UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

UN FRAMEWORK PARA EL CHEQUEO DE CONSISTENCIA EN MODELOS UML

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

SEBASTIÁN RODRIGO RIVAS CHIESSA

PROFESOR GUÍA:
MARIA CECILIA BASTARRICA PIÑEYRO

MIEMBROS DE LA COMISIÓN:
LUIS GUERRERO BLANCO
EDUARDO GODOY VEGA

SANTIAGO DE CHILE
2007

Agradecimientos

Es indudable que lo que hoy presento, y he logrado, no hubiera sido posible si en mi vida no existiera Dios. Él es mi verdadera fuente de lo que algunos consideran “inteligencia” o “talento”, y que me hicieron llegar hasta donde hoy me encuentro. Por eso lo primero es decir "Gracias por todo Dios mío”.

Sin embargo existen muchas personas que hicieron posible no sólo este trabajo sino todos mis estudios y mi desarrollo como profesional y persona. En este sentido debo manifestar mi agradecimiento eterno a mis padres, quienes me formaron como persona, me vieron crecer, me dieron todo su apoyo y cultivaron en mí los valores que hoy llevo por delante y que me permitieron conseguir este sueño. Gracias Papá por aquellas palabras que marcaron esa prueba de aptitud que fue el paso que me permitió iniciar todo este mundo, y que me hicieron sentir la tranquilidad que necesitaba entonces. “Hoy no comienza ni acaba tu vida”.

Muchas gracias a usted profesora Cecilia, usted fue quién confió en mi desde el principio y me ayudó en todo momento a lograr lo que hoy presento. Sin embargo me gustaría agradecerle más que a la profesora, a la persona solidaria, muy humana y dedicada que no dudó en prestarme ayuda a pesar de sus propios problemas de tiempo.

Gracias también a Jocelyn Simmonds por la buena disposición a conversar y prestarme ayuda.

Finalmente agradecer a la persona más importante en mi vida, a quién me ayudó a salir adelante a pesar de todas las dificultades que se me presentaron, a quién confió siempre en mi y en mis capacidades, a quién me ayudó a tomar las difíciles decisiones que tuve que tomar para salir adelante, y quien muchas veces con un pequeño gesto me daba la fuerza necesaria para seguir. Gracias Paulina Arce Recabal, mi gran amor.

Índice de Contenidos

Índice de Figuras

1. Introducción

Es indudable hoy en día que el desarrollo de software se ha convertido en una actividad de gran importancia, principalmente, debido a que puede repercutir en diversas actividades cotidianas de las personas. Dada dicha diversidad de actividades que abarcan los desarrollos, los equipos de trabajo suelen estar constituidos a su vez por personas que trabajan en áreas muy diversas. Resulta entonces vital para estos grupos de trabajo contar con un lenguaje común que les permita comunicarse de mejor manera. Justamente con esta intención es que nace UML (Unified Modeling Language) que se ha transformado en el lenguaje de modelado de sistemas de software más utilizado en la actualidad.

UML es un lenguaje gráfico para visualizar, especificar, construir y documentar sistemas de software. Básicamente ofrece una familia de diagramas para describir distintos aspectos del sistema, incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables. UML fue adoptado por el OMG (Object Management Group) en el año 1997 como el estándar de-facto para el modelamiento orientado a objetos. Desde entonces atravesó varias revisiones y refinamientos hasta llegar a la versión actual (UML 2.0) aprobada en octubre de 2004 [11].

Si bien, como fue mencionado anteriormente, UML 2.0 es el estándar dentro en la industria, esto no significa que sea definitivo ya que cuenta con una serie de dificultades. De hecho, no define una clara relación entre la semántica de los distintos diagramas, ni ofrece políticas de versionamiento en el caso de la evolución de un modelo. Estas dificultades son justificadas aduciendo a que no toda inconsistencia es accidental. Por ejemplo, cuando se hace un diseño abarcando desde lo global a lo particular, se inicia el proceso de diseño con un modelo incompleto, por lo tanto inconsistente.

Indudable es que el uso de herramientas CASE facilita bastante la labor del diseñador, sobre todo en desarrollos de gran tamaño y complejidad. Sin embargo, dada la posición de los creadores de UML respecto a la validez de las inconsistencias, el usuario de UML debe preocuparse de las inconsistencias en forma manual.

1.1 Motivación

UML, al ser un lenguaje, cuenta con una sintaxis y una semántica definidas. Por lo tanto, al modelar un concepto en UML, existen reglas sobre cómo deben agruparse los elementos del lenguaje y el significado de esta agrupación. La sintaxis y semántica de UML están definidas por su metamodelo que determina qué es válido dentro del lenguaje. El problema es que dicho metamodelo no establece restricciones de integridad entre los distintos diagramas pese a que se han identificado muchas restricciones evidentes.

La última especificación de UML es la 2.0. Dicha especificación se esperaba que diera solución a una gran cantidad problemas detectados en las versiones anteriores. Lamentablemente, uno de los problemas que aún persiste pese a la nueva especificación, se debe a que UML no contiene un mecanismo que permita especificar la relación entre distintas versiones de los diagramas y elementos de diagramas. El no contar con este mecanismo provoca que se pierda mucha información implícita que puede ser usada para revisar la consistencia lógica entre diagramas de un modelo y entre modelos. Con la idea de revisar la consistencia lógica de los diagramas se dio paso a la utilización de Description Logics (DL), un fragmento decidible de lógica de primer orden, que permite la detección de inconsistencias.

En la actualidad existen diversas herramientas de modelamiento UML que permiten tanto la creación de estructuras de código a partir del diseño, como la obtención de los diagramas a partir de código fuente. Lo anterior representa una oportunidad inmejorable de chequear la consistencia del diseño, pero dicha oportunidad se ve limitada dado que las herramientas actuales no contemplan dichos chequeos, lo cual pone en riesgo la calidad de los diseños, sobre todo cuando estos son grandes y no es posible revisarlos de manera manual.

El trabajo de formalizar y establecer chequeos de consistencia, fue tema de investigación [14] quedando planteado y resuelto el problema, pero mostrando a su vez la dificultad de aplicar dichos procesos, por lo tanto el problema que plantea su uso y la necesidad de crear un framework que permita “ocultar” del usuario final los formalismos utilizados para la detección de inconsistencias, y realizar dicho chequeo de manera transparente al usuario final interesado sólo en los resultados del chequeo más que en la manera en que se efectúa dicha prueba.

Posterior a la investigación descrita, se realizó el framework “Model Consistency Checker” [15] (desde ahora en adelante MCC) que implementa 5 de los 18 chequeos estudiados.

Lamentablemente, su utilidad queda reducida dada la obsolescencia de sus componentes, así como la dificultad de su instalación y uso. MCC utilizaba versiones hoy obsoletas tanto de Racer [7] como de Poseidon [6]. Ambos programas eran fundamentales para su funcionamiento y hoy han cambiado de manera tal que no permiten la interacción antes desarrollada.

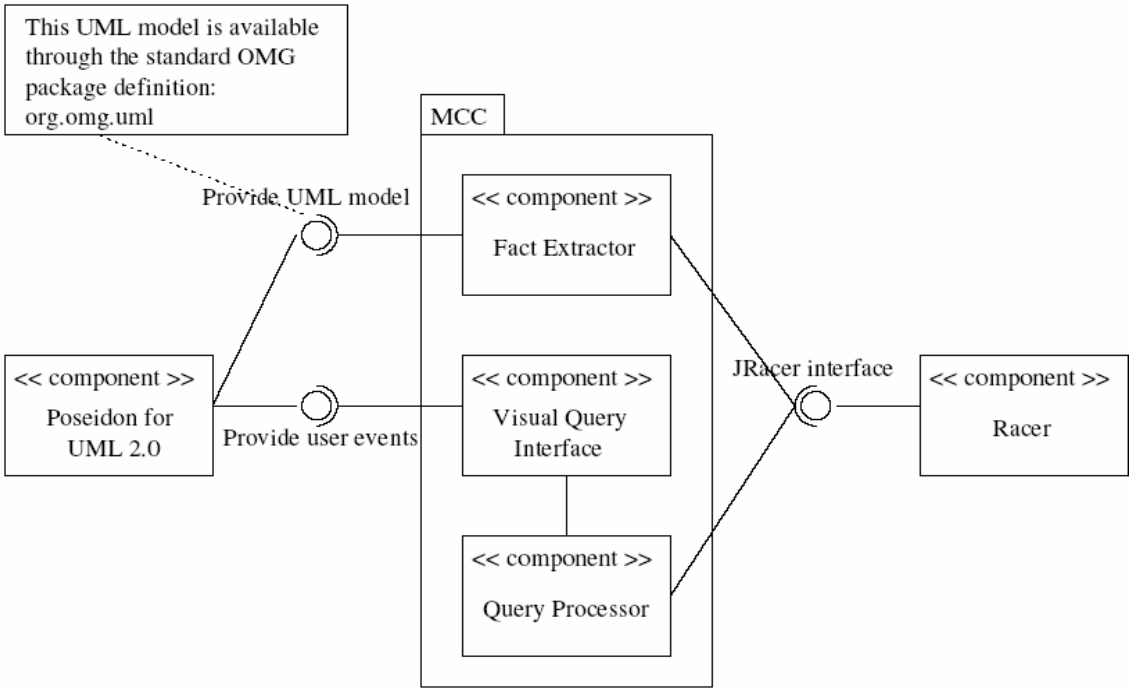

Figura 1 : Modelo UML de MCC y sus relaciones con otras componentes

MCC en la actualidad se encuentra disponible [16], y fue construido (ver Figura 1) utilizando Poseidon 2.6 [6] como interfaz gráfica UML y Racer [7] como motor de inferencia para los chequeos. Además se componía internamente de tres elementos. “Fact Extractor” que extrae la información del modelo UML desde Poseidon creando un modelo lógico que es posteriormente cargado en Racer, “Visual Query Interface” para solicitar los distintos chequeos de consistencia, y “Query Processor” para procesar las solicitudes.

1.2 Objetivos

1.2.1 Objetivo General

El objetivo general de este trabajo consiste en actualizar la herramienta gráfica MCC de manera de convertirla en una herramienta útil dadas las circunstancias actuales en que Poseidon y Racer han cambiado su funcionamiento. Se pretende además, como una forma de fomentar su

uso, simplificar su interfaz y disminuir al máximo la necesidad de configuraciones engorrosas para el usuario final.

1.2.2 Objetivos Específicos

Los objetivos específicos de este trabajo son:

- Estudiar el estado actual de MCC y sus componentes buscando las principales dificultades que hacen engorrosa su utilización.
- Proponer cambios en la versión de MCC, basado en la experiencia adquirida, rediseñando de ser necesario e implementando algunos chequeos en el nuevo escenario.
- Validar la nueva herramienta buscando posibles mejoras a futuro y pasos a seguir.

1.3 Metodología

Para desarrollar de forma adecuada el trabajo, es necesario llevar a cabo una serie de pasos:

- Análisis de situación actual (diseño e implementación) del framework MCC, estudiando sus posibilidades de extensión y adaptación a requerimientos actuales.
- Configurar ambiente de trabajo y herramientas necesarias.
- Estudio de cambios entre versiones de Racer y adaptación a versión actual.
- Estudio de cambios entre versiones antiguas y actuales de Poseidon y actualización del framework al ambiente vigente.
- Propuesta de cambios a MCC de manera de lograr una mejor aplicación
- Reimplementar la herramienta de acuerdo con los cambios sugeridos
- Validar nueva herramienta

2. Antecedentes y Contexto

2.1 UML

UML [11] es un lenguaje de modelamiento visual que provee una familia de diagramas que permiten especificar la estructura y el comportamiento del sistema a construir. Como lenguaje de modelamiento visual, define varios conceptos de modelamiento, la semántica de estos, la notación visual correspondiente y una guía de uso. UML como lenguaje ha sido ampliamente aceptado en el desarrollo de sistemas que usan orientación a objetos y componentes como paradigmas.

En un proyecto, existe la necesidad de especificar los requerimientos y el diseño. Para esto, UML ofrece tres categorías de modelos: de requerimientos, estructurales y dinámicos (especificación del comportamiento del sistema). Una especificación completa de un sistema debería incluir diagramas de las tres categorías, generando una visión global del sistema.

2.1.1 Metamodelo, formalismo, DL y chequeos de consistencia

UML 2.0 se especifica mediante un metamodelo. Es en este metamodelo donde se define en forma extensa cuáles son los elementos que existen y dónde se pueden usar, y se da un conjunto de reglas de buen uso para cada elemento. Para cumplir con el estándar, no es necesario contar con una herramienta que implemente control de consistencia entre modelos, diagramas y elementos. Esto significa que errores pueden ser introducidos en los modelos por descuidos (errores de ortografía, cambio de palabras por sinónimos, etc.), pero a veces, es necesario introducir inconsistencias en los modelos, dado que no existe un modelo completo en los pasos intermedios del proceso de diseño.

Es usual comenzar la etapa de diseño con un modelo más general, el cual es sucesivamente refinado durante el proceso completo de desarrollo del sistema. Por ejemplo, pueden existir especificaciones incompletas de clases, o links que hacen referencia a operaciones que no existen, dado que no todas las responsabilidades han sido definidas todavía. Estas inconsistencias son tolerables durante la fase de diseño, pero los modelos finales que serán la guía para los implementadores deben ser consistentes. Además, en la medida en que los modelos crezcan y adquieran mayor complejidad, es necesario pensar en una forma de llevar a cabo

revisiones automáticas de consistencia, para poder determinar si un modelo propuesto es válido o no.

Como UML en sí no tiene una semántica formal y no se pueden hacer revisiones automáticas de consistencia, resulta necesario introducir un formalismo que permita manipular los modelos y los diagramas, basándose en la semántica de estos y no su representación visual.

Description Logic (DL [3]) es un fragmento decidible de lógica de primer orden. Se define una terminología que permite describir al dominio que es modelado usando un lenguaje de representación. Esto se hace especificando los conceptos existentes en el dominio y las relaciones que los unen. El conjunto de conceptos y relaciones que definen un dominio se conoce como el Tbox. Estos conceptos y relaciones se pueden instanciar, imitando objetos reales del sistema a modelar. El conjunto de aserciones de instanciación se conoce como la Abox. Se puede entonces razonar con respecto a la consistencia de estos individuos.

