso, la "estructura" de las fórmulas dadas están caracterizando la propiedad. Es decir, reemplazar los símbolos proposicionales I y M1 por los que resulten adecuados en cada momento no altera el significado de la fórmula.

5.2 Rutina de CRC

En este ejemplo discutiremos los problemas del diseño correspondiente a una rutina para verificar el código de CRC de un archivo. El módulo Verif_CRC contiene las rutinas para calcular el código de CRC y luego notificar si se hallaron errores. Para obtener el código de CRC el módulo Computa_CRC usa rutinas de una librería matemática. Estas rutinas acceden al archivo utilizando rutinas de una librería de E/S. Si existen problemas durante la entrada/salida, estos se reportan utilizando la rutina de Muestra_Mens_de_Error.

En el diseño se utilizan tres relaciones entre módulos, dos básicas que ya fueron comentadas en la sección 2 (usa y es_parte_de) y una relación compleja *mod_usa_modPpal* de uso entre módulos y módulos principales (aquellos que contienen funciones).

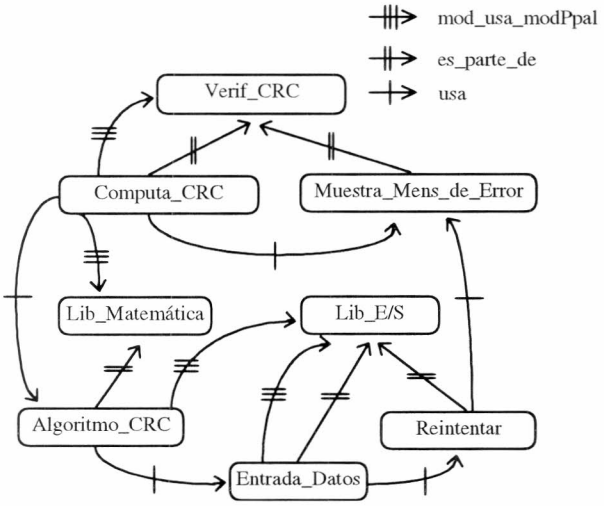

Figura N° 12: Modelo del Diseño de la Rutina de CRC

Como primer paso, ya que la relación mod_usa_modPpal es compleja podemos intentar reconstruirla en base a las relaciones básicas, chequeando de esta forma su correcta definición. Veamos que intuitivamente un módulo A mod_usa_modPpal un módulo B si A usa un módulo que es una parte de B. Es decir que la relación mod_usa_modPpal es la composición de las relaciones usa y es_parte_de y puede ser entonces definida mediante la fórmula $<>\varphi \leftrightarrow </><//>\varphi$.

Utilizando ahora el método de definición podemos obtener el modelo M^+ donde la relación que gobierna <> es explícita. En este nuevo modelo podemos chequear si $M^+ \models <>_T \leftrightarrow <///>_T$; si esta equivalencia no se verifica, la relación mod_usa_modPpal estaba mal definida en el modelo original.

Para visualizar gráficamente este error, podemos hallar $(M^+)^*$, donde todas las relaciones salvo las que nos ocupan fueron eliminadas. Según vimos en la sección de Abstracción de Modelos, basta tomar $\Gamma = \{<>_\top, <///>_\top\} \cup Mod$, donde Mod es el conjunto de proposiciones que nombran los módulos del modelo. La figura N° 13 muestra el modelo original y el resultado luego de la filtración del modelo expandido. Notar que en el modelo filtrado puede verse inmediatamente donde se produce el problema (la flecha entre Reintentar y Verif_CRC se había omitido en el modelo original).

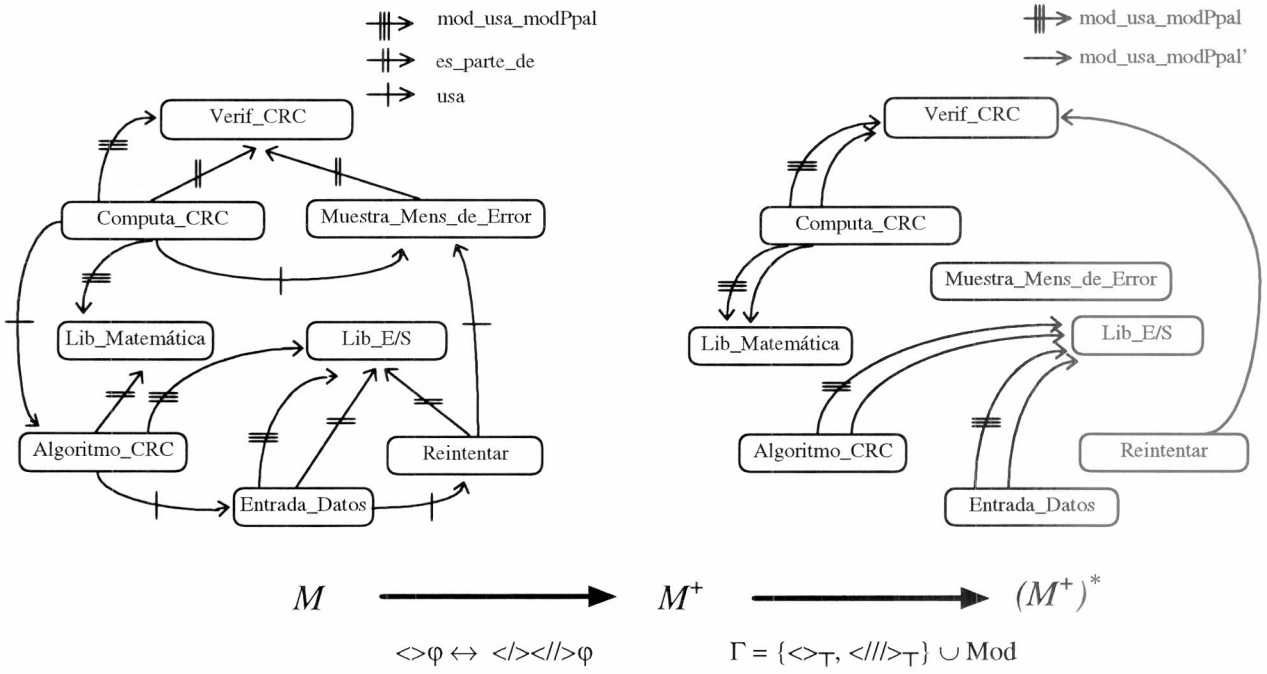

Figura N° 13: Chequeo de correctitud de la relación mod_usa_modPpal

En la figura N° 14 hemos corregido el primer error del diseño original.

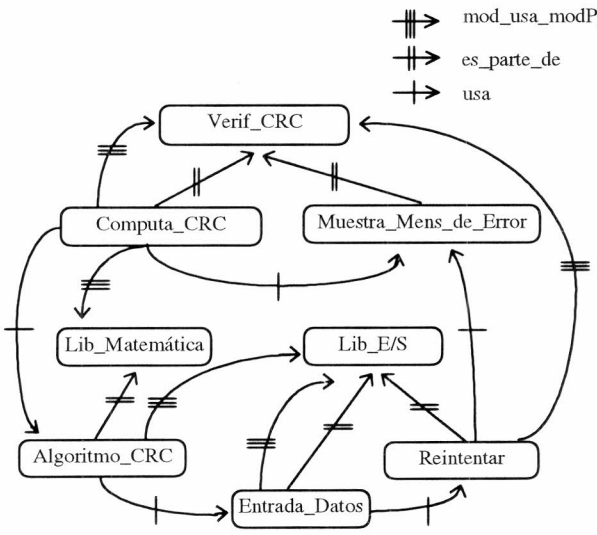

Figura N° 14: Modelo corregido

Quizás la omisión que tratamos recién sea sólo eso. Pero si dedicamos mayor atención al problema vemos que esta omisión estaba ocultando un problema de diseño mucho más profundo. Por un lado se intentaba respetar en la relación mod_usa_modPpal el orden parcial entre módulos que define la jerarquía de uso, y al mismo tiempo, la librería de E/S quería "sacar ventajas" al reutilizar la rutina de Muestra_Mens_de_Error. El par (Reintentar, Verif_CRC) en la relación mod_usa_modPpal nos muestra más claramente los problemas en la jerarquía.

Armemos la jerarquía de uso entre módulos principales utilizando $-///\rightarrow$ en la definición original (figura N° 12). Tomemos un módulo de Verif_CRC como Computa_CRC, puesto que éste está en relación mod_usa_modPpal con Lib_Matemática tenemos Verif_CRC < Lib_Matemática. Ahora, dado que Algoritmo_CRC $-///\rightarrow$ Lib_E/S, tenemos Lib_Matemática < Lib_E/S. Y por $-///\rightarrow$ ningún otra par se agrega al orden.

En el modelo corregido de la figura N° 14 sin embargo Reintentar $-///\rightarrow$ Verif_CRC hace que Lib_E/S < Verif_CRC produciendo un ciclo en la jerarquía.

Notar que la construcción de este ciclo lleva aparejado un análisis de mayor nivel de abstracción que un simple testeo de las relaciones básicas en el modelo. Es más la relación de uso explícita ($-//\rightarrow$) sí forma un orden parcial entre módulos.

Desarrollemos formalmente este problema y tratemos de hallar su solución.

Nos interesa definir la relación usa_modPpal, de uso entre módulos principales (según fue definida en la sección 2). Recordemos que un módulo principal X usa_modPpal otro módulo principal Y (distinto de si mismo), si existe una parte A que es_parte_de X y una parte B que es_parte_de Y tal que A usa B. Puesto que ya hemos corregido la relación mod_usa_modPpal podemos utilizarla para definir usa_modPpal como $<>\varphi \wedge p \leftrightarrow </>_i<///>(\varphi \wedge \neg p) \wedge p$. La fórmula anterior (el ejemplo más complejo de definición visto hasta el momento) define $\longrightarrow$ como la composición de $(-/\rightarrow)^{-1}$ y $-///\rightarrow$ menos la relación diagonal en W (el uso de p asegura esta última resta).

Siguiendo lo presentado en la sección Expansión de Modelos debemos demostrar que el modelo expandido (en la clase de modelos correspondientes a grafos de diseño) satisface la equivalencia:

Proposición: Sea $M = \{W, \{-a\rightarrow \mid a \in L\}\}$ un modelo que corresponda a un grafo de diseño tal que a, b ∈ L entonces $M^+ = <W, \{-a\rightarrow \mid a \in L\} \cup \{-c\rightarrow \mid c \notin L$ y m–c→m' sii existe m'' tal que m←a–m''–b→m' y $m \neq m'\}$, V> es una L∪{c}-expansión de M que satisface que para toda $\varphi \in \mathcal{L}_{L\cup\{c\}}$, p ∈ IP, $M^+ \models_m <c>\varphi \wedge p \leftrightarrow <a>_i<b>(\varphi \wedge \neg p) \wedge p$.

dem:

Por construcción M^+ es una L∪{c}-expansión de M.