En el caso de UML, el metamodelo es la guía usada para definir el dominio o TBox, y a continuación, los modelos de usuarios se traducen como individuos o ABox. La ventaja de este enfoque es que existen varias implementaciones de este tipo de sistema tales como Racer [7], que pueden ser integrados con herramientas UML para manipular formalmente los modelos.

Dicho de otra manera, es posible representar cada modelo UML formalmente utilizando Description Logics, base sobre la cual es posible razonar y realizar consultas que nos permiten conocer la existencia de inconsistencias. En [14], se ha definido un conjunto de 18 relaciones de consistencia entre los diagramas de clase, de secuencia y de estado que no son reforzadas por la definición del metamodelo de UML.

2.1.2 Formalizando un modelo UML en DL

La Figura 2 muestra la relación que existe entre el metamodelo de UML y los modelos creados por usuarios. Esta figura solo muestra una pequeña parte del metamodelo, que en este caso incluye algunas de las metaclases y meta-asociaciones necesarias para la especificación de diagramas de clase, secuencia y componentes. El metamodelo es especificado en su totalidad usando diagramas de clase. Cada elemento creado en un modelo de usuario instancia la metaclase correspondiente. Por ejemplo, en el diagrama de clase, `CarroDeCompras` es una clase, así que instancia la metaclase `Class`. La asociación entre las clases `CarroDeCompras` y `Cliente` es

una instancia de la metaclase `Association`. La relación de generalización entre las clases `Cliente` y `ClientePreferente` instancia la metaclase `Generalization`.

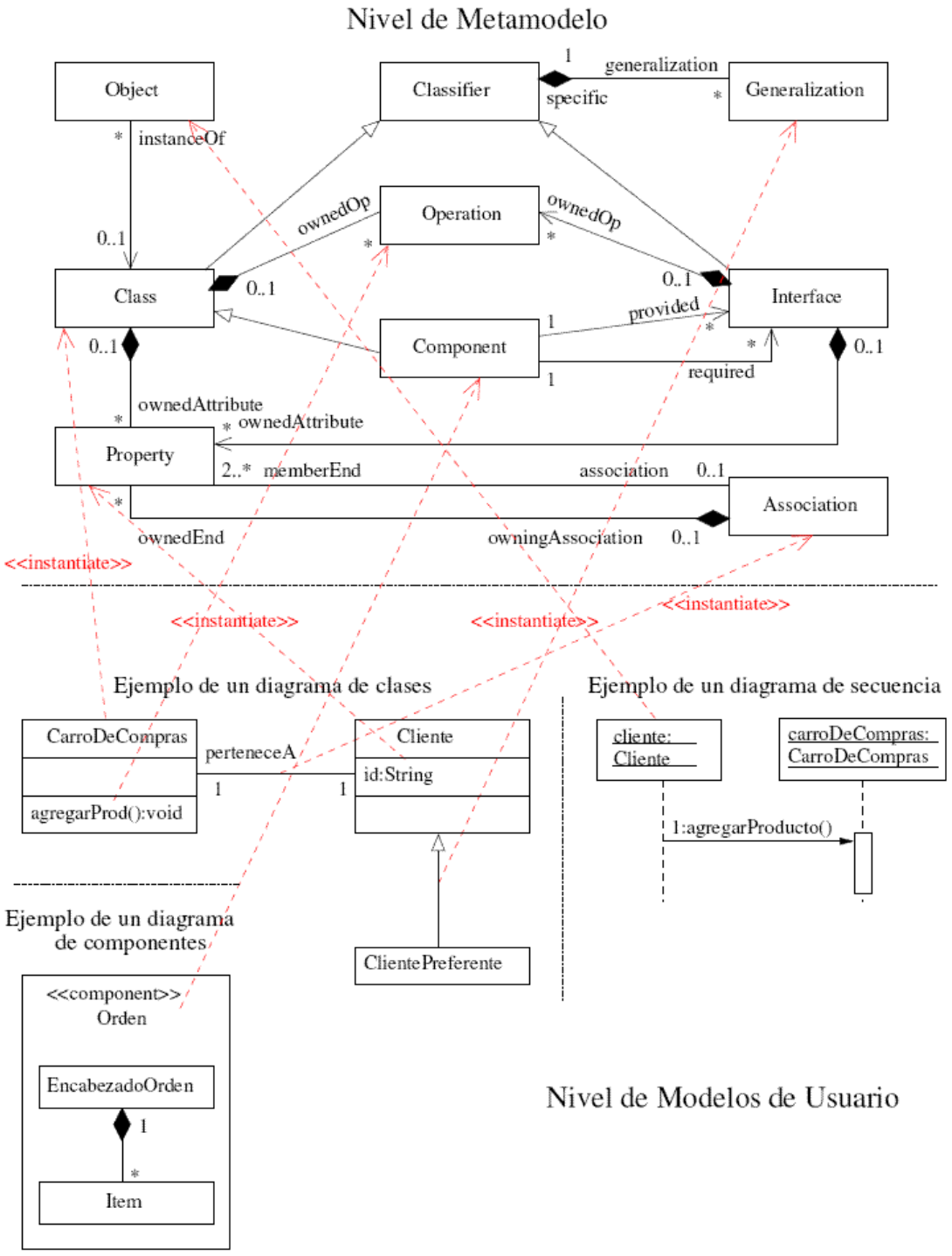

Figura 2 : Relación entre el metamodelo UML y modelos de usuario

Las relaciones de `<<instantiate>>` (instanciación) también se aplican a los elementos usados en los otros diagramas ofrecidos por UML. Por ejemplo, en el diagrama de componentes, la componente `Orden` es instancia de la metaclase `Component`. Las clases interiores de la componente, las encargadas de implementar los servicios ofrecidos por la componente, se rigen por el mismo subconjunto del metamodelo que los diagramas de clase. En el diagrama de secuencia, los objetos instancian la metaclase `Object`. Las propiedades que tiene cada elemento usado en un modelo de usuario está definido por la metaclase correspondiente. Dado el tamaño y complejidad del metamodelo, este está subdividido en paquetes, definiéndose en forma concreta cuáles paquetes son usados por cuáles diagramas. Más detalles sobre el metamodelo se pueden obtener en la especificación de UML.

En la figura 3 se muestra cómo se traduce una parte del metamodelo mostrado en la Figura 2 a DL, para ser usado como parte de la definición del dominio (Tbox) necesario para poder razonar acerca de modelos y diagramas en UML. Por ejemplo, existe la metaclase `ModelElement`, de la cual heredan todas las metaclases, y tiene un atributo `name`. Entonces, se define el concepto `ModelElement` que tiene el atributo `name`. Por definición, un `Model` es un `ModelElement`, dado que la metaclase `Model` hereda de `ModelElement`, por lo que la definición del concepto `Model` hereda la definición del atributo `name` al especificar que un `Model` es un `ModelElement`. Además, existe una asociación (`owned-element`) entre las metaclases `Model` y `ModelElement`, dado que un modelo contiene elementos. Esto se traduce como el rol `owned-element` entre los dos conceptos, especificada de la siguiente forma: `(all owned-element ModelElement)`. Esto significa que toda instancia que se relacione con una instancia del concepto `Model` a través de este rol debe ser instancia del concepto `ModelElement`.

```
; Definicion del concepto ModelElement
(implies ModelElement (a name))

; Definicion del concepto Model
(implies Model
   (and ModelElement (all owned-element ModelElement)))

; Definicion del concepto Class
(implies Class
  (and ModelElement
    (a isAbstract)
      (a isLeaf)))

; Definicion del concepto Object
(implies Object
  (and ModelElement
    (exactly 1 instance-of)
      (all instance-of Class)))
```

Figura 3 : Tbox parcial correspondiente al metamodelo de UML

De la misma forma, la metaclase `Class` hereda de `ModelElement`, definiendo además dos nuevos atributos: `isAbstract` y `isLeaf`. En la definición del concepto `Object`, restricciones cualificadas de rol se usan para forzar la multiplicidad establecida en el metamodelo para la asociación entre `Class` y `Object` (de que un objeto es instancia de exactamente una clase). Esto se traduce como (`exactly 1 instance-of`).

En la Figura 4 se muestra la traducción de una parte del diagrama de clase de ejemplo que se muestra en la figura 1, para que pase a formar parte del Abox. Todos los diagramas pertenecen a un modelo llamado modelo1, así que se crea un individuo que instancia el concepto `Model` y se asigna “modelo1” como valor del atributo `name`. El segundo conjunto de definiciones corresponde a la traducción de la clase `Cliente`. Lo primero que se hace nuevamente es crear un individuo que instancia al concepto `Class`. A continuación, se asigna el valor “Cliente” al atributo `name`. Después, se relaciona la clase con el modelo al que pertenece usando la relación `owned-element`. Finalmente, se llenan los valores de los atributos definidos para el concepto `Class` - en este caso, de que la clase `Cliente` no es abstracta ni es una hoja (es decir, tiene subclases).

El último conjunto de definiciones corresponde a la traducción del objeto `unCarroDe-Compras`. El atributo `name` del individuo es inicializado con el valor “unCarroDeCompras”. Este

objeto también pertenece a modelo1, así que también se relaciona usando el rol `owned-element`. La definición de este rol impone la restricción de que los participantes sean instancias de `ModelElement`, lo cual se cumple en este caso porque el concepto `ModelElement` subsume `Class` y `Object`. En la última línea se registra la relación entre la clase `CarroDeCompras` y la instancia `unCarroDeCompras`, usando el rol `instance-of`.

```
; Individuo que representa al modelo1
(instance inst-modelo1 model)
(constrained inst-modelo1 name-of-modelo1 name)
(constraints (string= name-of-modelo1 ”modelo1”))

; Individuo que representa la clase Cliente
(instance inst-Cliente class);
(constrained inst-Cliente name-of-Cliente name)
(constraints (string= name-of-Cliente C¸ liente”))
(related inst-modelo1 inst-Cliente owned-element)
(constrained inst-Cliente abstract-Cliente isAbstract)
(constraints (string= abstract-Cliente ”false”))
(constrained inst-Cliente leaf-Cliente isLeaf)
(constraints (string= leaf-Cliente ”false”))

; Individuo que representa al objeto unCarroDeCompras
(instance inst-unCarroDeCompras object)
(constrained inst-unCarroDeCompras name-of-unCarroDeCompras name)
(constraints (string= name-of-unCarroDeCompras ¨unCarroDeCompras”))
(related inst-modelo1 inst-unCarroDeCompras owned-element)
(related inst-unCarroDeCompras inst-CarroDeCompras instance-of)
```

Figura 4 : Abox parcial Figura 2

El orden de aparición de los individuos en las definiciones de los individuos no importa, dado que los sistemas basados en DL pueden manejar información incompleta. Información adicional puede ser ingresada posteriormente, mientras sea consistente con la información que ya se conoce. Por ejemplo, se podría colocar la definición de un individuo que representa a la clase `CarroDeCompras` a continuación de la definición del objeto que lo utiliza. Dadas las restricciones impuestas en la Tbox, el sistema deduce en este caso que el individuo `inst-CarroDeCompras` instancia al concepto `Class`, dado que el individuo `inst-unCarroDeCompras` asegura estar relacionado con este individuo a través del rol `instance-of`.

2.2 Model Consistency Checker (MCC)

Model Consistency Checker, como fue mencionado al introducir y motivar el estudio, fue desarrollado como parte de un trabajo de memoria finalizado en enero de 2005. Al finalizar dicho trabajo, MCC consistía básicamente en un sistema que aprovechaba por una parte los servicios ofrecidos por Poseidón para la creación de modelos UML, y por otra los ofrecidos por Racer para el racionamiento lógico. De esta manera se lograba contar con un framework que permitía chequear algunas inconsistencias en modelos UML.

2.2.1 Poseidon for UML

Poseidon for UML es una herramienta de modelamiento UML desarrollada por gentleware para el análisis, diseño y documentación en el proceso de desarrollo. Para ello posee una interfaz gráfica que permite modelar rápidamente y de manera intuitiva (ver Figura 5 y Figura 6). Es desarrollado en Java y permite la creación de plug-ins que son integrados en la herramienta.