Sean ahora $\varphi \in \mathcal{L}_{L\cup\{c\}}$, p ∈ IP, m cualquiera en W,

supongamos que $M^+ \models_m <a>_i<b>(\varphi \wedge \neg p) \wedge p$

sii $M^+ \models_m <a>_i<b>(\varphi \wedge \neg p)$ y $M^+ \models_m p$

sii existe m'', existe m' tales que m←a–m''–b→m' y $M^+ \models_{m'} (\varphi \wedge \neg p)$ y $M^+ \models_m p$

sii existe m'', existe m' tales que m←a–m''–b→m' y ($M^+ \models_{m'} \varphi$ y $M^+ \models_{m'} \neg p$) y $M^+ \models_m p$

(ahora, $M^+ \models_{m'} \neg p$ y $M \models_m p$ nos asegura $m \neq m'$ y podemos invertir los existenciales)

sii existe m' tal que m–c→m' y $M^+ \models_{m'} \varphi$ y $M^+ \models_{m'} \neg p$ y $M^+ \models_m p$

(el sentido ⇒ es directo, ⇐ se obtiene por las características del modelo y la definición de –c→, ya que $M^+ \models_{m'} p \wedge <c>\varphi$ implica $M^+ \models_{m'} p \wedge <c>(\varphi \wedge \neg p)$).

sii $M^+ \models_m <c>\varphi \wedge p$ □

Notar que nos interesará que la relación usa_modPpal sea transitiva, y como discutimos al presentar model checking esto puede obtenerse realizando las modificaciones adecuadas a dicho algoritmo.

El método de definición nos permite construir a partir del modelo M de la figura N° 15 (a la izquierda) el modelo M^+ donde la relación usa_modPpal es explícita (no mostrado en la figura). En este modelo podemos detectar ciclos utilizando el algoritmo de chequeo transitivo y verificando que al menos uno de los módulos principales m_i valida $p_i \wedge <>p_i$ donde p_i es la única proposición verdadera en m_i, ($Trans(M^+) \models_{mi} p_i \wedge <>p_i$), es decir que el módulo se utiliza a sí mismo al considerar a la relación usa_modPpal como transitiva (selfloops).

La parte derecha de la figura N° 15 muestra la filtración del modelo M^+ aislando la relación que nos interesa.

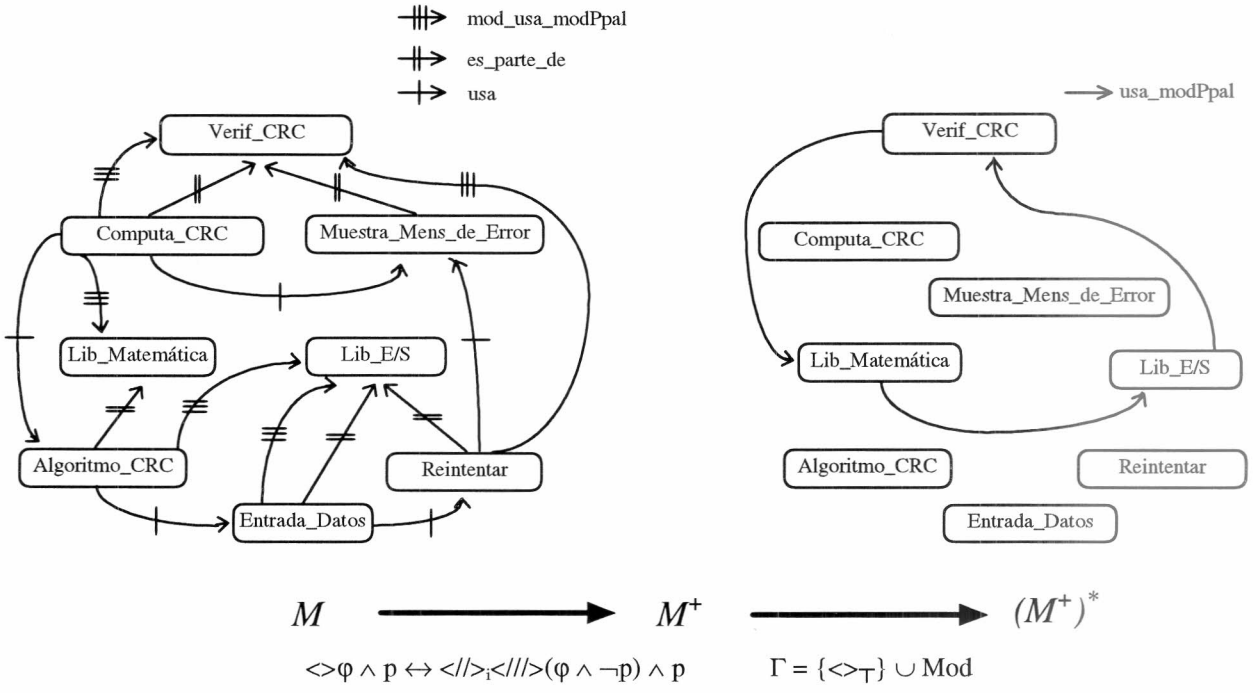

Figura N° 15: Chequeo de la jerarquía de componentes principales

Podemos utilizar la técnica clásica de sandwiching para romper el ciclo de la jerarquía. Como ya dijimos, el error puede localizarse en la relación Reintentar –///→ Verif_CRC que nace de la relación Reintentar –/→ Muestra_Mens_de_Error. Vemos ahora, que el módulo de mensajes de error no debería formar parte de Verif_CRC, y debe ser separado en una nueva librería de manejo de errores que podrá ser utilizada tanto por Verif_CRC como por Lib_E/S sin causar problemas en la jerarquía.

El modelo corregido puede verse en la figura N° 16.

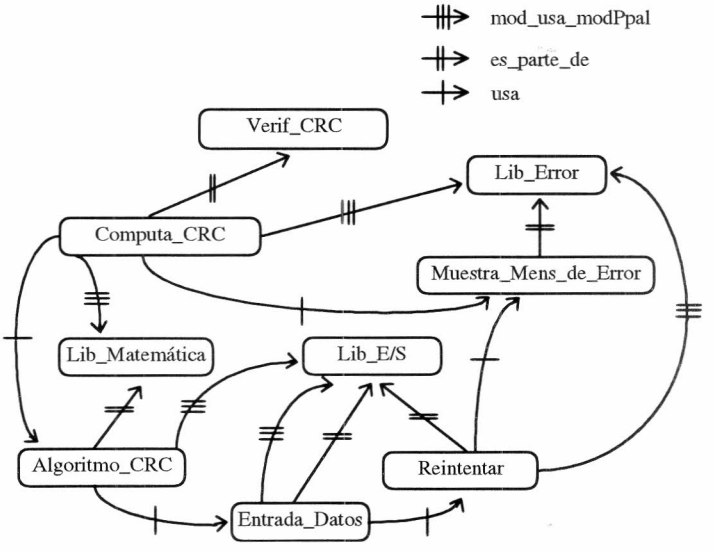

Figura N° 16: Diseño sin ciclo en la jerarquía

Notar que el ciclo en la jerarquía fue solucionado *sin modificar la relación usa entre módulos*, mostrando que el problema no se hallaba a nivel de uso entre módulos sino en la relación más abstracta entre módulos principales.

El ejemplo de ciclicidad en la jerarquía es clásico en la literatura, y suele utilizarse para medir las capacidades de una metodología de verificación de diseños. Por ejemplo en [Cosens et al., 1992] un diseño con problemas similares es presentado. Aunque las diferencias puntuales son varias (ya que el lenguaje utilizado es radicalmente distinto) el procedimiento de detección (selfloops) y solución (sandwiching) del problema es el mismo.

5.3 Diseño en el Paradigma de Objetos

Como un caso particular de la metodología de diseño formal dedicaremos esta sección al análisis del Paradigma de Objetos que posee en la actualidad un gran predominio.

Las reglas de buen diseño y el ejemplo (con mínimas modificaciones) que estudiaremos más abajo están citados en [Wirfs-Brock et al., 1990].

Veremos que según las técnicas propuestas por Wirfs-Brock et al. es usual que en los modelos de diseño orientado a objetos se posea mucha más información acerca de cada clase que la que estuvimos manejando hasta el momento, como por ejemplo el listado de las responsabilidades que la clase define, el tipo de clase (abstracta o concreta), etc.

Nuestro interés es mostrar como podemos manejar esta información extra en nuestro formalismo siguiendo la propuesta en [Bourdeau & Cheng, 1995] de transformar la información implícita dentro de la clase en explícita. Veremos esto en más detalle al desarrollar el ejemplo.

En la etapa de diseño en el ciclo de desarrollo de software orientado a objetos, se utilizan los grafos de jerarquía como herramienta para modelizar las relaciones de herencia entre clases relacionadas.

En este caso los módulos son clases y la relación entre estos es *es_subclase_de*, que define la herencia de superclase a subclase. Además, entre las clases se distingue entre abstractas (clases que no poseen instanciaciones, y se crean para factorizar propiedades en común) y concretas (clases que sí poseen instanciaciones, que serán los objetos activos durante la corrida). Cada clase define sus responsabilidades (actividades que es capaz de desempeñar y que caracterizan su comportamiento) las cuales, vía la relación jerárquica, se heredan de superclase a subclase. Las subclases sin embargo, son capaces de redefinir una responsabilidad heredada para adecuarla a su comportamiento particular.

Luego de haber determinado las clases del sistema y la jerarquía de herencia, se pueden analizar estas relaciones para identificar y resolver problemas en el diseño. El análisis se basa en los siguientes puntos:

- Modelar la jerarquía *es_subclase_de*.
- Factorizar las responsabilidades en común lo más alto posible en la jerarquía.
- Asegurar que las clases abstractas no hereden de clases concretas.
- Eliminar clases que no agregan funcionalidad.

Trabajaremos en base al ejemplo del Diseño de un Editor de Dibujos. En particular chequearemos el buen diseño de la jerarquía de la clase Elemento_de_Dibujo. El diseño de la clase se presenta en la siguiente figura (modificación de la figura 6-10 de [Wirfs-Brock et al., 1990]):

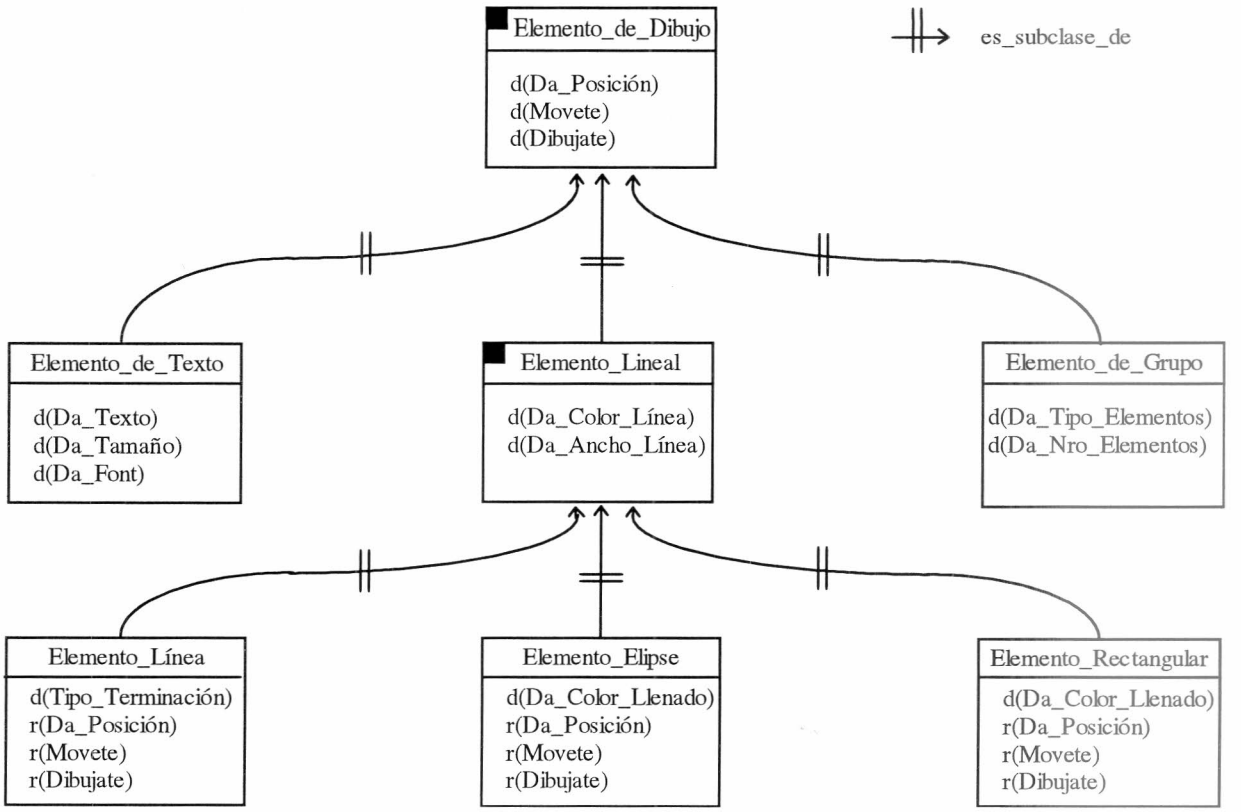

Figura N° 17: Diseño de la clase Elemento_de_Dibujo

Los recuadros representan las diferentes subclases y para cada una de ellas se especifican las responsabilidades que ellas definen (d) o redefinen (r), distinguiendo además las clases abstractas (marcadas en la esquina superior izquierda). Como dijimos anteriormente, las responsabilidades se heredan hacia abajo en la jerarquía, lo que nos permite especificar para cada subclase solamente las *nuevas* responsabilidades que la caracterizan.

A primera vista, el diseño parece contener mucha más información que la que manejamos en nuestros modelos de diseño. Sin embargo, mostraremos que es posible adaptar el diseño anterior a nuestro formalismo.

La flexibilidad que nos provee el uso de relaciones de accesibilidad nos permite incorporar toda la información adicional:

- La propiedad monádica es_clase_abstracta no es más que un subconjunto S del conjunto de clases, que podemos pensar como la relación diagonal en $S \times S$. En nuestro ejemplo S = {Elemento_de_Dibujo, Elemento_Lineal}, con lo que la relación sería $-/\rightarrow$ = {(Elemento_de_Dibujo, Elemento_de_Dibujo), (Elemento_Lineal, Elemento_Lineal)}.
- Las propiedades de definición (d) y redefinición (r) son en realidad propiedades diádicas que relacionan clases con responsabilidades y que por lo tanto pueden traducirse directamente como relaciones de accesibilidad ($-d\rightarrow$ y $-r\rightarrow$), una vez que hemos asignado un mundo a cada responsabilidad.

Veamos el resultado de este traducción:

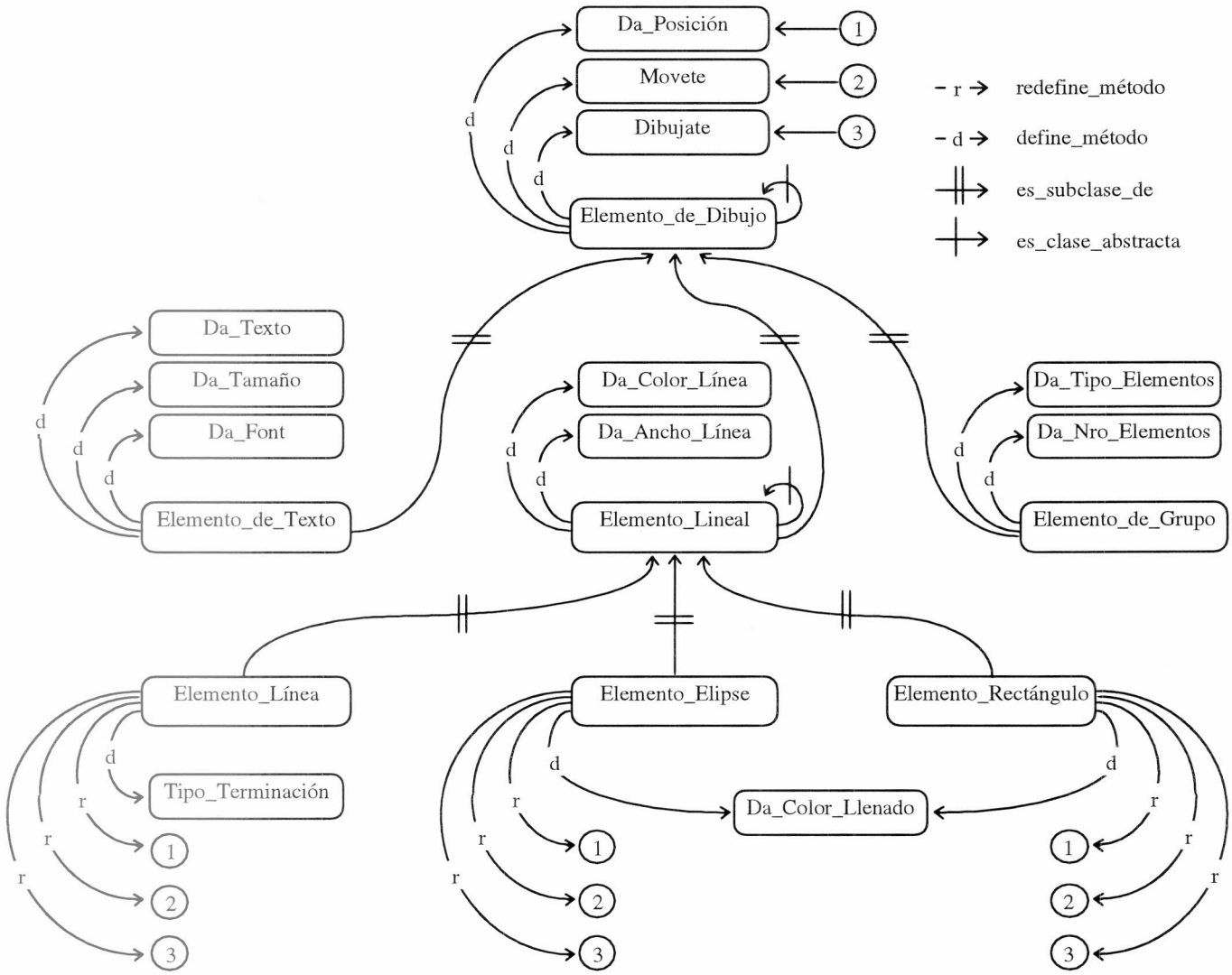

Figura N° 18: Modelo modal de la clase Elemento_de_Dibujo

Ahora podemos usar las herramientas presentadas en las secciones anteriores para modelizar y analizar cada uno de las reglas de buen diseño que deben tenerse en cuenta.

Modelar la jerarquía es_subclase_de: Una clase debería ser subclase de otra sólo si soporta todas las responsabilidades definidas por esta última. Otra forma de decir esto es que la herencia debe modelar a la relación *es_subclase_de*, donde cada clase debe ser un tipo específico de sus superclases. Modelizar la relación entre clases tiene dos propósitos: por un lado asegurar que ésta sea una jerarquía, y por otro ayudar a la verificación de que las subclases soporten por lo menos todas las responsabilidades de sus superclases. Esto incrementa su reusabilidad dado que es más fácil de ver donde puede ser insertada una nueva clase en una jerarquía ya existente.

Dada la forma en que modelamos las clases, la herencia de responsabilidades de clase a subclase es inmediata. Por definición, las responsabilidades de una clase están dadas por las responsabilidades definidas explícitamente (<d>p_i) más las definidas por sus superclases (<//><d>p_j, con –//→ transitiva), por lo que parte del chequeo a realizar en este punto es trivial.

El chequeo de la jerarquía es similar al realizado en el ejemplo anterior, pero mucho más simple, dado que debe realizarse directamente sobre la relación explícita es_subclase_de.

Factorizar las responsabilidades en común lo más alto posible en la jerarquía: Si un conjunto de clases soportan una responsabilidad en común, ellas deberían heredar dicha responsabilidad de una superclase en común. Si esta superclase no existe, se deberá crear una, pasando la responsabilidad a la nueva clase. Esta nueva clase será probablemente una clase abstracta. El principal beneficio de diseñar la jerarquía con clases abstractas se evidencia cuando se desea agregar nueva funcionalidad a una aplicación ya existente.