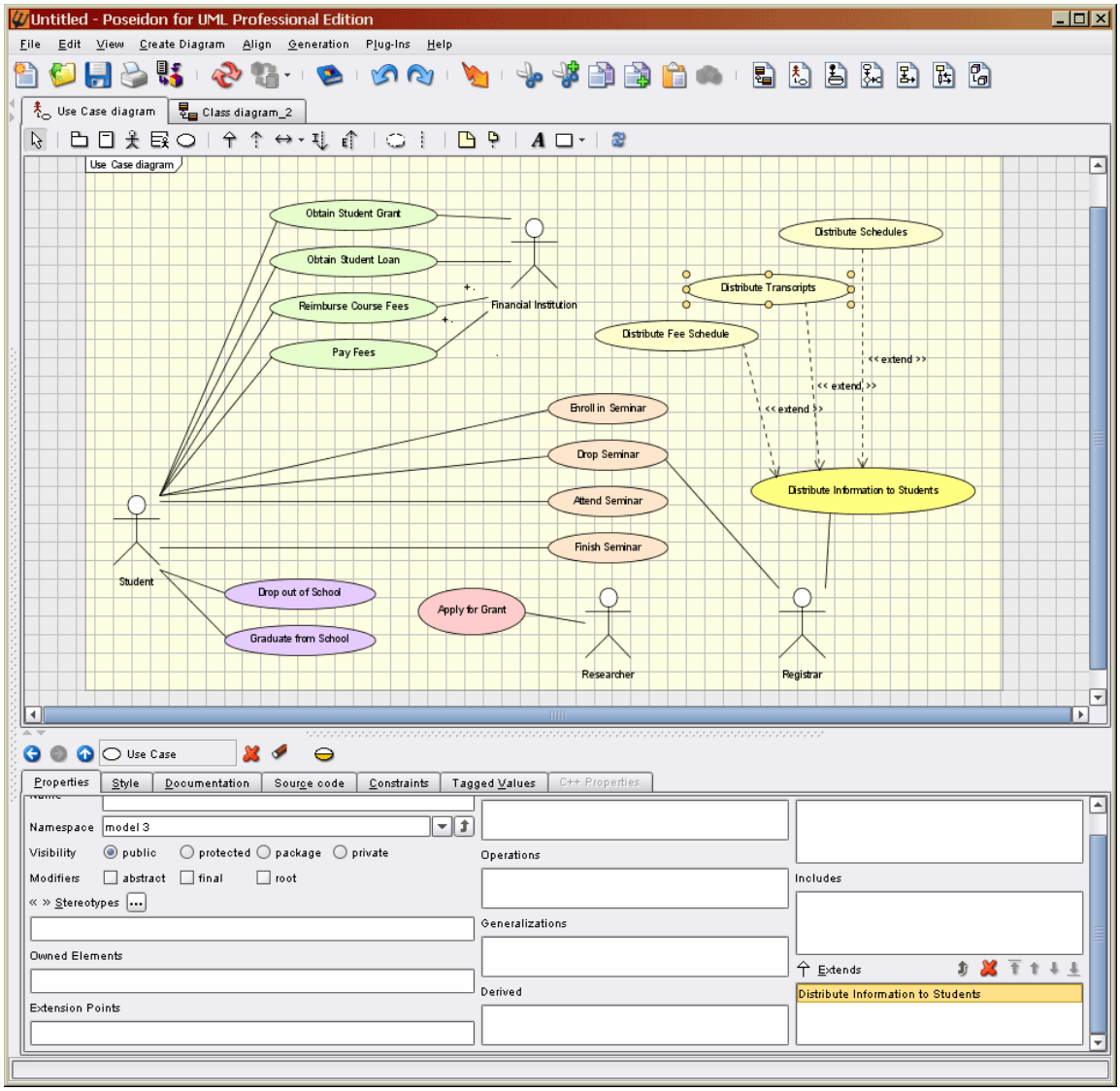

Figura 5 : Poseidon for UML – Diagrama de Casos de Uso

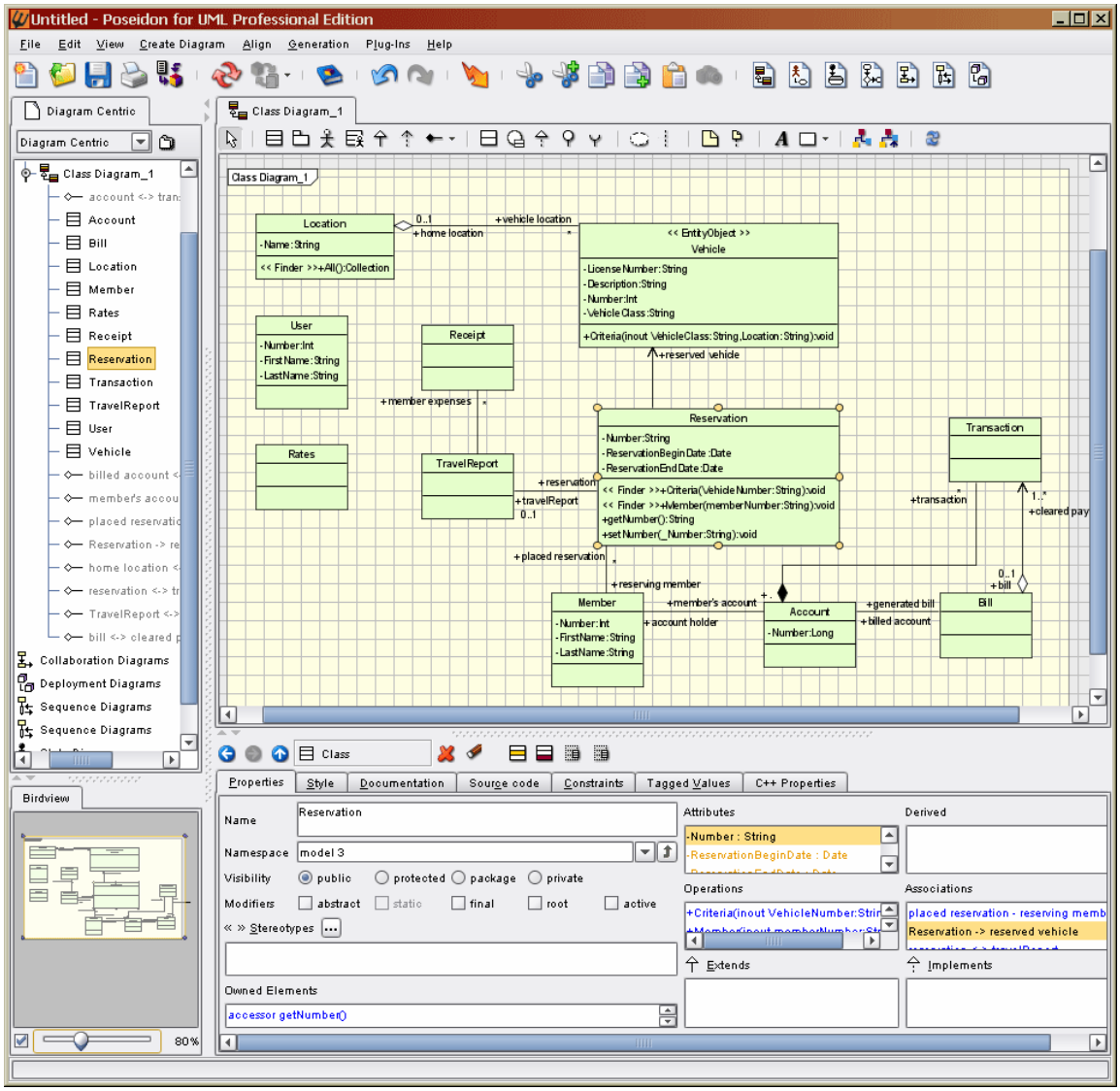

Figura 6 : Poseidon for UML – Diagrama de Clases

Pues bien, MCC es un plug-in de Poseidon. Su interacción consiste básicamente en tomar los modelos creados por el usuario en dicha aplicación y traducirlos a instancias del metamodelo, tal como se lo define en [6]. Posteriormente, mediante una interfaz gráfica integrada a Poseidón (ver Figura 7), el usuario es capaz de realizar peticiones de chequeos que son procesadas por MCC. El framework se encarga de realizar internamente las consultas pertinentes al motor de inferencia lógica y de finalmente mostrar los resultados mediante la interfaz gráfica descrita.

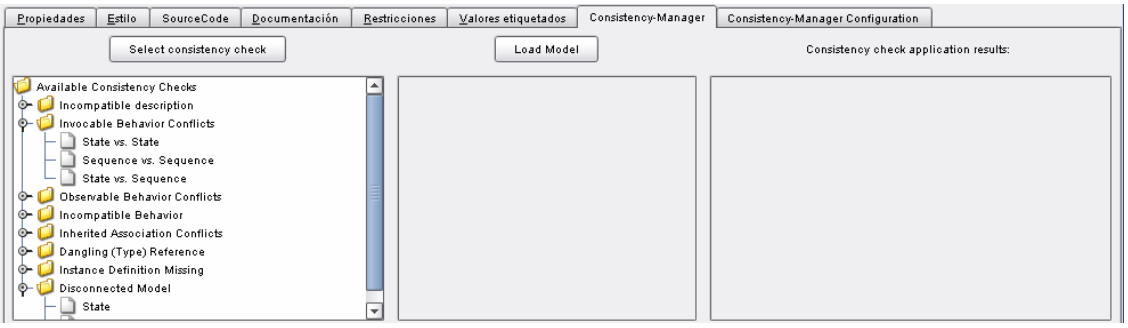

Figura 7 : MCC – Selector de Chequeos de Consistencia

2.2.2 Racer

Racer (por **R**enamed **A**Box and **C**oncept **E**xpression **R**easoner) es un sistema de representación del conocimiento (SRC) que ofrece por un lado un lenguaje de definición de terminologías muy expresivo, y por otro algoritmos de clasificación muy eficientes basados en los denominados algoritmos de “tableau” [1]. Ofrece una variante de DL conocida como ALCQHIR+ o también como SHIQ que en pocas palabras es una variante de la familia DL con un lenguaje de especificación de conceptos definido.

Su nombre comercial es RacerPro y cuenta además con una serie de herramientas que buscan simplificar su uso tales como RacerPorter, una interfaz de usuario gráfica que se puede apreciar en la Figura 8, o RacerPlus que es lo que denominan un banco de trabajo (workbench) basado en RacerPro. Por último, es posible obtener algunas APIs para RacerPro que permiten la comunicación con Lisp o Java, pero no son soportadas por Racer-Systems

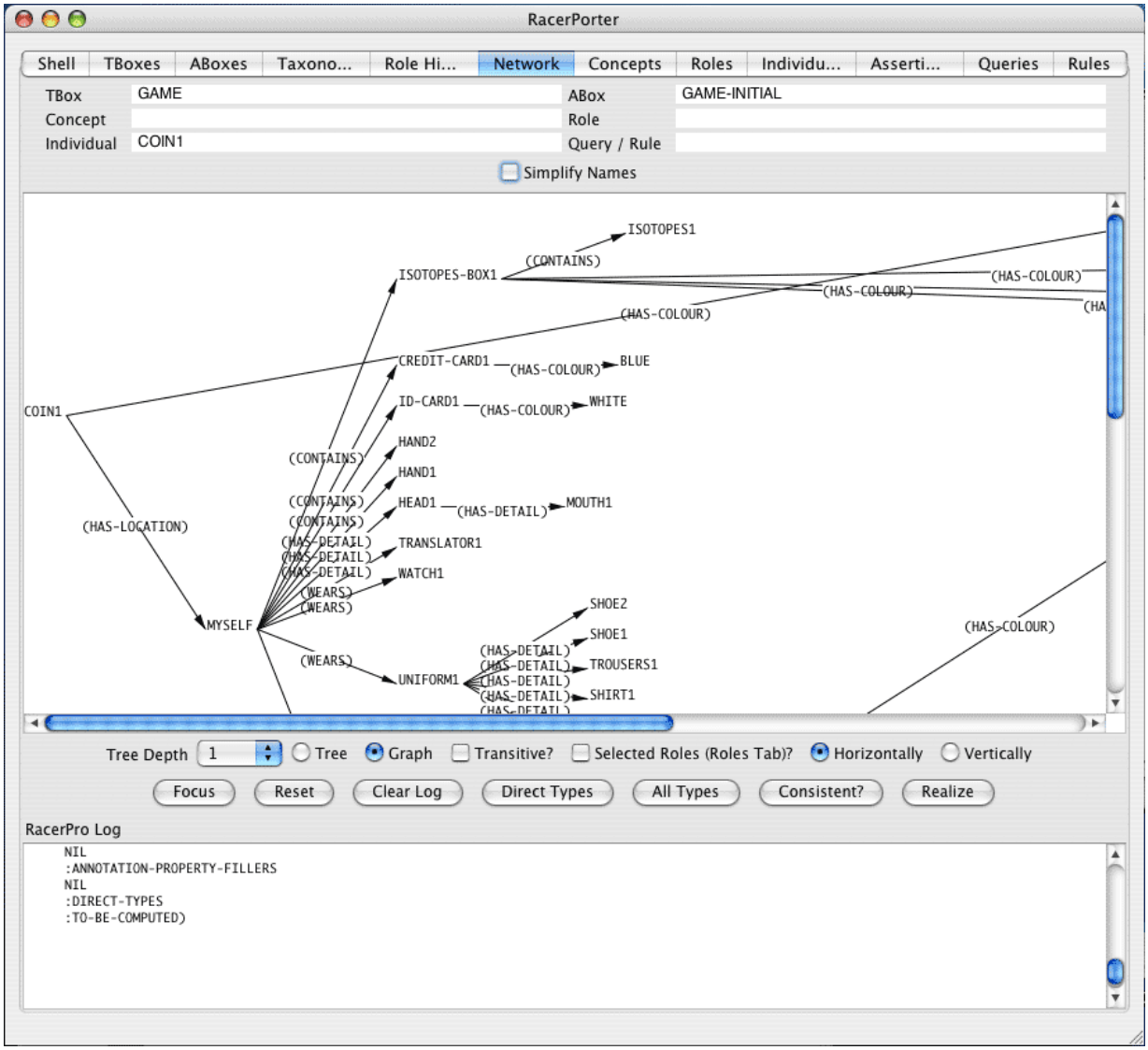

Figura 8 : RacerPorter – La interfaz de usuario gráfica de RacerPro

Como fue descrito anteriormente, Racer era el motor de inferencia lógica elegido para funcionar con MCC. La comunicación con dicho motor se realiza mediante una conexión TCP/IP, y consiste básicamente en simular una consola vía dicho protocolo, es decir debían enviarse líneas de comando que permitían crear instancias de elementos correspondientes del metamodelo, y posteriormente realizar consultas escritas en lenguaje nRQL [8] en el momento que el usuario requiera conocer la existencia de alguna inconsistencia. Las respuestas a dichas consultas son enviadas a través de la misma interfaz y son traducidas por el sistema buscando patrones en el texto de acuerdo a la respuesta esperada y mostradas al usuario de una manera más amigable.

Para establecer la comunicación descrita anteriormente, era necesario configurar previamente el framework, por lo que se incluía un panel en Poseidon (ver Figura 9) en el que era necesario especificar la ruta del archivo Racer que contenía el metamodelo, además del puerto y la dirección IP del servidor Racer.

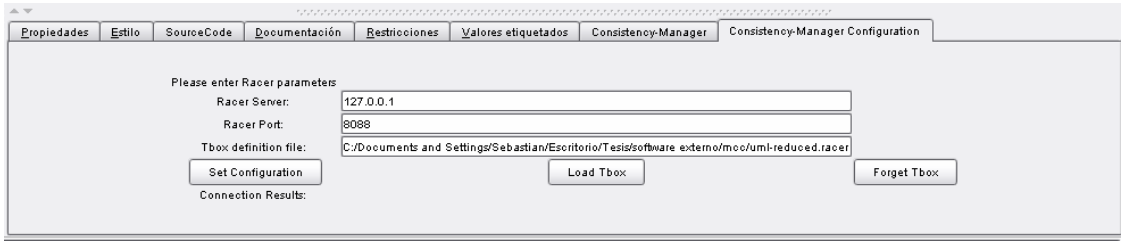

Figura 9 : MCC – Panel de Configuración

2.2.3 Chequeos Implementados

El trabajo presentado por Jocelyn Simmonds implementaba cinco de los 18 chequeos de consistencia detectados en [14], los cuales eran:

- Objeto abstracto
- Comportamiento no compatible (estado versus secuencia)
- Multiplicidades
- Instancias sin definición de clase
- Conflictos observables de comportamiento (entre máquinas de estado relacionadas por herencia).