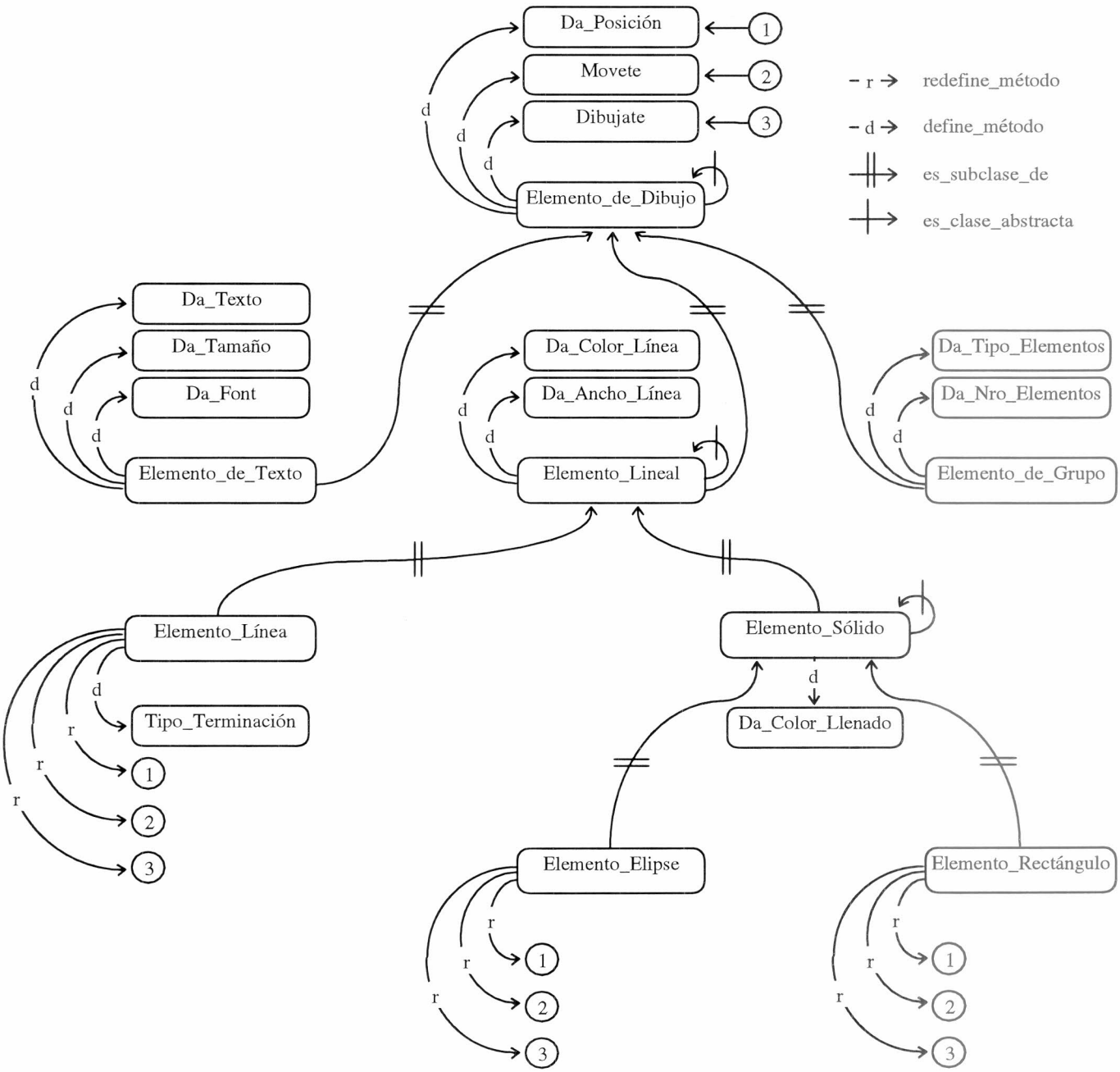

Figura N° 19: Factorización de la clase abstracta Elemento_Sólido

En nuestro ejemplo, el chequeo de correcta factorización puede realizarse en la siguiente forma:

Definamos un conjunto Γ de fórmulas como $\Gamma = \{(<d>p_i \wedge p_j) \rightarrow <//>[//]_i(<d>p_i \rightarrow p_j) \mid p_j$ son los nombres de las clases y p_i son las responsabilidades que corresponden a cada una de las clases $p_j\}$. Cada una de las fórmulas en Γ, por ejemplo (<d>Da_Tamaño $\wedge$ Elemento_de_Texto) $\rightarrow$ <//>[//]$_i$(<d>Da_Tamaño $\rightarrow$ Elemento_de_Texto) dice que si una clase (Elemento_de_Texto) define una responsabilidad (Da_Tamaño) entonces esta responsabilidad no se define en ningún otro módulo que sea parte de la jerarquía (en la superclase de Elemento_de_Texto es necesario que la definición de Da_Tamaño se realice en la clase Elemento_de_Texto). Si el modelo valida todas las fórmulas en Γ, podemos asegurar una buena factorización. Por otro lado, las fórmulas que no sean validadas nos indicarán casos de error.

En la figura N° 18 podemos chequear que las clases Elemento_Elipse y Elemento_Rectángulo comparten la responsabilidad Da_Color_Llenado que deberá por lo tanto ser factorizada en una nueva clase abstracta Elemento_Sólido.

Asegurar que clases abstractas no hereden de clases concretas: Por su naturaleza, las clases abstractas soportan sus responsabilidades en forma independiente de la implementación. Por lo tanto nunca deben heredar de clases concretas, las cuales pueden depender específicamente de la implementación. Si un diseño tiene esa clase de herencia, el problema puede ser resuelto creando otra clase abstracta de la cual tanto la clase abstracta como la concreta pueden heredar su comportamiento en común.

En el lenguaje modal la propiedad a chequear dice que es necesario para toda clase abstracta que sus superclases sean también clases abstractas: $</>_\top \rightarrow [//]</>_\top$ con la relación $-//\rightarrow$ transitiva. El algoritmo de model checking nos permitirá verificar que esta propiedad es validada por el modelo de nuestro ejemplo.

Eliminar clases que no agregan funcionalidad: Las clases que no tengan responsabilidades deben ser eliminadas. Si una clase hereda una responsabilidad que va a implementar de manera única, entonces agrega funcionalidad a pesar de no tener responsabilidades propias, y no debe ser eliminada. Por el contrario, clases abstractas que no definen o redefinen responsabilidades no tiene uso y deben ser eliminadas, ya que no incrementarán la reusabilidad.

La regla anterior puede también enunciarse como la obligación de toda clase de compartir parte del conocimiento del objeto que se está definiendo. O en otras palabras, que toda clase debe colaborar en la definición o redefinición de las responsabilidades que caracterizarán a la jerarquía de objetos.

Esta propiedad es simple de enunciar como $<d>_\top \vee <r>_\top$, que debe ser validada por el modelo. Las clases que no cumplan esta propiedad deberán ser eliminadas.

Notar que la propiedad anterior permite a clases abstractas la redefinición de responsabilidades. No es erróneo que una clase abstracta posea código de implementación a pesar de que usualmente definen sólo las interfaces de las responsabilidades para sus subclases. Sin embargo, el diseñador debe cuidar que el código incorporado a una clase abstracta no dañe su condición de tal.

Puede por ello ser útil detectar cuales son las clases que incorporen código de implementación. Esto puede realizarse chequeando la fórmula $</>_\top \wedge <r>_\top$. Las clases que validen la fórmula anterior deben ser revisadas por posibles problemas.

La aplicación de nuestros métodos al ejemplo de diseño orientado a objetos, muestra que la restricción sobre las valuaciones impuesta a los modelos de diseño no limita en forma excesiva al formalismo. Es más, creemos que los modelos restringidos son más simples de manipular (a pesar de ser de mayor tamaño que los modelos correspondientes a grafos de jerarquías usuales) y el uso de relaciones explícitas tiene en nuestro caso mayor relevancia, dado que nuestro lenguaje de descripción contiene operadores modales que serán su contrapartida en la sintaxis.

6 Trabajo Futuro

La presente tesis abre varias posibilidades de trabajo futuro, tanto puramente teóricas y de importancia para la lógica modal, como aplicadas al área de Ingeniería de Software:

La Teoría de Definición Modal que se comenzó a describir en esta tesis, debe completarse con un análisis de la interrelación que existe entre el poder expresivo del lenguaje y su capacidad para caracterizar distintas operaciones sobre relaciones. Sería también importante hallar un caso general de *definición modal* que evite trabajar con casos particulares como hemos hecho. Los casos de definiciones parciales que hemos mostrado dan origen a la pregunta de cuál es exactamente el poder expresivo que el lenguaje extendido posee.

Quedan por supuesto abiertas las preguntas que ya hemos planteado durante el análisis sintáctico. Por un lado estudiar cómo imfluye el uso de una lógica modal con nominales en la caracterización unívoca de un modelo de diseño (problema de valuaciones). Por otro lado, como vincular lógica modal y categorías para capturar la restricción de minimalidad en la descripción de los modelos (problema de minimalidad).

Puede realizarse un trabajo estrictamente relacionado con el área de complejidad y optimización, implementando algoritmos superiores a los propuestos. Una mejora del prototipo presentado en el Apéndice mediante la introducción de una interfase gráfica también sería deseable.

Los ejemplos de aplicación que se estudiaron en la tesis fueron elegidos para demostrar el poder de las herramientas propuestas, por ser casos conocidos en la literatura. Creemos que no constituyen ejemplos triviales, pero de todas maneras sería interesante aplicar nuestras técnicas en más y más variados casos. Esto permitirá encontrar nuevas propiedades a verficiar e investigar si las técnicas son suficientes para manejarlas. Notar que la búsqueda de ejemplos de aplicación no necesariamente debe estar restringida al área de diseño. Areas como la ya nombrada de Administración de la Configuración de un Sistema parecen adaptarse perfectamente a nuestro formalismo.

Como un punto aparte, debe estudiarse el verdadero poder de abstracción del método de filtración. Los resultados exhibidos son alentadores y deberían ser examinados en profundidad.

El último ejemplo muestra un caso en donde no es necesario modificar la metodología propuesta para manejar modelos con mayor información en los módulos. Esto no quiere decir sin embargo que esta transformación pueda ser llevada a cabo en todos los casos. Podríamos pensar en varios ejemplos donde los modelos contienen mayor información como una especificación de su comportamiento, datos acerca de restricciones de su implementación, etc. En estos casos varias de nuestras propuestas deberán ser adaptadas.

Muy relacionada al área de Diseño de Software se encuentra el área de Arquitectura. Es posible que el presente trabajo sea aplicable también a la descripción de Arquitecturas de Software. En este área sin embargo, el problema tiene mayor complejidad. Los grafos de representación utilizados parecen capturar correctamente los componentes estáticos del sistema, pero no así los dinámicos que definen la interacción entre los componentes [Garlan & Shaw, 1993; Allen & Garlan, 1994]. En estos grafos, las relaciones cobran una importancia fundamental. Por ejemplo, del hecho que dos componentes están en relación cliente-servidor, podemos inferir el comportamiento de ambos componentes y la forma en que se comunican (el cliente realiza pedidos al servidor en cualquier momento, el servidor responde a pedidos sin conocer quién lo solicita, etc.). Existe trabajo actual en esta área, por ejemplo en [Dean & Cordy, 1995] se describe un lenguaje sintáctico no modal para la descripción de arquitecturas.

7 Conclusiones

En este trabajo presentamos una metodología para la representación de diseños de sistemas de software y para la verificación de propiedades sobre éstos. Los grafos de diseño son considerados modelos de una lógica modal extendida; y los métodos usados para verificar propiedades se desarrollan a partir de técnicas típicas de la lógica modal clásica, extendida para cubrir las diferencias en el formalismo modificado. Presentamos los procedimientos o herramientas lógicas para derivar el modelo modal asociado a todo diseño, el algoritmo de chequeo de propiedades, la técnica de definición de nuevas relaciones y el método de filtración de modelos. Estos métodos son demostrados en tres ejemplos de aplicación de diverso tipo.

Otra vez, dadas las características del trabajo podemos ofrecer conclusiones desde dos puntos de vista: el trabajo estrictamente modal y el trabajo de aplicación al tema específico de verificación de propiedades en la etapa de diseño.

En el primer caso, el trabajo contiene algunos resultados de importancia. Se presenta una lógica polimodal con operadores inversos (**KPI**) con el fin de utilizarla como lenguaje de especificación de propiedades a chequear mediante un algoritmo de model checking. Los lenguajes de verificación de programas existentes no requieren de operadores inversos (aunque en ciertos casos se utilizan operadores de cuantificación sobre ramas del modelo) dado que trabajan sobre la corrida del programa y en dichos modelos los operadores de futuro alcanzan para la descripción del progreso del sistema. En los diseños (que por convención son conexos), los operadores inversos permiten el acceso a todo otro módulo desde cualquier punto.

La lógica **KPI** es en verdad una familia de lógicas, cada una de las cuales está caracterizada por el sort de labels que define su alfabeto. Se dedicó especial atención a los problemas que se presentan al manejar modelos de diferente signatura. En la literatura se asume usualmente un sort de labes fijo al cual responden todos los modelos de la clase.

En relación con este punto, la definición de filtraciones es ahora un morfismo entre modelos de distinto sort (reducción del conjunto de labels) y la técnica de definición incorpora el concepto de L-expansión de un modelo (extensión del conjunto de labels). Los resultados referentes a la Teoría de Definición Modal presentados en este trabajo son sólo las bases necesarias para la aplicación buscada. Es interesante el hecho que no se hallan podido encontrar referencias a un trabajo similar, lo que anima a realizar una investigación lógica más profunda del tema.

Finalmente, se analizaron las limitaciones de la lógica **KPI** para la demostración sintáctica de propiedades de diseño. Estas limitación no son subsanadas en la restricción finita **KPIVn** (e hipotéticamente tampoco serían solucionadas agregando nominales aunque se obtendría una lógica no finita) debido al problema de minimalidad en la caracterización de modelos de diseño.

El trabajo en el área de Ingeniería de Software puede resumirse en los siguientes puntos:

A partir de la similitud entre los modelos utilizados en dos áreas distintas, se definió una lógica específica para el uso en la verificación y descripción de diseños. Luego se procedió a recopilar de la literatura nociones básicas de buen diseño aceptadas en forma general y se intentó su traducción a nociones modales que pudieran ser chequeadas automáticamente sobre los modelos de diseño.

El resultado obtenido fue un lenguaje formal de especificación de propiedades, un algoritmo de chequeo automático y lo más importante, el diseño de un conjunto de herramientas que permiten manipular los modelos según los deseos del diseñador.

Este conjunto de herramientas es en verdad una adaptación de métodos tradicionales de la lógica (modal y clásica). Cabe destacar la forma en que estos métodos una vez extendidos y adaptados a la aplicación, permiten realizar las tareas que usualmente se encuentran en el proceso de Ingeniería de Software. Específicamente, el uso de filtraciones para la obtención de abstracciones del diseño y el uso de la teoría de definición modal para la definición de relaciones derivadas.

Como muestran los ejemplos de aplicación, los métodos propuestos parecen ser efectivos. Es importante además la facilidad con que los mismos pueden ser implementados, como lo muestra el Apéndice.

Apéndice

Se incluye a continuación el código de una implementación prototipo de las ideas expuestas en esta tesis. La implementación fue realizada en el lenguaje funcional Standard ML of New Jersey [AT&T Bell Laboratories, 1993; Ullman, 1994], en su versión para Macintosh.

La aplicación se dividió en 8 módulos, la mayoria de ellos fueron implementados mediante TADs:

- *Utilidades en Listas*. Funciones específicas sobre listas utilizadas por los otros módulos.
- *Conjunto*. Tipo polimórfico implementado sobre listas, haciendo irrelevante el orden.
- *Fórmula*. Implementa la sintaxis de la lógica KPI: constructores de fórmulas y funciones como subfórmulas y tamaño (o complejidad) de una fórmula.
- *Relación*. Implementa dos tipos: Relación y Conjunto de Relaciones. Una relación es un conjunto de pares sobre un determinado dominio, más un nombre que la identifica.
- *Valuación*. Permite construir funciones de valuación que toman un par (mundo, proposición) y retorna un valor booleano.
- *Modelo*. Implementa el tipo Modelo de Kripke para la lógica KPI, define además la noción semántica de validez en un mundo, validez en un modelo y satisfacibilidad en un modelo.
- *KPI*. Implementa los algoritmos de Model Checking y Filtración de Modelos. Utilizando el algoritmo de model checking se redefinen las semánticas dadas en el módulo *Modelo*.
- *Ejemplo Kwic*. Para probar los algoritmos implementados, se construye en el nuevo lenguaje, el diseño del sistema Kwic, y se repiten los ejemplos que se enuncian en la tesis.

Finalmente, luego del Ejemplo Kwic se muestra la salida obtenida al ejecutar los dos algoritmos principales de la implementación: Model Checking y Filtración. La salida fue ligeramente formateada para una mayor legibilidad.