Además, las definiciones de dichas relaciones de consistencia cumplían con la última especificación de UML (2.0) y todo el modelo UML era traducido a instancias del metamodelo implementado en el motor de inferencia lógica.

2.3 Dificultades herramienta MCC

Al iniciar el presente trabajo, MCC contaba con diversas dificultades que hacían imposible su uso. Dichas dificultades son detalladas a continuación:

2.3.1 Dificultades Poseidon

Si bien la interacción con Poseidón era un punto resuelto en el trabajo desarrollado por Jocelyn Simmonds, y en Enero de 2005 funcionaba de manera correcta, al momento de iniciar el presente trabajo era imposible instalar el framework en dicha aplicación debido principalmente a que la versión 2.6 había sufrido varias actualizaciones y modificaciones. Debido a dichas actualizaciones, la versión actual de Poseidon correspondía a la 4.1.2. Posteriormente dicha versión fue nuevamente actualizada, con lo que la última versión corresponde a la 5.0.

Sin embargo, es necesario agregar que es posible descargar la versión de Poseidon (2.6) que funcionaba con el framework. El problema es que no es viable obtener una licencia válida para esta. Por todo ello MCC resultaba sencillamente obsoleto en este aspecto.

Otra dificultad encontrada es que esta herramienta desarrollada por gentleware se volvió comercial y no es posible conseguir una licencia gratuita más que por 30 días a modo de prueba.

2.3.2 Dificultades Racer

Similar a lo ocurrido con Poseidon, Racer también sufrió modificaciones desde Enero de 2005 y al momento de iniciar el trabajo no era posible realizar los chequeos implementados, con lo cual la interacción con MCC debía ser estudiada en profundidad y adaptada para funcionar de manera adecuada con la versión actual. Además también se volvió comercial, aunque a diferencia de Poseidon, es posible conseguir licencias gratuitas para el desarrollo de estudios académicos.

2.3.3 Dificultades propias de MCC

En el último periodo de trabajo de Jocelyn Simmonds, desafortunadamente el código fuente de MCC fue perdido, con lo que al iniciar este trabajo se contaba con sólo una porción de lo presentado. Específicamente, el código fuente contaba con sólo 2 chequeos implementados (Objeto Abstracto y Comportamiento no compatible) y con el traductor UML-Racer incompleto que sólo constaba del traductor de los diagramas de clase y secuencia.

Por otro lado, MCC fue desarrollado utilizando Java 1.4 [18] que si bien sigue siendo una tecnología soportada por Sun Microsystems, hoy en día es obsoleta considerando la existencia de Java 5 [19] y la próxima aparición de Java 6 [20]. Por esto es recomendable actualizar dicho código para evitar posibles conflictos posteriores debido a versiones de la maquina virtual de Java.

3 Cambios en MCC

El trabajo fue realizado de la siguiente manera. Lo primero fue actualizar el contenido existente de tal manera de lograr que funcione con las actualizaciones de sus componentes. A continuación, y basado en la experiencia adquirida en la primera etapa, se propondrían cambios, que posteriormente serían analizados y de ser viables implementados de manera de lograr una mejor aplicación.

3.1 MCC con versiones actuales de Racer y Poseidon

3.1.1 Funcionamiento RacerPro 1.9

Si bien Racer cambió su versión posterior al trabajo realizado en enero de 2005, sus mejoras no afectan directamente la comunicación existente con MCC. Esto último debido a que las mejoras estuvieron principalmente por el lado del rendimiento, la corrección de errores, la adopción de nuevos estándares tales como SWRL [28] y mejoras en RacerPorter [12] que es la interfaz gráfica nativa de RacerPro. La lista detallada de los cambios entre versiones es visible en [13].

Por otra parte el protocolo de transmisión utilizado para establecer la comunicación entre MCC y Racer sigue siendo el mismo (TCP/IP), por lo que los cambios se redujeron al mínimo y se trató de pequeñas correcciones debido a redundancias en el carácter de fin de línea. Además, se cambiaron algunas líneas de código Java de manera de evitar advertencias debido al cambio a Java 5 mencionado anteriormente.

Sin embargo, y tras corregir los errores descritos anteriormente, se produjeron una serie de problemas de comunicación entre Racer y MCC que se repetían entre Racer y RacerPorter. Lo anterior hizo sospechar de un problema propio de Racer que tras consultar directamente con soporte, y descartar posibles orígenes como son el firewall u otros programas, fue atribuido a problemas propios del Sistema Operativo Windows XP, que si bien es soportado por la aplicación, genera conflictos debido a algunas actualizaciones. Es importante destacar este problema, ya que son el origen de un cambio en el motor de inferencia propuesto y analizado posteriormente.

3.1.2 Funcionamiento Poseidon 4.1.2 y Poseidon 5.0

Si bien Poseidon ha cambiado en diversos aspectos, entre los que se cuentan mejoras en el aspecto gráfico, optimización, nuevas funcionalidades, etc., la mayoría de estos cambios no afectan a la integración con MCC. La principal diferencia que afecta dicha integración es debido al cambio de Java 1.3 o 1.4 a Java 5, con lo cual las clases dejaron de funcionar debido a que existían nombres de variables prohibidas y cambios en los paquetes de Poseidon que hicieron imposible ejecutar MCC. No obstante, tras lograr ejecutar MCC en la última versión de Poseidon, se presentaron una serie de problemas debido a que internamente Poseidon ahora representa de manera distinta los objetos UML. Por ejemplo antes un mensaje en un diagrama de secuencia era representado por la clase `Stimulus` del package `org.omg.uml.behavioralelements.commonbehavior`, en cambio ahora se representa por la clase `Message` del package `org.omg.uml2.interactions2`, lo cual trae consigo una serie de cambios en la forma de obtener los datos necesarios para posteriormente crear las relaciones en Racer. En este aspecto cabe mencionar que se encontraron grandes dificultades debido a la nula documentación existente de dichos cambios en Poseidon, a la falta de un API (del inglés Application Programming Interface - Interfaz de Programación de Aplicaciones) completa del paquete uml2, y a la falta de respuesta del soporte de Poseidon.

3.2 Sugerencias de cambio

Es posible notar a esta altura que uno de los principales problemas de la aplicación, es que tiene dos dependencias externas que deben funcionar correctamente para que este mismo lo haga. Es por ello que resulta lógico querer eliminar alguna de ellas para contar con un programa más autocontenido, es decir que permita depender tan sólo de una aplicación externa y no de dos como lo es ahora.

Pues bien, basado en la experiencia adquirida durante la actualización de MCC, fue posible ver que el problema más frecuente que se presenta en la aplicación es debido a problemas con la comunicación de MCC con Racer. De hecho, para lograr dicha comunicación, es necesario permitir el paso a través de los posibles firewalls, prohibir las actualizaciones automáticas en el caso de usar Windows, y configurar MCC de manera adecuada, para lo cual se debe conocer la ruta en la que se encuentra el archivo racer con el metamodelo, además del número IP y el puerto en los que se está ejecutando Racer. Todos estos puntos son irrelevantes para el uso común de la

aplicación, y sin embargo deben ser de dominio del usuario final para lograr que MCC se ejecute correctamente.

Por lo anterior se propuso cambiar la dependencia con Racer en nuestra aplicación, y por otro lado se vuelve necesario esconder de manera definitiva al usuario final la existencia de un motor de inferencia lógica, quién en la mayoría de los casos no se interesa más que en las posibles inconsistencias de sus modelos.

3.2.1 Alternativas de motores de inferencia lógica

Pensando en una solución que permita esconder de manera definitiva la existencia de un motor de inferencia lógica en nuestra aplicación, las primeras alternativas descartadas fueron aquellas desarrolladas en lenguajes de programación distintos al de nuestra aplicación. Lo anterior debido a que la idea final es incluir como parte de nuestra aplicación un conjunto de librerías externas que cumplan ese rol.

Es necesario dejar en claro que el hecho de cambiar de motor de inferencia implicaba también un cambio profundo en la aplicación, esto debido a que Racer utiliza un lenguaje propio para realizar consultas e inserciones en su ABox llamado nRQL [8] (new Racer Query Language), el cual no es implementado más que por el propio Racer.

Los motores que cumplían con poder incluirse mediante una biblioteca en nuestra aplicación, y que fueron analizados en profundidad son:

- Pellet [2]: Es un razonador OWL-DL [23] de código abierto escrito en Java. Originalmente fue desarrollado en los laboratorios Mindswap de la Universidad de Maryland, y fundado por un diverso grupo de organizaciones. Pellet está basado en los llamados “tableaux algorithms” [1] desarrollados para Description Logics (DL). Soporta la expresividad completa de OWL-DL incluyendo el razonamiento acerca de nominales (clases enumeradas). Además, en su versión 1.4 Pellet soporta todas las especificaciones propuestas en OWL 1.1, con la excepción de los tipos de datos n-arios (n-ary datatypes). Pellet incluye un motor de consultas en el ABox. Dichas consultas pueden ser formuladas utilizando SPARQL. Pero solo aquellas consultas que corresponden a ABox conjuntivas (conjunctive ABox) son soportadas. Estas son consultas SPARQL que satisfacen las siguientes tres condiciones:

    1. No contienen variables en la posición del predicado.

2. Cada propiedad utilizada en la posición del predicado es cualquiera (datatype o objeto) propiedad definida en la ontología de una de las siguientes propiedades: `rdf:type`, `owl:sameIndividualAs`, `owl:differentFrom`.
3. Si `rdf:type` es utilizado en la posición del predicado, una constante URI denotando una clase OWL (o una clase expresión) es utilizada en la posición del objeto

- KAON2[10]: Es una infraestructura para manejar las ontologías OWL-DL[23], SWRL[28] y F-Logic[5], que provee las siguientes funcionalidades:
    1. Un API para el desarrollo y mantención de ontologías OWL-DL, SWRL, y F-Logic vía programación.
    2. Un servidor que provee acceso a las ontologías de manera distribuida utilizando RMI (Remote Method Invocation del inglés Método de Invocación Remota)
    3. Un motor de inferencia para responder consultas conjuntivas expresadas utilizando la sintaxis SPARQL
    4. Una interfaz DIG[4], que permite el acceso desde herramientas tales como Protégé[17]
    5. Un modulo que permite extraer instancias de una ontología desde una base de datos relacional
    6. Como fue descrito anteriormente las consultas pueden ser formuladas utilizando SPARQL. Pero no toda la especificación es soportada. Tal y como fue dicho solo aquellas consultas que corresponden a consultas conjuntivas son soportadas, ya que para el resto es necesario aplicar lógica de segundo orden. Además, los patrones OPTIONAL y GRAPH no son soportados porque son difíciles de formalizar en lógica.
- Jena [9]: Jena es un framework hecho en Java para la construcción de aplicaciones Web Semánticas [26]. Está desarrollado como código abierto, y incluye las siguientes funcionalidades:

1. Un API para RDF.
2. Lectura y escritura de archivos RDF en RDF/XML, N3 y N-Triples
3. Un API para OWL
4. Almacenamiento en memoria y persistente
5. Un motor de inferencia lógica consultado utilizando SPARQL

Como se puede observar los motores mencionados tienen bastantes similitudes. Las principales diferencias encontradas fueron que Pellet no ofrece un API para desarrollar ontologías OWL-DL, y Jena a diferencia de los otros motores, soporta el lenguaje SPARQL de manera completa. Por lo anterior se decidió utilizar Jena como motor de inferencia lógica.

3.3 Mejoras en MCC

Los principales cambios aplicados a MCC consisten en eliminar la dependencia con Racer de manera de lograr una aplicación más autocontenida y que solucione en gran medida los problemas que tiene para su uso actual, además se pretende eliminar la necesidad de configuraciones y de esta manera lograr mayor facilidad para el usuario final. Con este objetivo se utilizará un nuevo motor de inferencia de nombre Jena [9], desarrollado en Java y que utiliza tecnologías estándares como OWL-DL [23] y SPARQL [27].

3.3.1 RDF y OWL-DL

RDF [24] o Marco de Descripción de Recursos (del inglés Resource Description Framework) es básicamente un framework para metadatos, desarrollado por el World Wide Web Consortium (W3C). Este modelo se basa en la idea de convertir las declaraciones de los recursos en expresiones con la forma sujeto-predicado-objeto (conocidas en términos RDF como tripletes). El sujeto es el recurso, es decir aquello que se está describiendo. El predicado es la propiedad o relación que se desea establecer acerca del recurso. Por último, el objeto es el valor de la propiedad o el otro recurso con el que se establece la relación. Por ejemplo en la declaración “Los alumnos del DCC tienen código 059”, “Los alumnos del DCC” sería el sujeto RDF, “tienen código” el predicado y “059” el objeto.

OWL (Web Ontology Language [23]) o Lenguaje de Ontologías para la Web, es una recomendación W3C desde el 10 de febrero de 2004. En realidad, es una extensión del lenguaje RDF y emplea sus tripletas, aunque es un lenguaje con más poder expresivo que éste. OWL está diseñado para usarse cuando cierto tipo de información necesita ser procesada por programas o aplicaciones, en oposición a situaciones donde el contenido solamente necesita ser presentado. Puede usarse para representar explícitamente el significado de términos en vocabularios y las relaciones entre aquellos términos. Esta representación de los términos y sus relaciones se denomina una ontología.

OWL se basa o evoluciona a partir de una serie de recomendaciones previas W3C, de la siguiente manera:

- XML: provee una sintaxis superficial para documentos estructurados, pero no impone restricciones semánticas en el significado de estos documentos.