Nota: Debemos advertir que la implementación fue realizada en un muy corto tiempo y se apuntó más a la simpleza y rapidez que a preveer futuras expansiones y modificaciones. El objetivo principal fue mostrar una versión ejecutable de las propuestas introducidas en la tesis.

```
(*****************************)
(**** LIBRERIA DE LISTAS ****)
(*****************************)

signature SIG_LIST_LIB =
   sig

   (* Retorna la lista de los elementos que satisfacen la condicion *)
   val find : ('a -> bool) -> 'a list -> 'a list

   (* Retorna .AND.(ai) para ai en la lista de booleanos *)
   val andlist : bool list -> bool

   (* Retorna .OR.(ai) para ai en la lista de booleanos *)
   val orlist : bool list -> bool

   (* Inserta en orden un elemento a una lista ordenada, o lo elimina si ya
      esta presente, el primer parametro debe ser la funcion "mayor o igual" *)
   val insord : (''a * ''a -> bool) -> ''a list -> ''a -> ''a list

   (* Retorna la lista ordenada, eliminando duplicados, el primer parametro
      debe ser la funcion "mayor o igual" *)
   val lista_ord : (''a * ''a -> bool) -> ''a list -> ''a list

   (* Imprime una lista de enteros *)
   val print_list_int : int list -> unit

   (* retorna un elemento de la lista que satisface la condicion *)
   val findone : ('a -> bool) * 'a list -> 'a

   end;

structure str_list_lib : SIG_LIST_LIB =
   struct

   (* val find : ('a -> bool) -> 'a list -> 'a list *)
   fun find pred [] = [] |
       find pred (a::rest) = if pred a then a::(find pred rest) else (find pred rest);

   (* val andlist : bool list -> bool *)
   fun andlist(S) = fold (fn(x, y) => (x andalso y)) (S) true;

   (* val orlist : bool list -> bool *)
   fun orlist(S) = fold (fn(x, y) => (x orelse y)) (S) false;

   (* val lista_ord : (''a * ''a -> bool) -> ''a list -> ''a list *)
   fun insord pr [] a = [a] |
       insord pr (a::S) b = if (a = b) then
                               (a::S)
                            else
                               if pr(a,b) then
                                  b::(a::S)
                               else
                                  a::(insord pr S b);

   (* val lista_ord : (''a * ''a -> bool) -> ''a list -> ''a list *)
   fun lista_ord pr [] = [] |
       lista_ord pr (a::S) = insord pr (lista_ord pr S) a;

   (* val print_list_int : int list -> unit *)
   fun print_list_int([]) = print "" |
       print_list_int(s::ls : int list) = (print " ";
                                           print s;
                                           print_list_int(ls));

   (* val findone : ('a -> bool) * 'a list -> 'a *)
   exception ERROR_FINDONE;

   fun findone(f, []) = raise ERROR_FINDONE |
       findone(f, a::l) = if (f(a)) then
                             a
                          else
                             findone(f, l);

   end;
open str_list_lib;
```

```
(*******************)
(**** CONJUNTOS ****)
(*******************)

infix 6 U_;
infix 6 dif_;
infix 5 en_;

signature SIG_CONJ =
   sig
      type ''a CONJ

   (* Retorna true si el conjunto es vacio, falso en otro caso *)
   val es_vacio : ''a CONJ -> bool

   (* Predicado de pertenencia, true si el elemento esta en el conjunto,
      falso en otro caso *)
   val en_ : ''a * ''a CONJ -> bool

   (* Retorna el par formado por un elemento cualquiera del conjunto y el
      conjunto sin dicho elemento *)
   val choice : ''a CONJ -> ''a * ''a CONJ

   (* Retorna el conjunto formado por los elementos de ambos conjuntos *)
   val U_ : ''a CONJ * ''a CONJ -> ''a CONJ

   (* Elimina un elemento dado de un conjunto *)
   val sacar_elem : ''a CONJ * ''a -> ''a CONJ

   (* Retorna el conjunto formado por los elementos del primer conjunto que
      no estan en el segundo conjunto *)
   val dif_ : ''a CONJ * ''a CONJ -> ''a CONJ

   (* Transforma una lista en un conjunto (eliminando por ejemplo elementos repetidos) *)
   val uniques : ''a list -> ''a CONJ

   end;

structure str_conj : SIG_CONJ =
   struct
      type ''a CONJ = ''a list;

(* val es_vacio : ''a CONJ -> bool *)
fun es_vacio(S) = null(S);

(* val en_ : ''a * ''a CONJ -> bool *)
fun x en_ S = exists (fn z => z = x) S;

(* val choice : ''a CONJ -> ''a * ''a CONJ *)
exception ERROR_CHOICE;
fun choice([]) = raise ERROR_CHOICE |
    choice(a::S) = (a, S);

(* val U_ : ''a CONJ * ''a CONJ -> ''a CONJ *)
fun R U_ S = R @ S;

(* val sacar_elem : ''a CONJ * ''a -> ''a CONJ *)
fun sacar_elem([], x) = [] |
    sacar_elem(x::S, y) = if (x=y) then
                             sacar_elem(S, y)
                          else
                             x::sacar_elem(S, y);

(* val dif_ : ''a CONJ * ''a CONJ -> ''a CONJ *)
fun [] dif_ D = [] |
    D dif_ [] = D |
    (x::C) dif_ (y::D) = sacar_elem(x::C, y) dif_ D;

(* val uniques : ''a list -> ''a CONJ *)
fun uniques([]) = [] |
    uniques(a::S) = if (a en_ S) then
                       uniques(S)
                    else
                       a::uniques(S);

end;
open str_conj;
```

```
(******************)
(**** FORMULAS ****)
(******************)

infix 6 v_;
infix 6 y_;
infix 5 imp_;
infix 5 ssi_;
infix 5 ge_;

signature SIG_FORM =
sig
   datatype FORM = ! of FORM |
                   v_ of FORM * FORM |
                   T |
                   L of string * FORM |
                   LI of string * FORM |
                   p of string;

   (* Retorna la constante F de falsum *)
   val F: FORM

   (* Operador modal de posibilidad *)
   val M: string * FORM -> FORM

   (* Operador modal de posibilidad inversa *)
   val MI: string * FORM -> FORM

   (* Operador de conjuncion *)
   val y_: FORM * FORM -> FORM

   (* Operador de implicacion *)
   val imp_: FORM * FORM -> FORM

   (* Operador de doble implicacion *)
   val ssi_: FORM * FORM -> FORM

   (* Retorna el conjunto de subformulas de una formula dada *)
   val sub : FORM -> FORM str_conj.CONJ

   (* Retorna el tama#o de una formula, definido como el tama#o (en numero
      de elementos de su conjunto de subformulas *)
   val tam: FORM -> int

   (* Predicado de "mayor o igual" o grado de complejidad definido por
      comparacion del tama#o de las formulas *)
   val ge_ : FORM * FORM -> bool

   (* Dado un conjunto (finito) de formulas, retorna la conjuncion de todas
      las formuals del conjunto *)
   val andform : FORM str_conj.CONJ -> FORM

   (* Dado un conjunto de formulas, retorna el conjunto de labels
      utilizados en dichas formulas *)
   val labels : FORM str_conj.CONJ -> string str_conj.CONJ

   (* Imprime por pantalla una formula *)
   val print_form: FORM -> unit

   (* Imprime por pantalla un conjunto de formulas *)
   val print_conjformulas : FORM str_conj.CONJ -> unit
end;

structure str_form : SIG_FORM =
struct

   datatype FORM = ! of FORM |
                   v_ of FORM * FORM |
                   T |
                   L of string * FORM |
                   LI of string * FORM |
                   p of string;
```

```
    (* val F: FORM*)
    val F = !T;

    (* val M: string * FORM -> FORM *)
    fun M(n, f) = !(L(n, !f));

    (* val MI: string * FORM -> FORM *)
    fun MI(n, f) = !(LI(n,!f));

    (* val y_: FORM * FORM -> FORM *)
    fun f y_ g = !((!f) v_ (!g));

    (* val imp_: FORM * FORM -> FORM *)
    fun f imp_ g = (!f) v_ g;

    (* val ssi_: FORM * FORM -> FORM *)
    fun f ssi_ g = (f imp_ g) y_ (g imp_ f);

    (* val sub : FORM -> FORM list *)
    fun sub(T) = [T] |
        sub(p(n)) = [p(n)] |
        sub(!(f)) = !(f)::sub(f) |
        sub(f v_ g) = (f v_ g) :: (sub(f) @ sub(g)) |
        sub(L(n,f)) = L(n, f) :: sub(f) |
        sub(LI(n,f)) = LI(n, f) :: sub(f);

    (* val tam: FORM -> int *)
    fun tam(T) = 1 |
        tam(p(n)) = 1 |
        tam(!(f)) = 1 + tam(f) |
        tam(f v_ g) = 1 + (tam(f) + tam(g)) |
        tam(L(n,f)) = 1 + tam(f) |
        tam(LI(n,f)) = 1 + tam(f);

    (*val ge_ : FORM * FORM -> bool *)
    fun f ge_ g = tam(f) >= tam(g);

    (* val andform : FORM CONJ -> FORM *)
    fun andform([]) = T |
        andform(a::S) = (a y_ andform(S));

    (* val labels_oc : FORM CONJ -> string CONJ *)
    fun labels_oc([]) = [] |
        labels_oc(T::S) = labels_oc(S) |
        labels_oc(p(n)::S) = labels_oc(S) |
        labels_oc((!(f))::S) = labels_oc(sub(f)) U_ labels_oc(S) |
        labels_oc((f v_ g)::S) = labels_oc(sub(f)) U_ labels_oc(sub(g)) U_ labels_oc(S) |
        labels_oc((L(n,f))::S) = n::(labels_oc(sub(f)) U_ labels_oc(S)) |
        labels_oc((LI(n,f))::S) = n::(labels_oc(sub(f)) U_ labels_oc(S));

    (* val labels : FORM CONJ -> string CONJ *)
    fun labels(S) = uniques(labels_oc(S));

    (* val print_form: FORM -> unit;*)
    fun print_form(T) = print " T " |
        print_form(p(n)) = (print " p("; print n; print ") ") |
        print_form(!(f)) = (print " !("; print_form(f); print ") ") |
        print_form(f v_ g) = (print " ("; print_form(f); print "v"; print_form(g); print ") ") |
        print_form(L(n,f)) = (print " ["; print n; print "]"; print_form(f)) |
        print_form(LI(n,f)) = (print " ["; print n; print "]i"; print_form(f));

    (* val print_conjformulas: FORM str_conj.CONJ -> unit *)
    fun print_conjformulas([]) = print "" |
        print_conjformulas(f::ls) = (print "|";
                                     print_form(f);
                                     print "|";
                                     print_conjformulas(ls));

end;
open str_form;
```

```
(******************)
(**** RELACION ****)
(******************)

signature SIG_RELACION =
   sig
      type ''a RELACION
      type ''a RELACIONES

   (* Dada una relacion, retorna el nombre de la relacion *)
   val da_nombre : ''a RELACION -> string

   (* Dada una relacion, retorna el conjunto de pares que la formas (es
      decir, la relacion sin nombre) *)
   val da_rel : ''a RELACION -> (''a * ''a) str_conj.CONJ

   (* Dada una relacion y un elemento e, relorna la lista de sucesores de e
      por la relacion *)
   val accesibles : ''a RELACION * ''a -> ''a list

   (* Dada una relacion y un elemento e, retorna la lista de antecesores
      (sucedores inversos) de e por la relacion *)
   val accesiblesi : ''a RELACION * ''a -> ''a list

   (* Dado un conjunto de relaciones y el nombre de una relacion, retorna
      la relacion con dicho nombre *)
   val da_rel_conj : ''a RELACIONES * string -> ''a RELACION

   (* Imprime una relacion sobre conjuntos de enteros por pantalla *)
   val print_rel : (int str_conj.CONJ) RELACION -> unit

   (* Imprime un conjunto de relaciones sobre conjuntos de enteros por pantalla *)
   val print_rels : (int str_conj.CONJ) RELACIONES -> unit
end;

structure str_relacion : SIG_RELACION =
   struct
      type ''a RELACION = string * (''a * ''a) str_conj.CONJ;
      type ''a RELACIONES = (''a RELACION) str_conj.CONJ;

(* val da_nombre : ''a RELACION -> string *)
fun da_nombre((x,y)) = x;

(* val da_rel : ''a RELACION -> (''a * ''a) str_conj.CONJ *)
fun da_rel((x,y)) = y;

(* val acc : ('a * 'a) CONJ * 'a -> 'a list *)
fun acc([], x) = [] |
    acc((w,z)::R, x) = if (x = w) then
                          z::acc(R, x)
                       else
                          acc(R, x);

(* val acci : ('a * 'a) CONJ * 'a -> 'a list *)
fun acci([], x) = [] |
    acci((w,z)::R, x) = if (x = z) then
                           w::acci(R, x)
                        else
                           acci(R, x);

(* val accesibles : 'a RELACION * 'a -> 'a list *)
fun accesibles(R, x) = acc(da_rel(R), x);

(* val accesiblesi : 'a RELACION * 'a -> 'a list *)
fun accesiblesi(R, x) = acci(da_rel(R), x);

(* val da_rel_conj : ''a RELACIONES * string -> ''a RELACION *)
exception ERROR_DA_REL_CONJ;

fun da_rel_conj'([], s) = raise ERROR_DA_REL_CONJ |
    da_rel_conj'(R::CR, s) = if (da_nombre(R) = s) then
                                R
                             else
                                da_rel_conj'(CR, s);

fun da_rel_conj(CR, s) = da_rel_conj'(CR, s) handle
    ERROR_DA_REL_CONJ => (print "ERROR: 'da_rel_conj' llamada con conjuto vacio.";(s,[]));
```

```
(* val print_rel_s : (int str_conj.CONJ * int str_conj.CONJ) str_conj.CONJ -> unit *)
fun print_rel_s([]) = print "" |
    print_rel_s((x,y)::ls) = (print "(";
                              print_list_int(x);
                              print ",";
                              print_list_int(y);
                              print " ) ";
                              print_rel_s(ls));

(* val print_rel : (int str_conj.CONJ) RELACION -> unit *)
fun print_rel((n: string, r)) = (print "Relacion ";
                                 print n;
                                 print ": ";
                                 print_rel_s(r);
                                 print "\n");

(* val print_rels : (int str_conj.CONJ) RELACIONES -> unit *)
fun print_rels([]) = print "" |
    print_rels(R::CR) = (print_rel(R);
                         print "\n";
                         print_rels(CR));

end;
open str_relacion;
```

```
(*******************)
(**** VALUACION ****)
(*******************)

signature SIG_VALUACION =
sig
   type ''a VALUACION

   (* Dada una lista formada por pares (mundo, lista de proposiciones)
      retorna la valuacion asociada a dicha lista *)
   val list2val : (''a * str_form.FORM list) list -> ''a VALUACION

end;

structure str_valuacion : SIG_VALUACION =
struct
   type ''a VALUACION  = (''a * str_form.FORM) -> bool;

(* val list2val : (''a * str_form.FORM list) list -> ''a VALUACION *)
fun list2val([]) = (fn(x,y) => false) |
    list2val((x, R)::S) = (fn(y,w) => if (x = y) then
                                         (w en_ R)
                                      else
                                         list2val(S) (y,w));
end;
open str_valuacion;
```

```
(****************)
(**** MODELO ****)
(****************)

infix 4 sat_;
infix 4 Vsat_;
infix 4 Esat_;

signature SIG_MODELO =
   sig
      type ''a MODELO

   (* Predicado "satisface", dado un modelo, un mundo y una formula, retorna
      true si el modelo en dicho mundo hace valida la formula segun la
      definicion semantica de la logica KPI *)
   val sat_ : (''a MODELO * ''a) * str_form.FORM -> bool

   (* Cuantificacion universal sobre mundos del predicado "satisface", dado un modelo y
      una formula, retorna true si la formula es validad en todo mundo del modelo *)
   val Vsat_ :''a MODELO * str_form.FORM -> bool

   (* Cuantificacion existencial sobre mundos del predicado "satisface", dado
      un modelo y una formula, retorna true si existe un mundo en el cual la
      el modelo valida la formula *)
   val Esat_ :''a MODELO * str_form.FORM -> bool

end;

structure str_modelo : SIG_MODELO =
   struct
      type ''a MODELO = ''a str_conj.CONJ * ''a str_relacion.RELACIONES * ''a str_valuacion.VALUACION;

(* val sat_ : (''a MODELO * ''a) * str_form.FORM -> bool *)
infix 4 sat_';

exception ERROR_SATISF;

fun (([], CR, V), x) sat_' f = raise ERROR_SATISF |
    ((w::W, CR, V), x) sat_' T = true |
    ((w::W, CR, V), x) sat_' (p(n)) = V(x, p(n)) |
    ((w::W, CR, V), x) sat_' (!(f)) = not(((w::W, CR, V), x) sat_' f) |
    ((w::W, CR, V), x) sat_' (f v_ g) = (((w::W, CR, V), x) sat_' f)
                                    orelse (((w::W, CR, V), x) sat_' g) |
    ((w::W, CR, V), x) sat_' (L(a, f)) = let
                                           val funcion = fn x => (((w::W, CR, V), x) sat_' f)
                                         in
                                           andlist(map funcion (accesibles(da_rel_conj(CR,a),x)) )
                                         end |
    ((w::W, CR, V), x) sat_' (LI(a, f)) = let
                                            val funcion = fn x => (((w::W, CR, V), x) sat_' f)
                                          in
                                            andlist(map funcion (accesiblesi(da_rel_conj(CR,a),x)) )
                                          end;

fun (M, x) sat_ f = (M, x) sat_' f handle
    ERROR_SATISF => ("ERROR: El modelo no puede tener dominio vacio."; false)

(* val Vsat_ ''a MODELO * str_form.FORM -> bool *)
infix 4 Vsat_'

fun ([], CR, V) Vsat_' f = raise ERROR_SATISF |
    (w::W, CR, V) Vsat_' f = andlist(map (fn x => ((w::W, CR, V), x) sat_ f) (w::W));

fun M Vsat_ f = M Vsat_' f handle
    ERROR_SATISF => ("ERROR: El modelo no puede tener dominio vacio."; false)

(* val Esat_ ''a MODELO * str_form.FORM -> bool *)
infix 4 Esat_'

fun ([], CR, V) Esat_' f = raise ERROR_SATISF |
    (w::W, CR, V) Esat_' f = orlist(map (fn x => ((w::W, CR, V), x) sat_ f) (w::W));

fun M Esat_ f = M Esat_' f handle
    ERROR_SATISF => ("ERROR: El modelo no puede tener dominio vacio."; false)

end;
open str_modelo;
```

```
(*************)
(**** KPI ****)
(*************)

infix 4 csat_;
infix 4 Vcsat_;
infix 4 Ecsat_;

signature SIG_KPI =
   sig

   (* Algoritmo de Model Checking.  Dado un modelo y una formula f retorna la
      formada por los pares (mundo, conjunto de formulas) tal que todo
      mundo del modelo tiene asignado el conjunto de subformulas de f que
      son validadas por el *)
   val check : ''a str_modelo.MODELO * str_form.FORM -> (''a * str_form.FORM str_conj.CONJ) list

   (* Predicado "satisface" definido utilizando el algoritmo de model
      checking *)
   val csat_ : (''a str_modelo.MODELO * ''a) * str_form.FORM -> bool

   (* Predicado "satisface en todo mundo" definido utilizando el algoritmo
      de model checking *)
   val Vcsat_ : ''a str_modelo.MODELO * str_form.FORM -> bool

   (* Predicado "satisface en un mundo" definido utilizando el algoritmo de
      model checking *)
   val Ecsat_ : ''a str_modelo.MODELO * str_form.FORM -> bool

   (* Algoritmo de Filtracion.  Dado un modelo M y un conjunto de formulas CF
      retorna el modelo resultado de filtrar M por la clausura por
      subformulas de CF *)
   val filtrar : ''a str_modelo.MODELO * str_form.FORM str_conj.CONJ
                 -> (''a str_conj.CONJ) str_modelo.MODELO

   (* Rutina que imprime por pantalla el resultado del algoritmo de model
   checking, cuando el modelo tiene por dominio los enteros (la restriccion
   se debe a las limitaciones impuestas por SML a las funciones de impresion) *)
   val print_check_int : (int * str_form.FORM str_conj.CONJ) list -> unit

   (* Rutina que imprime por pantalla el resultado del algoritmo de
   filtrado, cuando el modelo tiene por dominio los enteros (la restriccion
   se debe a las limitaciones impuestas por SML a las funciones de impresion) *)
   val print_filtrar_int: (int str_conj.CONJ) str_modelo.MODELO -> unit

end;

structure str_kpi : SIG_KPI =
struct

(* val condL : ''a RELACION * ''a * (''a * str_form.FORM str_conj.CONJ) list * str_form.FORM -> bool *)
fun condL(R, w, l, f) = let
                           val funcion = (fn (x, y) => (not((w,x) en_ R) orelse (f en_ y)))
                        in
                           andlist(map funcion l)
                        end;

(* val condLI : ''a RELACION * ''a * (''a * str_form.FORM str_conj.CONJ) list * str_form.FORM -> bool *)
fun condLI(R, w, l, f) = let
                            val funcion = (fn (x, y) => (not((x, w) en_ R) orelse (f en_ y)))
                         in
                            andlist(map funcion l)
                         end;

(* val check_oc : ''a str_modelo.MODELO * str_form.FORM list * (''a, str_form.FORM str_conj.CONJ) list ->
(''a, str_form.FORM str_conj.CONJ) list *)
fun check_oc((W, CR, V), [], L2)
     = L2 |
    check_oc((W, CR, V), T::L1, L2)
     = let
          val funcion = (fn (x, y) => (x, T::y))
       in
          check_oc((W, CR, V), L1, map funcion L2)
       end |
```

```
    check_oc((W, CR, V), (p(s))::L1, L2)
     = let
          val funcion = (fn (x, y) => if (V(x, p(s))) then
                                         (x, (p(s))::y)
                                      else
                                         (x, y))
       in
          check_oc((W, CR, V), L1, map funcion L2)
       end |
    check_oc((W, CR, V), (!(f))::L1, L2)
     = let
          val funcion = (fn (x, y) => if not(f en_ y) then
                                         (x, (!(f))::y)
                                      else
                                         (x, y))
       in
          check_oc((W, CR, V), L1, map funcion L2)
       end |
    check_oc((W, CR, V), (f v_ g)::L1, L2)
     = let
          val funcion = (fn (x, y) => if (f en_ y) orelse (g en_ y) then
                                         (x, (f v_ g)::y)
                                      else
                                         (x, y))
       in
          check_oc((W, CR, V), L1, map funcion L2)
       end |
    check_oc((W, CR, V), L(a, f)::L1, L2)
     = let
          val funcion = (fn (x, y) => if (condL(da_rel(da_rel_conj(CR, a)), x, L2, f)) then
                                         (x, (L(a, f))::y)
                                      else
                                         (x, y))
       in
          check_oc((W, CR, V), L1, map funcion L2)
       end |
    check_oc((W, CR, V), LI(a, f)::L1, L2)
     = let
          val funcion = (fn (x, y) => if (condLI(da_rel(da_rel_conj(CR, a)), x, L2, f)) then
                                         (x, (LI(a, f))::y)
                                      else
                                         (x, y))
       in
          check_oc((W, CR, V), L1, map funcion L2)
       end;

(* val check : ''a str_modelo.MODELO * str_form.FORM -> (''a * str_form.FORM str_conj.CONJ) list *)
exception ERROR_VACIO;
fun check'(([], CR, V), f) = raise ERROR_VACIO |
    check'((w::W, CR, V), f)
       = check_oc((w::W, CR, V), (lista_ord (op ge_) (sub(f))), (map (fn x => (x,[])) (w::W)));

fun check(Modelo, f) = check'(Modelo, f) handle
    ERROR_VACIO => (print "ERROR: El modelo no puede tener dominio vacio."; []);

(* val csat_ : (''a str_modelo.MODELO * ''a) * str_form.FORM -> bool *)
infix 4 csat_';
fun (([], CR, V), x) csat_' f = raise ERROR_VACIO |
    ((w::W, CR, V), x) csat_' f = let
                                     val funcion = (fn (z1,z2) => (not(x = z1) orelse (f en_ z2)))
                                  in
                                     andlist( map funcion (check((w::W, CR, V), f)) )
                                  end;

fun (Modelo, x) csat_ f = (Modelo, x) csat_' f handle
    ERROR_VACIO => (print "ERROR: El modelo no puede tener dominio vacio."; false);

(* val Vcsat_ : ''a str_modelo.MODELO * str_form.FORM -> bool *)
infix 4 Vcsat_';
fun ([], CR, V) Vcsat_' f = raise ERROR_VACIO |
    (w::W, CR, V) Vcsat_' f = let
                                 val funcion = (fn (x,y) => (f en_ y))
                              in
                                 andlist( map funcion (check((w::W, CR, V), f)) )
                              end;
```

```
fun Modelo Vcsat_ f = Modelo Vcsat_' f handle
    ERROR_VACIO => (print "ERROR: El modelo no puede tener dominio vacio."; false);

(* val Ecsat_ : ''a str_modelo.MODELO * str_form.FORM -> bool *)
infix 4 Ecsat_';
fun ([], CR, V) Ecsat_' f = raise ERROR_VACIO |
    (w::W, CR, V) Ecsat_' f = let
                                 val funcion = (fn (x,y) => (f en_ y))
                              in
                                 orlist( map funcion (check((w::W, CR, V), f)) )
                              end;

fun Modelo Ecsat_ f = Modelo Ecsat_' f handle
    ERROR_VACIO => (print "ERROR: El modelo no puede tener dominio vacio."; false);

(* pasa_a_clases : (''a * str_form.FORM str_conj.CONJ) list * (''a str_conj.CONJ) str_conj.CONJ
                -> (''a str_conj.CONJ) str_conj.CONJ *)
fun pasa_a_clases([], CL) = CL |
    pasa_a_clases((w, C)::S, CL)
       = let
            val (iguales, distintos) = ( (find (fn (x,y) => (y = C)) ((w, C)::S)),
                                         (find (fn (x,y) => not(y = C)) ((w, C)::S)) )
         in
            pasa_a_clases(distintos, ((map (fn (x,y) => x) iguales)::CL))
         end;

(* val construye_W : ''a str_modelo.MODELO * str_form.FORM str_conj.CONJ
                  -> (''a str_conj.CONJ) str_conj.CONJ *)
fun construye_W((W, CR, V), C) = pasa_a_clases(check((W, CR, V), andform(C)), []);

(* val da_clase : (''a str_conj.CONJ) str_conj.CONJ * ''a -> ''a str_conj.CONJ *)
fun da_clase(CL, a) = findone((fn x => a en_ x), CL);

(* val arma_Rels : ''a RELACIONES * (''a str_conj.CONJ) str_conj.CONJ * (''a str_conj.CONJ) RELACIONES *)
fun arma_Rels([], Clases , Rels) = Rels |
    arma_Rels(((nombre, r)::CR), Clases, Rels)
       = let
            val funcion = fn (x, y) => (da_clase(Clases, x), da_clase(Clases, y))
         in
            let
               val Relacion = uniques(map funcion r)
            in
               arma_Rels(CR, Clases, ((nombre, Relacion)::Rels))
            end
         end;

(* val construye_Rels : ''a str_modelo.MODELO * str_form.FORM str_conj.CONJ
                    -> (''a str_conj.CONJ) RELACIONES *)
fun construye_Rels((W, CR, V), C) = let
                                       val NCR = find (fn (x,y) => (x en_ labels(C))) CR
                                    in
                                       arma_Rels(NCR, construye_W((W, CR, V), C),[])
                                    end;

(* val construye_V : ''a str_modelo.MODELO * str_form.FORM str_conj.CONJ
                  -> (''a str_conj.CONJ) VALUACION *)
fun construye_V((W, CR, V), C) = (fn (S,x) => ((x en_ sub(andform(C))) andalso V(hd(S), x)));

(* val filtrar : ''a str_modelo.MODELO * str_form.FORM str_conj.CONJ
               -> (''a str_conj.CONJ) str_modelo.MODELO *)
fun filtrar(([], CR, V), C) = raise ERROR_VACIO |
    filtrar((w::W, CR, V), C) = ( construye_W((w::W, CR, V), C),
                                  construye_Rels((w::W, CR, V), C),
                                  construye_V((w::W, CR, V), C) );

(* val print_check_int : (int * str_form.FORM str_conj.CONJ) list -> unit *)
fun print_check_int([]) = print "" |
    print_check_int((x, CF)::ls : (int * str_form.FORM str_conj.CONJ) list)
       = (print "Mundo: ";
          print x;
          print "\tFormulas: ";
          print_conjformulas(CF);
          print "\n";
          print_check_int(ls));
```

```
(* val print_mundos : (int CONJ) CONJ -> unit *)
fun print_mundos([]) = print "" |
    print_mundos(l::ls) = (print "|";
                           print_list_int(l);
                           print " |";
                           print_mundos(ls));

(* val print_filtrar_int: (int str_conj.CONJ) str_modelo.MODELO -> unit *)
fun print_filtrar_int((W, CR, V)) = (print "Mundos:\n";
                                     print_mundos(W);
                                     print "\n\nRelaciones:\n";
                                     print_rels(CR));

end;
open str_kpi;
```

```
print_check_int(check(kwic_modelo, kwic_formula));

Mundo: 1
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | [espartede] !( T ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 2
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | [espartede] !( T ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 3
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | [espartede] !( T ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 4
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( [espartede]i !( T ) ) |
             | [espartede] !( T ) |
             | T |
Mundo: 5
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( [espartede]i !( T ) ) |
             | [espartede] !( T ) |
             | T |
Mundo: 6
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( [espartede]i !( T ) ) |
             | [espartede] !( T ) |
             | T |
Mundo: 7
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | [espartede] !( T ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 8
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | [espartede] !( T ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 9
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | !( [espartede] !( T ) ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 10
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | !( [espartede] !( T ) ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 11
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | !( [espartede] !( T ) ) |
             | [espartede]i !( T ) |
             | T |
```

```
Mundo: 12
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | !( [espartede] !( T ) ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 13
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | !( [espartede] !( T ) ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 14
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | !( [espartede] !( T ) ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 15
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | !( [espartede] !( T ) ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 16
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | !( [espartede] !( T ) ) |
             | [espartede]i !( T ) |
             | T |
Mundo: 17
   Formulas: | ( !( !( [espartede]i !( T ) ) ) v [invoca] !( [espartede] !( T ) ) ) |
             | [invoca] !( [espartede] !( T ) ) |
             | !( !( [espartede]i !( T ) ) ) |
             | !( [espartede] !( T ) ) |
             | [espartede]i !( T ) |
             | T |
```