- XML Schema es un lenguaje para restringir la estructura de los documentos XML y también extiende XML con tipos de datos.
- RDF es un modelo de datos para objetos ("recursos") y para las relaciones entre ellos, provee una semántica simple para este modelo de datos, a la vez que este modelo de datos puede ser representado en sintaxis XML.
- RDF Schema es un vocabulario para describir propiedades y clases de recursos RDF, con una semántica para generalización de jerarquías de aquellas propiedades y clases.
- OWL añade más vocabulario para describir propiedades y clases: entre otras, relaciones entre clases (ejemplo, inconexas), cardinalidad (ejemplo "exactamente uno"), igualdad, más ricos tipos de propiedades, características de las propiedades (por ejemplo, simetría), y clases enumeradas.

OWL ofrece tres sub-lenguajes de expresión incremental diseñados para ser usados por comunidades específicas de desarrolladores y usuarios según el nivel de expresividad que precisen éstos.

- OWL Lite: Utilizado por usuarios que primordialmente necesitan una clasificación jerárquica y restricciones simples. Por ejemplo, soporta restricciones cardinales, pero solamente permite valores cardinales de 0 ó 1. Así pues, es más simple proveer herramientas de soporte para OWL Lite. OWL Lite ofrece una rápida ruta de migración para tesauros y otras taxonomías. En resumen, OWL Lite tiene una más baja complejidad formal que OWL DL.
- OWL DL: Utilizado por usuarios que quieren la máxima expresividad mientras conservan completamente la computacionalidad (todas las conclusiones son garantizadas para ser computables) y resolubilidad (todas las computaciones terminarán en tiempo finito). OWL DL incluye todos los constructos del lenguaje OWL, pero pueden usarse solamente bajo ciertas restricciones (por ejemplo, mientras una clase puede usarse por una subclase de muchas clases, una clase no puede ser una instancia de otra clase). OWL DL se denomina así debido a su correspondencia con Description Logic (DL).
- OWL Full: Utilizado por usuarios que requieren el máximo de expresividad y la libertad sintáctica de RDF sin garantías computacionales. Por ejemplo, en OWL

Full una clase puede ser tratada simultáneamente como una colección de individuos y como un individuo por derecho propio. OWL Full permite a una ontología aumentar el significado del vocabulario predefinido (RDF ó OWL). Es poco probable que algún software racional pueda soportar por completo el razonamiento para cada característica de OWL Full.

Pues bien, debido a que MCC utiliza Description Logic para el razonamiento, la versión de OWL utilizada por la aplicación corresponde a OWL DL. Para esto fue necesario traducir todos los elementos existentes en Racer. Para explicar la manera en que se efectúa dicha traducción, es necesario volver sobre el ejemplo en la sección 2.1.2., para el cual se tenía la estructura especificada en la Figura 10 para el Tbox.

```
; Definicion del concepto ModelElement
(implies ModelElement (a name))

; Definicion del concepto Model
(implies Model
   (and ModelElement (all owned-element ModelElement)))

; Definicion del concepto Class
(implies Class
  (and ModelElement
    (a isAbstract)
      (a isLeaf)))

; Definicion del concepto Object
(implies Object
  (and ModelElement
    (exactly 1 instance-of)
      (all instance-of Class)))
```

Figura 10 : Tbox parcial correspondiente al metamodelo de UML en Racer

```
<!-- Definicion del concepto ModelElement -->
<rdf:Description rdf:about="http://mcc/ModelElement">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Restriction"/>
  <owl:onProperty rdf:resource="http://mcc/atributoName"/>
  <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1
  </owl:cardinality>
</rdf:Description>

<!-- Definicion del concepto Model -->
<rdf:Description rdf:about="http://mcc/Model">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdfs:subClassOf rdf:resource="http://mcc/ModelElement"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Restriction"/>
  <owl:onProperty rdf:resource="http://mcc/owned_element"/>
  <owl:allValuesFrom rdf:resource="http://mcc/ModelElement"/>
</rdf:Description>

<!-- Definicion del concepto Class -->
<rdf:Description rdf:about="http://mcc/Class">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdfs:subClassOf rdf:resource="http://mcc/ModelElement"/>
  <rdfs:subClassOf rdf:resource="http://mcc/ClassWithIsLeaf"/>
  <rdfs:subClassOf
rdf:resource="http://workspace/mcc/ClassWithIsAbstract"/>
</rdf:Description>
<rdf:Description rdf:about="http://mcc/ClassWithIsLeaf">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Restriction"/>
  <owl:onProperty rdf:resource="http://mcc/atributoIsLeaf"/>
  <owl:cardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
</rdf:Description>
<rdf:Description rdf:about="http://mcc/ClassWithIsAbstract">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Restriction"/>
  <owl:onProperty rdf:resource="http://mcc/atributoIsAbstract"/>
  <owl:cardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
</rdf:Description>

<!-- Definicion del concepto Object -->
<rdf:Description rdf:about="http://mcc/Object">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdfs:subClassOf rdf:resource="http://mcc/ModelElement"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Restriction"/>
  <owl:onProperty rdf:resource="http://mcc/instance_of"/>
  <owl:cardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
  <owl:allValuesFrom rdf:resource="http://mcc/Class"/>
</rdf:Description>
```

Figura 11 : Tbox parcial correspondiente al metamodelo de UML en OWL-DL

En el modelo expuesto anteriormente se tenía que una metaclase `ModelElement` que tenía un atributo `name`, por lo que se definía el concepto `ModelElement` que tenía un atributo `name`. Sin embargo en esta ocasión la sintaxis es distinta como se indica en la Figura 11. Allí se indica en el primer tag XML, que se describirá (`rdf:Description`) al objeto `ModelElement` cuyo identificador es `http://mcc/ModelElement`. En el tag `rdf:type` a continuación se define dicho el elemento `ModelElement` como una Clase OWL, elemento principal de las ontologías. A continuación, se define nuevamente mediante un tag `rdf:type` la existencia de una restricción en el elemento. Dicha restricción está especificada en la propiedad (`owl:onProperty`) `atributoName` y se trata de que la cardinalidad (`owl:onProperty`) debe ser 1, es decir el elemento `ModelElement` debe tener un nombre.

Luego el concepto `Model` se define nuevamente como una clase, pero a su vez se dice subclase (`rdfs:subClassOf`) de `ModelElement`, por lo que hereda sus propiedades y restricciones, que en este caso corresponde a que debe tener un `atributoName`. Además se le asocia una restricción sobre la propiedad `owned-element` y es que todos los valores deben corresponder a instancias de la clase `ModelElement`.

La clase `Class` se comporta de manera distinta, lo que sucede es que Jena implementa todas las restricciones como tag's hijos directos del tag que describe la clase (`rdf:Description`). Por esto al describir dos restricciones, suele suceder que en realidad al momento de necesitar evaluarlas reconozca sólo una debido a la existencia de tag's iguales. Desde el punto de vista de OWL-DL esto es solucionable dado que el tag `rdf:type` que contiene el atributo `Restriction`, debe contener su descripción como tag hijos de este, tal y como lo muestra la Figura 12. De todas maneras, Jena no presenta el mismo problema con las subclases, por lo que se optó por crear superclases que contuvieran sólo las restricciones y que luego son heredadas en la clase `Class` tal y como lo muestra la figura.

Finalmente el concepto `Object` hereda de `ModelElement` de la forma explicada anteriormente, y tiene dos restricciones sobre la relación `instante-of`, la primera es que tenga solo un elemento, y la segunda es que todos los elementos que relacionen con `instante-of` sean del tipo `Class`.

```
<!-- Definicion del concepto Class -->
<rdf:Description rdf:about="http://mcc/Class">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Restriction">
    <owl:onProperty rdf:resource="http://mcc/atributoIsLeaf"/>
    <owl:cardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
  </rdf:type>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Restriction">
    <owl:onProperty rdf:resource="http://mcc/atributoIsAbstract"/>
    <owl:cardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
  </rdf:type>
</rdf:Description>
```

Figura 12 : Definición correcta del elemento Class en OWL-DL

Ya descrita la forma en que se crea el denominado Tbox o conceptos y relaciones del dominio, es necesario conocer además la manera de crear instancias o individuos. Para ello nuevamente serán referidos los individuos creados en la sección 2.1.2 para la cual se tenían las inserciones de la Figura 13

```
; Individuo que representa al modelo1
(instance inst-modelo1 model)
(constrained inst-modelo1 name-of-modelo1 name)
(constraints (string= name-of-modelo1 ”modelo1”))

; Individuo que representa la clase Cliente
(instance inst-Cliente class);
(constrained inst-Cliente name-of-Cliente name)
(constraints (string= name-of-Cliente Cliente”))
(related inst-modelo1 inst-Cliente owned-element)
(constrained inst-Cliente abstract-Cliente isAbstract)
(constraints (string= abstract-Cliente ”false”))
(constrained inst-Cliente leaf-Cliente isLeaf)
(constraints (string= leaf-Cliente ”false”))

; Individuo que representa al objeto unCarroDeCompras
(instance inst-unCarroDeCompras object)
(constrained inst-unCarroDeCompras name-of-unCarroDeCompras name)
(constraints (string= name-of-unCarroDeCompras ¨unCarroDeCompras”))
(related inst-modelo1 inst-unCarroDeCompras owned-element)
(related inst-unCarroDeCompras inst-CarroDeCompras instance-of)
```

Figura 13 : Abox en Racer

```
<rdf:RDF
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:j.0="http://mcc/">

...

<!-- Individuo que representa al modelo1 -->
<rdf:Description rdf:about="http://mcc/inst-modelo1">
  <rdf:type rdf:resource=" http://mcc/Model"/>
  <j.0:atributoName>modelo1</j.0:atributoName>
</rdf:Description>

<!-- Individuo que representa la clase Cliente -->
<rdf:Description rdf:about="http://mcc/inst-Cliente">
  <rdf:type rdf:resource="http://mcc/Class"/>
  <j.0:atributoName>Cliente</j.0:atributoName>
  <j.0:owned_element rdf:resource="http://mcc/inst-modelo1"/>
  <j.0: atributoIsAbstract>false</j.0: atributoIsAbstract>
  <j.0: atributoIsLeaf>false</j.0: atributoIsLeaf >
</rdf:Description>

<!-- Individuo que representa al objeto unCarroDeCompras -->
<rdf:Description rdf:about="http://mcc/inst- unCarroDeCompras ">
  <rdf:type rdf:resource="http://mcc/Object"/>
  <j.0:atributoName>unCarroDeCompras</j.0:atributoName>
  <j.0:owned_element rdf:resource="http://mcc/inst-modelo1"/>
  <j.0:instance_of rdf:resource="http://mcc/inst-CarroDeCompras"/>
</rdf:Description>

...

</rdf:RDF>
```

Figura 14 : Abox en OWL-DL

En este punto es necesario mencionar que tal y como RDF, OWL-DL utiliza un tag llamado `<rdf:RDF>` como la raíz del documento. Es en este tag que también se incluyen una serie de referencias a los “RDF namespace” a utilizar. Por ejemplo en la Figura 14 se utiliza la referencia `xmlns:j.0="http://mcc/"` lo cual indica que por ejemplo el tag `j.0:atributoName` está especificado en el documento encontrado en `http://mcc/`.

En OWL-DL las inserciones se realizan describiendo los nuevos elementos a través del tag `rdf:Description`, en el cual debemos darle un nombre a modo de identificación a la instancia. A continuación y como hijos del tag `rdf:Description`, se debe utilizar el tag `rdf:type` para mencionar a que clase instancia el objeto y finalmente se dan los valores de posibles relaciones o

atributos utilizando para ello el tag `j.0:nombreAtributo` o `j.0:nombreRelacion` según corresponda, cuidando de cambiar el sufijo `j.0` de acuerdo al especificado en el tag raíz. Para asignar los valores correspondientes se utiliza `rdf:resource` en caso de querer relacionar con otro objeto o instancia, o se ingresa el valor del atributo entre los tag de inicio y fin del atributo. Algunos ejemplos concretos en la Figura 14 que corresponden a las aserciones anteriormente expuestas en la Figura 13.

3.3.2 SPARQL

SPARQL es un lenguaje de consultas y un protocolo para acceder RDF diseñado por grupo de acceso a los datos RDF de la W3C [25]. Como lenguaje de consultas, SPARQL es orientado a los datos, en el sentido que solo consulta la información existente en los modelos; no hay inferencia en el lenguaje en si.

Lo primero necesario para entender el funcionamiento de SPARQL es entender los datos que consulta. Como fue descrito anteriormente, RDF se basa en la idea de convertir las declaraciones de los recursos en expresiones con la forma sujeto-predicado-objeto. Pues bien, la idea de SPARQL es realizar consultas en dichas tripletas.

Por ejemplo la Figura 15 muestra un extracto de la Figura 14 que representa un individuo en OWL-DL. Pues bien, ese individuo puede ser representado a través de tripletas de la siguiente forma: Primero el sujeto será el referido por el atributo `rdf:about`, el predicado será el determinado por el tag hijo, en un primer caso `rdf:type` y el objeto será `http://mcc/Class`, por lo tanto tenemos la tripleta :

```
http://mcc/inst-Cliente - rdf:type - http://mcc/Class
```

que indica que el individuo `ints-Cliente` es del tipo `Class`. De forma similar podemos deducir el conjunto de tripletas que representan completamente al individuo y sus relaciones que se observa en la Figura 16

```
<rdf:Description rdf:about="http://mcc/inst-Cliente">
  <rdf:type rdf:resource="http://mcc/Class"/>
  <j.0:atributoName>Cliente</j.0:atributoName>
  <j.0:owned_element rdf:resource="http://mcc/inst-modelo1"/>
  <j.0: atributoIsAbstract>false</j.0: atributoIsAbstract>
  <j.0: atributoIsLeaf>false</j.0: atributoIsLeaf >
</rdf:Description>
```

Figura 15 : Individuo en OWL-DL

```
http://mcc/inst-Cliente rdf:type                http://mcc/Class
http://mcc/inst-Cliente j.0:atributoName        Cliente
http://mcc/inst-Cliente j.0:owned_element       http://mcc/inst-modelo1
http://mcc/inst-Cliente j.0: atributoIsAbstract false
http://mcc/inst-Cliente j.0: atributoIsLeaf     false
```

Figura 16 : Individuo en N-Triples

Ya teniendo claro el dominio sobre el que estamos realizando consultas veamos un ejemplo de consulta en la Figura 17 considerando como dominio las tripletas representadas en Figura 16.

```
PREFIX j.0: <http://mcc/>

SELECT ?name
WHERE { <http://mcc/ins-Cliente> j.0:atributoName ?nombre }
```

Figura 17 : Consulta SPARQL

Ejecutando dicha consulta en el dominio especificado se obtendría el resultado que aparece en la Figura 18.

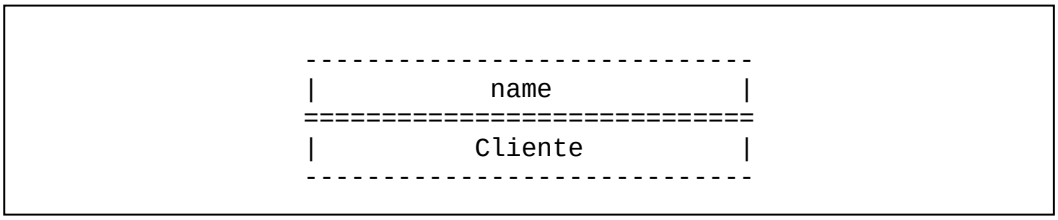

Figura 18 : Resultado consulta SPARQL de Figura 17 en el dominio representado por la Figura 16

SPARQL funciona igualando el patrón triple que se encuentra en la cláusula `WHERE` con las tripletas del gráfico RDF. Para el caso de ejemplo, el sujeto y el predicado de nuestra tripleta son valores fijos, por lo que el patrón permitirá solo tripletas con solo esos valores. El objeto es una variable y no hay otras restricciones, por lo que los patrones que calzarán con nuestra consulta son aquellos para en los que coincidan el sujeto y predicado y el resultado serán los objetos con las soluciones para `name`. Se puede apreciar también que el símbolo '?' precede a las variables en consulta, pero no forma parte del nombre de la variable por lo que no aparece en el resultado.

Las consultas pueden contener más de un patrón en la cláusula WHERE, en cuyo caso entre las cláusulas se debe agregar el símbolo '`.`' equivalente a un "y" lógico que quiere decir que los resultados deben cumplir con el primer y el segundo patrón para ser soluciones válidas.

Existen adicionalmente un conjunto de funciones extras para permitir la concordancia de patrones tales como `regex(?x, "pattern" [, "flags"])` que funciona de manera similar a un `LIKE` en lenguaje SQL, es decir permite que los valores que coincidan con un determinado patrón (`pattern`) en la variable `?x`, sean valores permitidos en la consulta. Además es posible buscar variables que coincidan con determinadas expresiones algebraicas tales como `?x > 10`, etc.

Supongamos que en determinados casos necesitamos conocer los datos de las edades de un conjunto de personas. Supongamos además, que la información es incompleta por lo que solo se cuenta con las edades de un subconjunto de ellos. Para este tipo de casos y otros similares, es posible escribir consultas que nos permitan obtener la información de todas las personas, y en los casos en que exista la edad de ellos. Para esto, SPARQL permite en su sintaxis incluir la función OPTIONAL, que consulta por los datos, pero que no falla en caso de que estos no existan. Para el ejemplo deberíamos incluir una cláusula similar a:

```
OPTIONAL { ?persona info:edad ?edad}.
```

Para finalizar, sólo mencionar algunas opciones extras soportadas por SPARQL y de comportamiento similar a SQL tales como ORDER BY, DISTINCT y OFFSET/LIMIT.

3.3.3 Jena

Como fue mencionado anteriormente, Jena es un motor de inferencia que permite la definición de ontologías OWL-DL (entre otras) y soporta consultas SPARQL. Para ello, provee una serie de clases y métodos que permiten manipular fácilmente el modelo, manejar sus clases e individuos, crear restricciones, realizar consultas, lectura/escritura de archivos OWL-DL, y un sin fin de otras funcionalidades definidas en su API.

1. Creación del metamodelo UML en Jena

La creación de modelos en Jena se realiza a través de la clase `ModelFactory`. Ella provee acceso a la creación de diversos tipos de modelos tales como RDF, OWL, etc., para ello basta con utilizar el método `createOntologyModel` en nuestro caso, con el parámetro `OntModelSpec.OWL_DL_MEM` que especifica que crearemos un modelo OWL-DL que se manejará en memoria. Dicho método retorna una instancia de la clase `Modelo` que representará nuestra ontología. Dicha clase a su vez contiene una gran cantidad de métodos que permiten crear clases, propiedades, tipos de datos, restricciones, etc.

A modo de ejemplo se puede observar en la Figura 20 el fragmento del metamodelo representado en la Figura 19 escrito en Java utilizando el API de Jena.

```
<!-- Definicion del concepto ModelElement -->
<rdf:Description rdf:about="http://mcc/ModelElement">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Restriction"/>
  <owl:onProperty rdf:resource="http://mcc/atributoName"/>
  <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1
  </owl:cardinality>
</rdf:Description>

<!-- Definicion del concepto Object -->
<rdf:Description rdf:about="http://mcc/Object">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdfs:subClassOf rdf:resource="http://mcc/ModelElement"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Restriction"/>
  <owl:onProperty rdf:resource="http://mcc/instance_of"/>
  <owl:cardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
  <owl:allValuesFrom rdf:resource="http://mcc/Class"/>
</rdf:Description>
```

Figura 19 : Tbox parcial del metamodelo de UML en OWL-DL

```
//Primero creamos una ontología OWL-DL
OntModel umlModel =
ModelFactory.createOntologyModel(OntModelSpec.OWL_DL_MEM);
//Definimos un string que será la base de todos los elementos
String base = "http://mcc/";

//Ahora creamos la clase ModelElement a partir de un string
OntClass ModelElement = umlModel.createClass(base + "ModelElement");
//Creamos el atributo nombre y decimos que se trata de un String
DatatypeProperty atributoName= umlModel.createDatatypeProperty(base
+"atributoName");
atributoName.addRange(XSD.xstring);
//Y finalmente decimos que ModelElement tiene un name
umlModel.createCardinalityRestriction(ModelElement.getURI(),
atributoName, 1);

//Creamos la clase ontologica Object
OntClass Object = umlModel.createClass(base + "Object");
//Decimos que Object hereda las propiedades de ModelElement
Object.addSuperClass(ModelElement);
//Y finalmente creamos las restricciones sobre instance_of
umlModel.createCardinalityRestriction(Object.getURI(),instance_of, 1);
umlModel.createAllValuesFromRestriction(Object.getURI(),instance_of,
Class);
```

Figura 20 : Tbox parcial correspondiente al metamodelo de UML utilizando Jena para crear una ontología OWL-DL

2. Instanciación o creación de individuos utilizando Jena

De manera similar al caso anterior, Jena provee un API para la creación de individuos del modelo. Para ello es necesario utilizar los objetos OntClass construidos con anterioridad y solicitarles la creación de un individuo con un nombre específico. A continuación, a dichos individuos es posible agregarle los valores de sus propiedades y atributos.

Nuevamente en la Figura 21 se presenta un individuo descrito en OWL-DL que posteriormente en la Figura 22 es presentado utilizando el API de Jena.

```
<!-- Individuo que representa la clase Cliente -->
<rdf:Description rdf:about="http://mcc/inst-Cliente">
  <rdf:type rdf:resource="http://mcc/Class"/>
  <j.0:atributoName>Cliente</j.0:atributoName>
  <j.0:owned_element rdf:resource="http://mcc/inst-modelo1"/>
  <j.0: atributoIsAbstract>false</j.0: atributoIsAbstract>
  <j.0: atributoIsLeaf>false</j.0: atributoIsLeaf >
</rdf:Description>
```

Figura 21 : Individuo Clase Cliente en OWL-DL

```
OntClass Class, Model;
OntProperty owned_element;
...

Individual modelo1 = Model.createIndividual("inst-modelo1")
...


Individual cliente = Class. createIndividual("inst-Cliente");
cliente.addProperty(atributo_name, "Cliente");
cliente.addProperty(owned_element, modelo1);
cliente.addProperty(atributoIsAbstract, false);
cliente.addProperty(atributoIsLeaf, false);
```

Figura 22 : Individuo Clase Cliente en OWL-DL utilizando Jena

3. Consultas SPARQL en Jena

Similar a la creación de ontologías, Jena provee un API para la creación y ejecución de consultas SPARQL. Para ello es necesario utilizar `QueryFactory` para la creación de la consulta a partir de un `String`, y `QueryExecutionFactory` para la creación de una instancia

de ejecución de la consulta creada sobre un modelo determinado. Posteriormente y gracias al objeto retornado por `QueryExecutionFactory`, se ejecuta la consulta y se obtiene un conjunto con el resultado (`ResultSet`) que puede ser recorrido secuencialmente. Un ejemplo completo es presentado en la Figura 23 basados en la consulta de ejemplo en la Figura 17.

```
String NL = System.getProperty("line.separator");
String queryString = "PREFIX j.0: <http://mcc/>" + NL +
                  " SELECT ?name " +
                  " WHERE { " +
                  "   <http://mcc/ins-Cliente> j.0:atributoName ?nombre" +
                  " } ";
Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, umlModel) ;
try {
      ResultSet rs = qexec.execSelect() ;

      while(rs.hasNext()){
            QuerySolution rb = rs.nextSolution() ;
            Resource name = rb.getResource("name").getString();
            System.out.println("Nombre: " + name);
      }

} finally{
      qexec.close() ;
}
```

Figura 23 : Ejemplo de ejecución de consulta SPARQL utilizando Jena

Finalmente, si simuláramos la ejecución del ejemplo en el dominio representado por la Figura 16, la ejecución daría como resultado “`Nombre: Cliente`”.

3.3.4 Chequeos Implementados

Como fue mencionado anteriormente, aunque en [15] se implementaron 5 chequeos de consistencia, en realidad al iniciar este trabajo se contaba con sólo 2 que consistían en “Objeto Abstracto” y “Comportamiento no compatible”. Sin embargo la utilidad de contar con dichos chequeos en la nueva versión de MCC era nula, dado que se utilizó un nuevo motor de inferencia con nuevas tecnologías incompatibles con nRQL que era el lenguaje utilizado anteriormente en los chequeos. En resumen, posterior a la inclusión de Jena como motor de inferencia no se contaban con chequeos implementados.

Como una forma de mostrar la correctitud de las traducciones del modelo UML al metamodelo en OWL-DL, se decidió implementar primero 3 chequeos básicos que consultaran

distintos aspectos de este. Finalmente se decidió agregar una cuarta detección de inconsistencias, de mayor complejidad en consultas, para mostrar que es posible implementar los chequeos utilizando SPARQL en vez de nRQL. La lista final de chequeos implementados es la siguiente:

- Objeto abstracto: Se genera cuando en un diagrama de secuencias se instancia una clase abstracta que no tiene hijos. Si la clase abstracta tiene subclases concretas, el comportamiento especificado en el diagrama de secuencia sería aplicable a una instancia de dichas subclases.

  Este chequeo utiliza instancias del metamodelo que corresponden a un diagrama de clases, por lo tanto permite verificar la correctitud de la traducción.

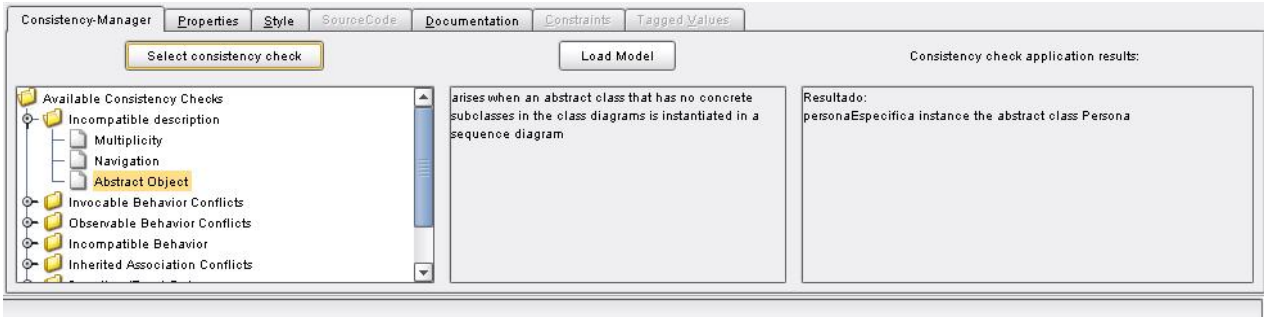

Figura 24 : Ejemplo de detección de inconsistencia “Objeto Abstracto”

- Diagrama de secuencia desconectado: Esta inconsistencia ocurre cuando un diagrama tiene uno o varios objetos del diagrama que no están conectados al diagrama de secuencia principal. Usualmente ocurre cuando un objeto (lifeline) del diagrama de secuencias es borrado de manera accidental.

  Utiliza gran parte de las instancias del metamodelo que es posible utilizar en un diagrama de secuencia, lo que asegura que se dispone de una traducción correcta de estos diagramas.

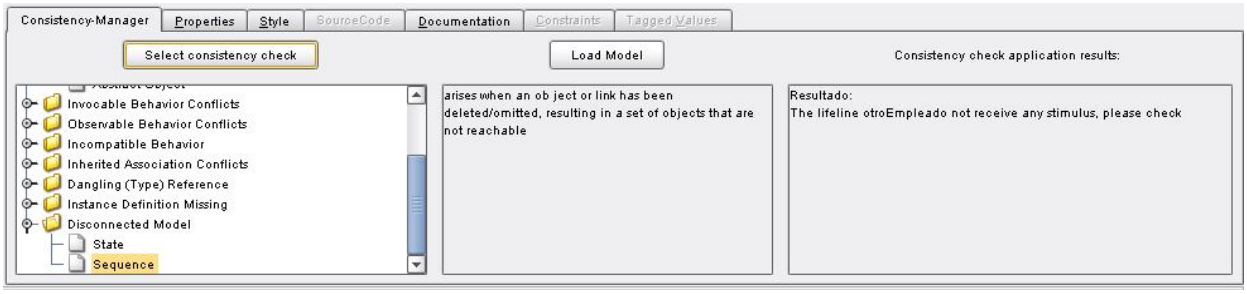

Figura 25 : Ejemplo de detección de inconsistencia “Diagrama de secuencia desconectado”

- Diagrama de estados desconectado: Esta inconsistencia ocurre cuando un diagrama tiene uno o varios estados o transiciones que no están conectadas con el diagrama principal. Usualmente ocurre cuando un estado o transición del diagrama es borrado accidentalmente.