print_filtrar_int(filtrar(kwic_modelo, [kwic_formula]));

```
Mundos:
   | 9 10 11 12 13 14 15 16 17 |
   | 4 5 6 |
   | 2 3 7 8 |
   | 1 |

Relaciones:
   Relacion espartede: ( 9 10 11 12 13 14 15 16 17, 4 5 6 )

   Relacion invoca: ( 1, 2 3 7 8 )
                    ( 1, 9 10 11 12 13 14 15 16 17 )
                    ( 2 3 7 8, 9 10 11 12 13 14 15 16 17 )
                    ( 4 5 6, 9 10 11 12 13 14 15 16 17 )
```

Referencias

[Agustí et al., 1995]
Agustí, J., Robertson, D. & Puigsegur, J., *GraSp: A GRAphical SPecification Language for the Preliminary Specification of Logic Programs.*
Institut d'Investigació en Intel.ligència Artificial (CSIC). Internal Report. 1995.

[Allen & Garlan, 1994]
Allen, R. & Garlan, D., *Formalizing Architectural Connection.*
Proc. International Conference on Software Engineering. May 1994.

[AT&T Bell Laboratories, 1993]
AT&T Bell Laboratories, *Standard ML of New Jersey. User's Guide (Version 0.93).*
AT&T Bell Laboratories. 1993.

[Beth, 1953]
Beth, E., *On Padoa's Method in the Theory of Definition.*
Koninklijke Nederlandse Akad. Van Wetenshch 56, A, Math. Sciencies, pp. 330-339.

[Blackburn, 1993]
Blackburn, P., *Nominal Tense Logic.*
Notre Dame Journal of Formal Logic N° 34. 1993.

[Bourdeau & Cheng, 1995]
Bourdeau, R & Cheng, B., *A Formal Semantics for Object Model Diagrams.*
IEEE Transactions on Software Engineering, Vol. 21, N° 10. October 1995.

[Burgess, 1984]
Burgess, J. P., *Basic Tense Logic.*
Handbook of Philosophical Logic, Vol. II. D. Reidel Publishing Company. 1984.

[Chellas, 1980]
Chellas, B. F., *Modal Logic, An Introduction.*
Cambridge University Press. 1980.

[Clarke et. al., 1986]
Clarke, E., Emerson, E & Sistla, A, *Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications.*
ACM Transactions on Programming Languages and Systems, Vol. 8, N° 2, 1986.

[Cosens et al., 1992]
Cosens, M., Mendelzon, A. & Ryman, A., *Visualizing and Quering Software Structures.*
ICSE'92 Proceedings of the 14th International COnference on Software Engineering. 1992.

[Dean & Cordy, 1995]
Dean, T. & Cordy, J, *A Syntactic Theory of Software Architecture.*
IEEE Transactions on Software Engineering, Vol 21, N° 4. April 1995.

[Fiadeiro et al., 1991]
Fiadeiro, J., Sernadas, C., Maibaum, T. & Saake, G., *Proof-theoretic Semantics of Object-Oriented Specification Constructs.*
Kent, Khosla and Meersman (eds) Object-Oriented Databases: Analysis, Design and Construction. North-Holland. 1991.

[Garlan & Shaw, 1993]
Garlan, D. & Shaw, M., *An Introduction to Software Architecture.*
Advances in Software Engineering and Knowledge Engineering, Vol I. World Scientific Publishing Company. 1993.

[Gauthier & Pont, 1970]
Gauthier, R. & Pont, S., *Designing Systems Programs.*
Prentice Hall, Englewood Cliffs, N.J. 1970.

[Ghezzi et al., 1991]
Ghezzi, C., Jazayeri, M. & Mandrioli, D., *Fundamentals of Software Engineering.*
Prentince Hall. 1991.

[Harel, 1984]
Harel, D., *Dynamic Logic.*
Gabbay & Guenthner (eds). Handbook of Philosophical Logic, Vol. II. 1984.

[Hintikka, 1962]
Hintikka, J., *Knowledge and Belief.*
Cornell University Press. 1962.

[Hirsch & Areces, 1995]
Hirsch, D. & Areces, C., *From Boxes to Worlds.*
Anales del Primer CACiC. 1995.

[Hughes & Cresswell, 1984]
Hugues, G. E. & Cresswell, M. J., *A Companion to Modal Logic.*
Methuen and Co. Ltd. 1984.

[Kripke, 1963a]
Kripke, S., *Semantical Analysis of Modal Logic I, Normal Propositional Calculi.*
ZML Vol. 9. 1963.

[Kripke, 1963b]
Kripke, S., *Semantical Considerations on Modal Logics.*
Acta Philosophica Fennica. Modal and Many-valued Logics. 1963.

[Lamport, 1994]
Lamport, L., *The Temporal Logic of Actions.*
ACM Transactions on Programming Languages and Systems, Vol 16, N°3, 1994.

[Lemmon & Scott, 1977]
Lemmon, E. & Scott, D., *The "Lemmon Notes": An Introduction to Modal Logic.*
K. Segerberg (ed.). Oxford, Blackwell. 1977.

[Maibaum et al., 1984]
Maibaum, T., Sadler, M. & Veloso, P., *Logical Specification and Implementation.*
Lecture Notes in Computer Science N° 18. Springer-Verlag. New York. 1984.

[Page-Jones, 1980]
Page Jones, M., *The Practical Guide to Structured Systems Design.*
Prentice Hall, 1980.

[Parnas, 1972]
Parnas, D. L., *On the Criteria to be used in Decomposing Systems into Modules.*
Communications of ACM, December 1972, pages 1053-1058

[Parnas, 1979]
Parnas, D. L., *Designing Software for Ease of Extension and Contraction.*
IEEE Transactions on Software Engineering, March 1979, Volume SE-5, N° 2, pages 128-138.

[Paul & Prakash, 1996]
Paul, S. & Prakash, A., *A Query Algebra for Program Databases.*
IEEE Transactions on Software Engineering, Vol. 22, N° 3, March 1995.

[Pnueli, 1977]
Pnueli, A., *The Temporal Logic of Programs.*
In Proceedings of the 18th. Annual Symposium on the Foundations of Computer Science. IEEE. New York. 1977.

[Popkorn, 1994]
Popkorn, S., *First Steps in Modal Logic.*
Cambridge University Press. 1994.

[Pratt, 1978]
Pratt, V., *Application of Modal Logic to Programming.*
Studia Logica N° 39. 1978.

[de Rijke, 1993]
de Rijke, M., *Extending Modal Logic.*
ILLC Dissertation Series. University of Amsterdam. 1993.

[Stirling, 1992]
Stirling, C., *Modal and Temporal Logics.*
Handbook of Computer Science, Abramsky et al. (ed.). Vol. II. Oxford University Press. 1992.

[Tarski, 1956]
Tarski, A., *Logic, Semantics, Metamathematics. Papers from 1923 to 1938.*
John Corcoran (ed.). Hackett Publishing Company. 1956.

[Ullman, 1994]
Ullman, J., *Elements of ML Programming.*
Prentice Hall. 1994.

[Wirfs-Brock et al., 1990]
Wirfs-Brock, R., Wilkerson, B. & Wiener, L., *Designing Object-Oriented Software.*
Prentice Hall. 1990.