  Para la detección de esta inconsistencia, se utiliza gran parte de las instancias del metamodelo que es posible utilizar en un diagrama de estados, lo que asegura que se dispone de una traducción correcta de estos diagramas.

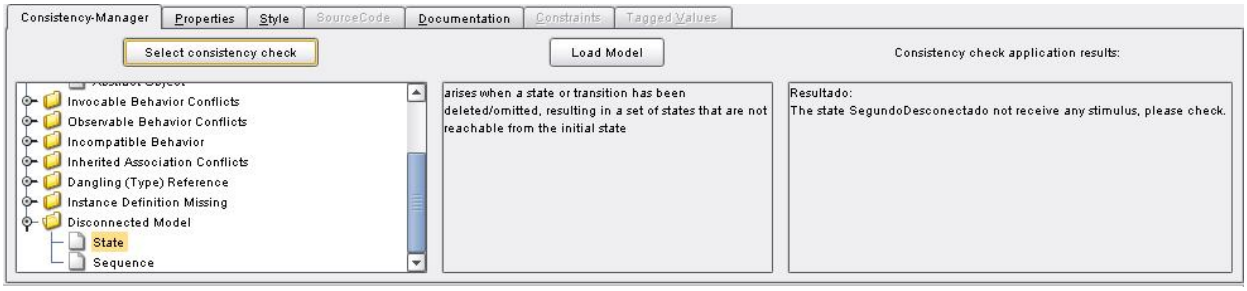

Figura 26 : Ejemplo de detección de inconsistencia “Diagrama de estados desconectado”

- Comportamiento no compatible: Esta inconsistencia ocurre cuando la secuencia de mensajes recibidos por un objeto en un diagrama de secuencia es inconsistente con el comportamiento establecido en la diagrama de estados de la clase del objeto.

  Para este chequeo, es necesario contar con las traducciones correctas tanto de los diagramas de secuencia como los de estado. Además la complejidad de las consultas SPARQL es mucho mayor, lo que nos permite por una parte verificar que es posible realizar consultas tan potentes como en nRQL y por otro probar el rendimiento del motor frente a estas consultas.

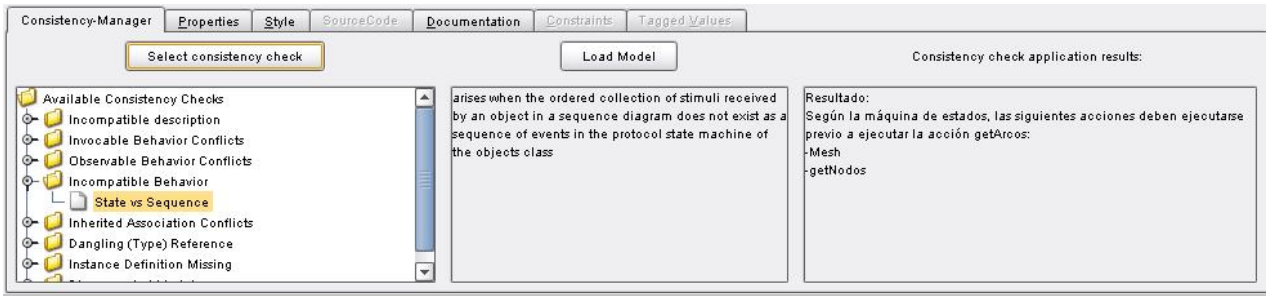

Figura 27 : Ejemplo de detección de inconsistencia “Comportamiento no compatible”

3.3.5 Uso de MCC

Uno de los principales aspectos considerados a la hora de tomar de decisiones durante el trabajo, fue priorizar la simplicidad de uso de MCC. Esto debido a que si bien la versión de Jocelyn Simmonds pretendía ocultar el uso de un motor de inferencia lógica para chequear consistencia, esto no se lograba de manera absoluta, principalmente dada la necesidad de instalar previamente el motor de inferencia, ejecutarlo cada vez que se quisiesen chequear inconsistencias, y configurar MCC dándole una ruta absoluta a un archivo utilizado de extensión “.racer”, además de un puerto y número IP. Estos puntos fueron mejorados drásticamente en esta nueva versión, consiguiéndose que para utilizar MCC deban seguirse tan sólo 3 pasos:

1- Instalación de MCC como Plug-In de Poseidon (solo la primera vez)

Esta instalación se realiza utilizando el llamado “Panel de Plug-Ins” de Poseidon accesible desde el menú superior denominado “Plug-Ins”. En dicho panel, se debe presionar el botón “Agregar…”, posteriormente localizar el archivo de la aplicación (mcc.jar), y finalmente presionar “Instalar” (ver Figura 28). Además, para el uso de MCC es necesario agregar un número válido de licencia (incluido al momento de descargar la aplicación) en el panel de “Administración de Licencias” en el menú de ayuda (ver Figura 29).

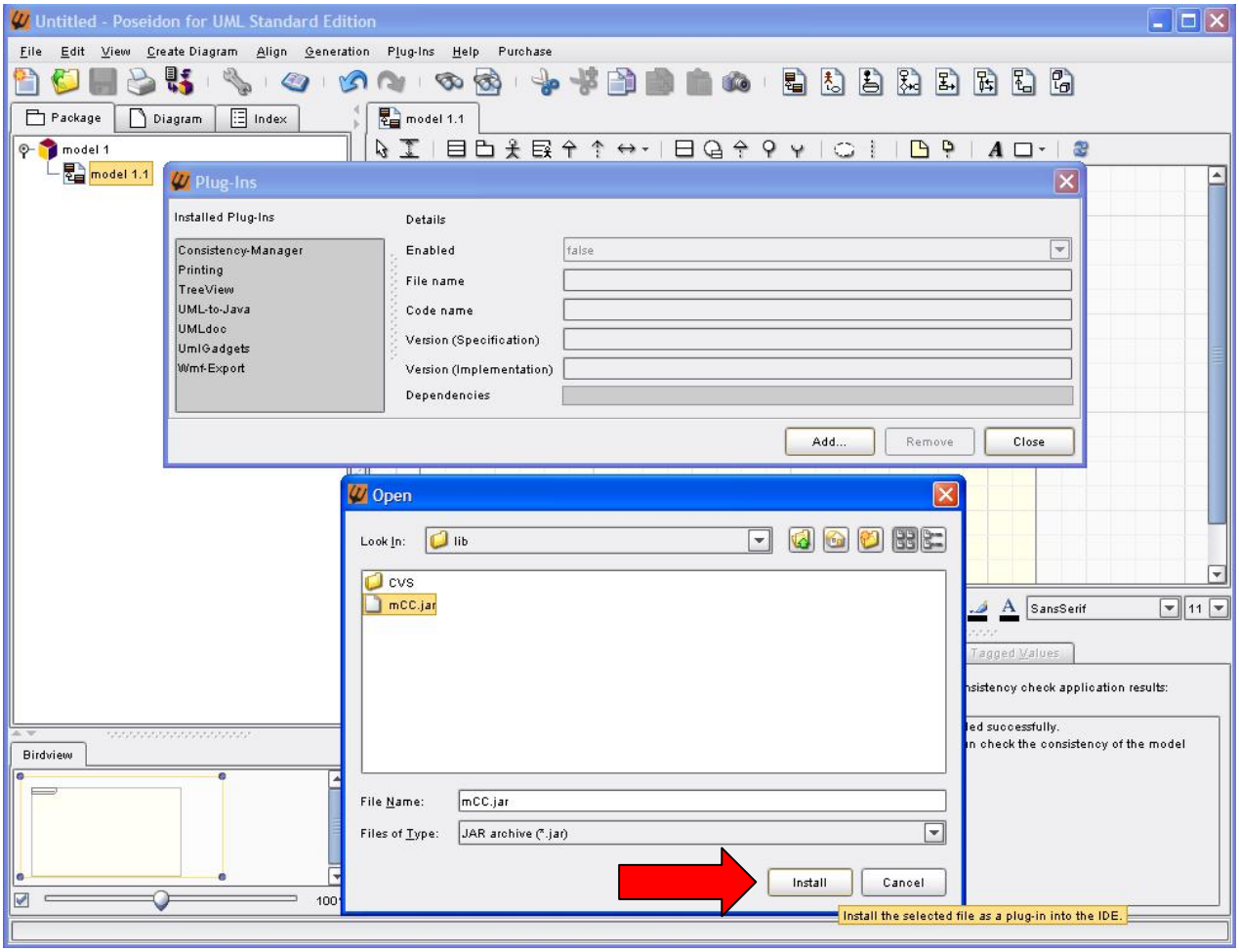

Figura 28 : Instalación de MCC como Plug-In de Poseidon

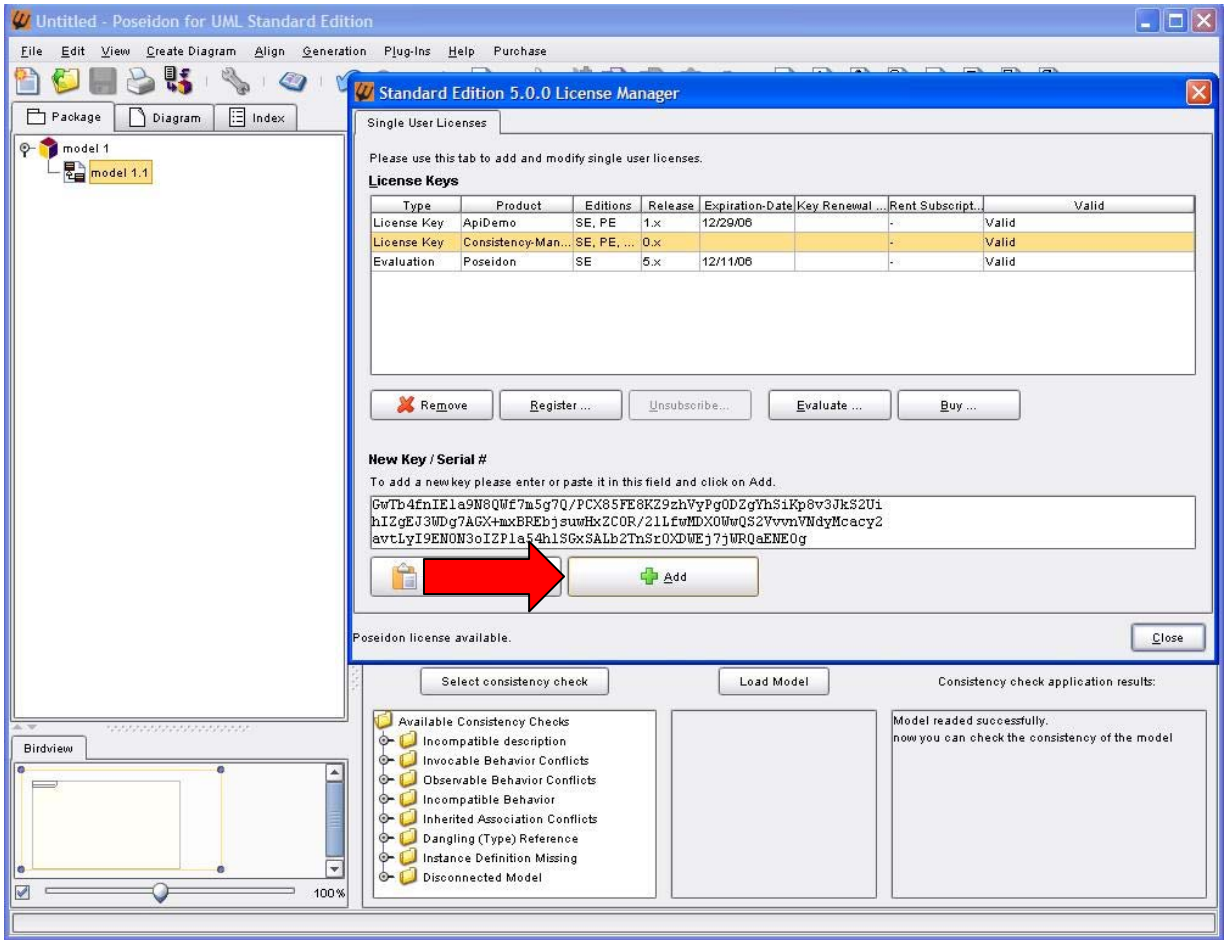

Figura 29 : Instalación de la licencia de MCC

2- Cargar el modelo UML en trabajo

Previo a iniciar la detección de posibles inconsistencias, es necesario cargar el modelo UML que debe ser chequeado. Para esto basta con abrir en Poseidon el modelo a trabajar y posteriormente en el panel de MCC presionar el botón “Load Model” (ver Figura 30). La carga puede tomar algunos segundos tras los cuales una frase indicando que se puede iniciar la detección de inconsistencias aparecerá en el panel a la derecha de la aplicación.

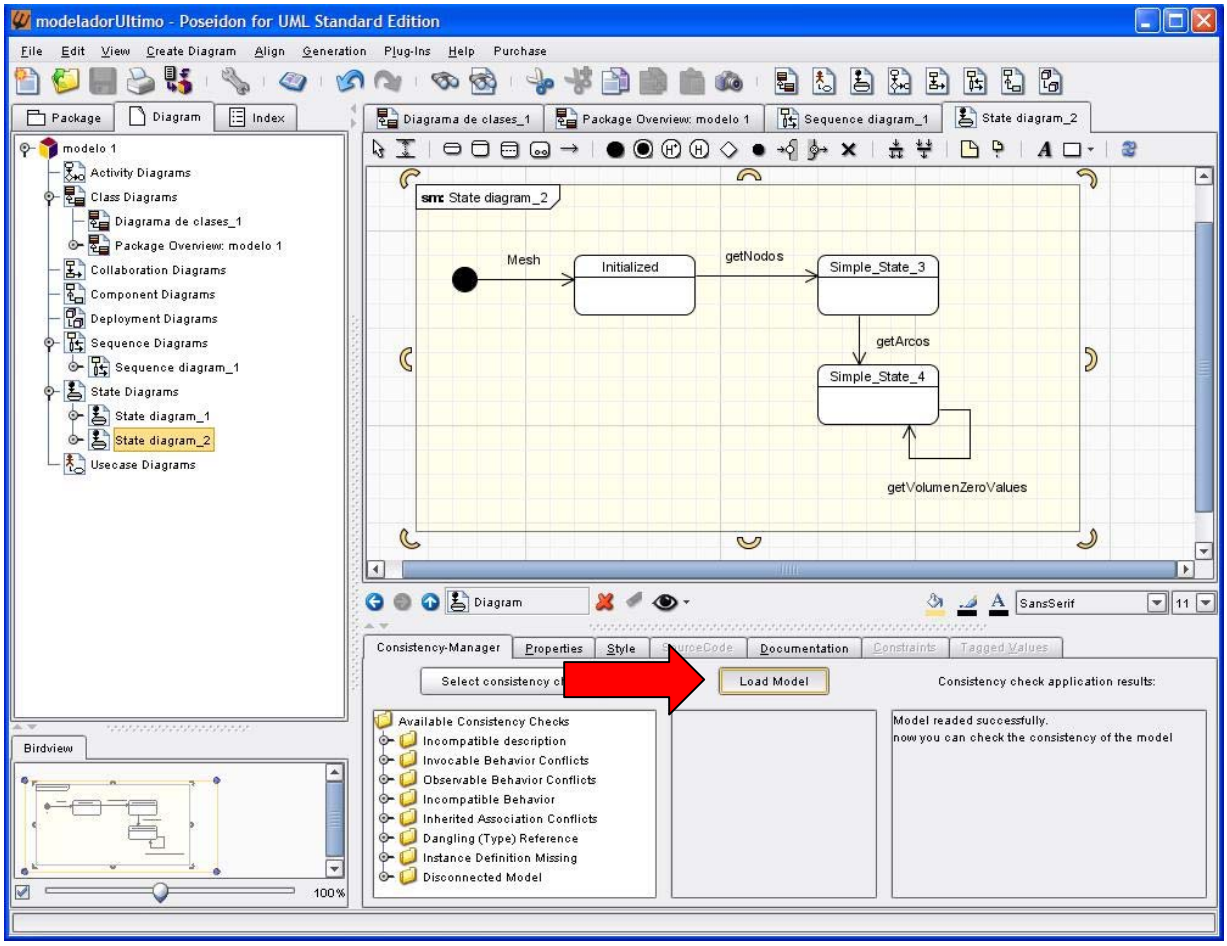

Figura 30 : Carga de Modelo UML a MCC

3- Seleccionar el chequeo a consultar

Finalmente basta con seleccionar un chequeo en la lista (categorizada según [14]) a la izquierda de la aplicación y presionar el botón “Select consistency check” con lo cual se consulta el modelo cargado previamente y los resultados se muestran en el panel derecho.

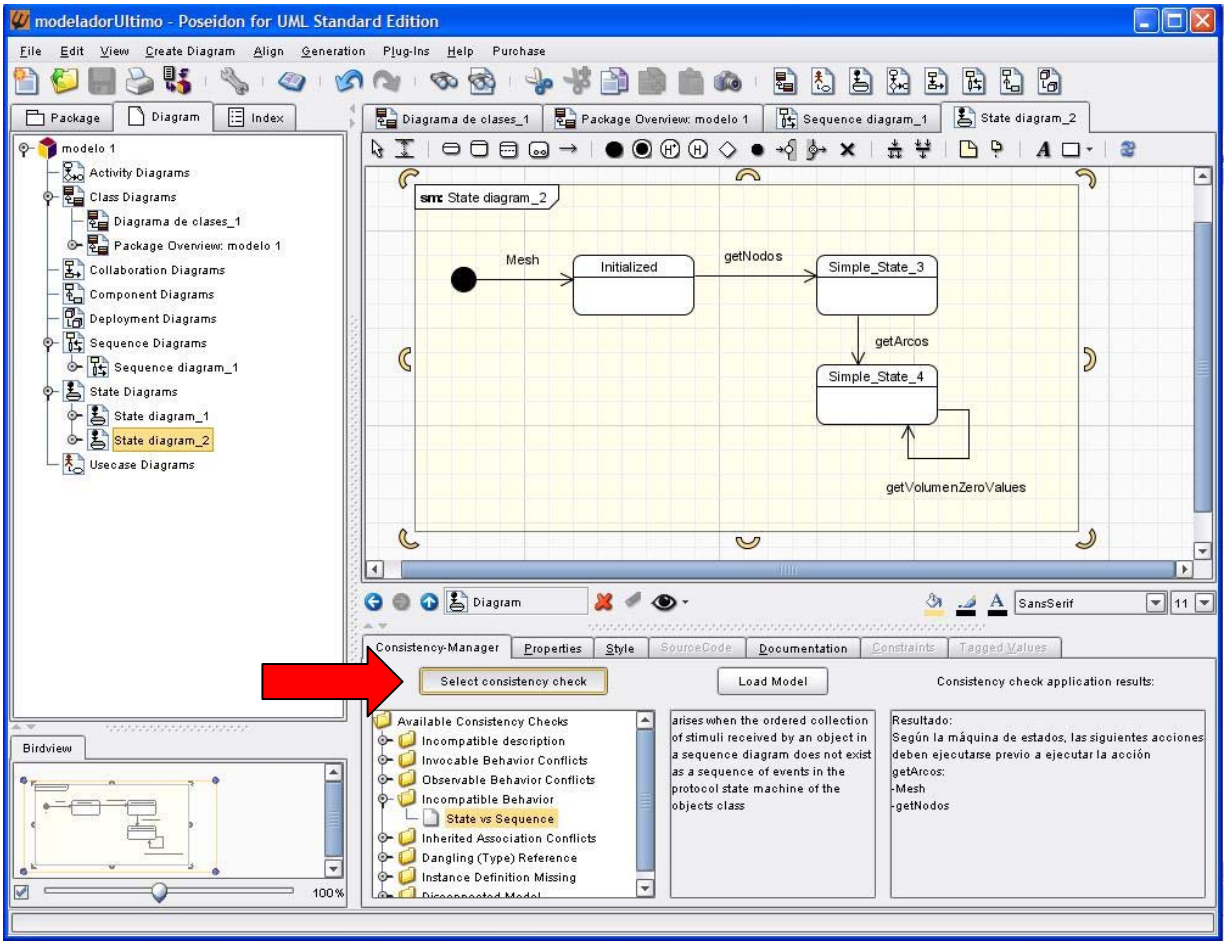

Figura 31 : Selección de un chequeo de consistencia en MCC

Importante:

Una observación importante del uso de MCC es que debido a problemas de incompatibilidad de Poseidon con algunas librerías y clases externas, para el correcto funcionamiento de MCC, es necesario que la librería externa “xercesLib.jar” se encuentre en el CLASSPATH Java de la aplicación. Es posible realizar dicha inclusión en el archivo de partida de Poseidon.

4. Ejemplo de validación

De manera de validar la herramienta desarrollada, es necesario incluir un ejemplo de uso real, en que se demuestre la dificultad de detectar inconsistencias de manera manual, y por lo tanto la utilidad de contar con una herramienta que lo haga en forma automatizada.

4.1 Presentación del ejemplo

La herramienta hallada es un generador de mallas de superficie basado en triangulaciones apropiadas para la simulación del crecimiento de árboles. A grandes rasgos, la herramienta permite:

(a) Leer una malla inicial almacenada en archivos, en diferentes formatos.

(b) Simular el crecimiento representado por el proceso de "move", el cual desplaza cada uno de los vértices de la malla una cierta cantidad en la dirección especificada en el archivo de entrada.

(c) Detectar posibles colisiones producidas por el desplazamiento de puntos vecinos y corregirlas. Para esto se permiten reparaciones de la malla que incluyen corrección de dirección de desplazamiento y eliminación de triángulos.

(d) Mejoramiento de la calidad de los triángulos.

Esta herramienta cuenta con una serie de clases que permiten implementar la funcionalidad descrita, y que se muestran en la Figura 32.

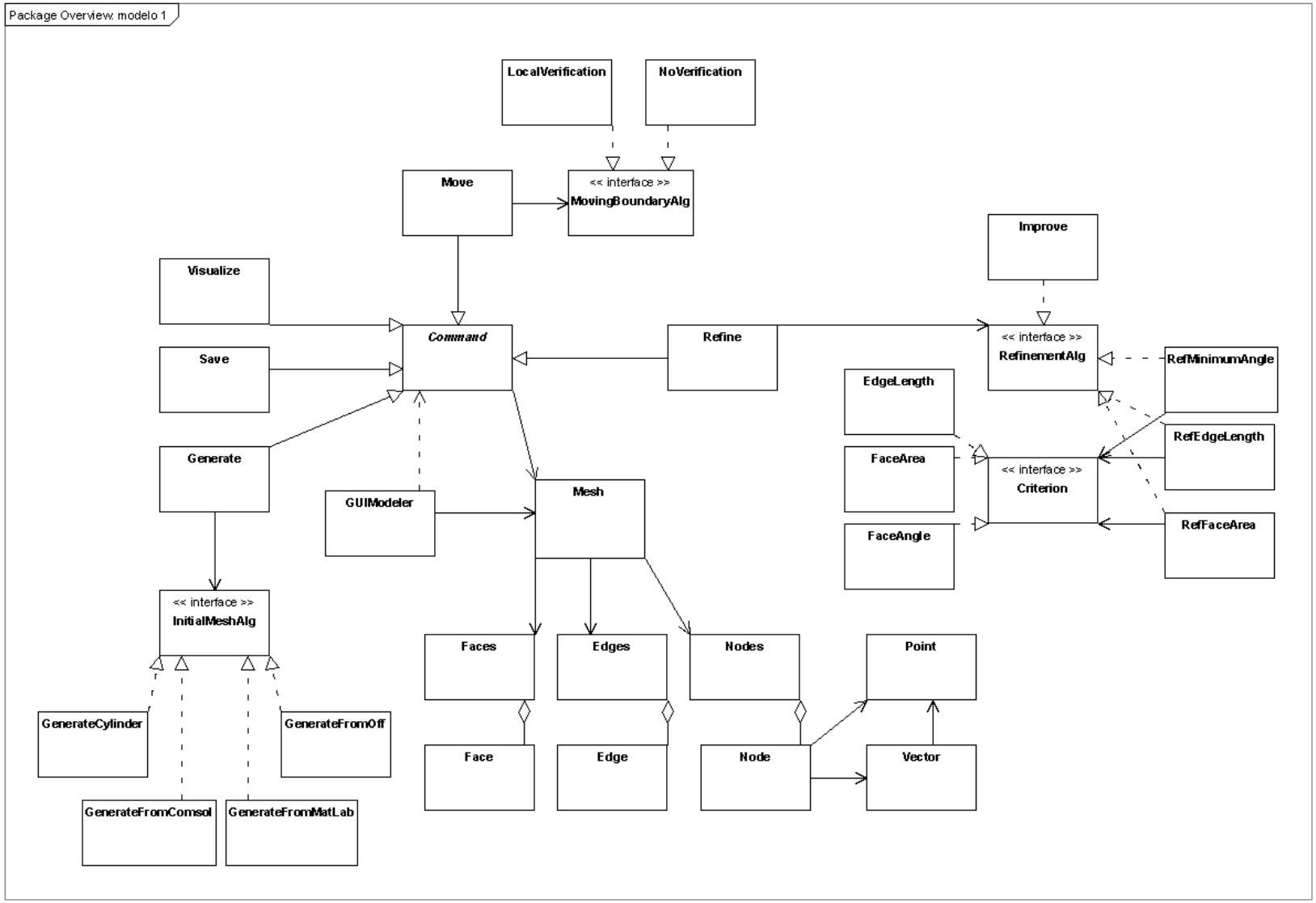

Figura 32 : Diagrama de clases de ejemplo de validación.

Además se cuenta con un diagrama de estados de la clase Mesh (Malla) mostrado en la Figura 33 y que describe que el correcto uso de una malla es inicializarla, pedir sus nodos, luego sus arcos y sus posibles valores de volumen cero.

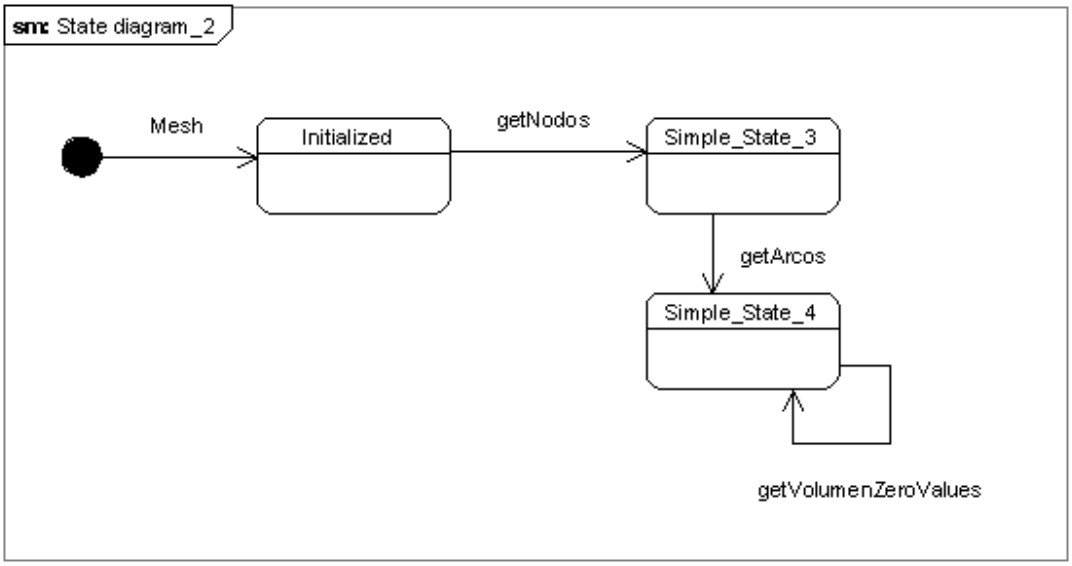

Figura 33 : Diagrama de estados de la clase “Mesh” en ejemplo de validación.

Finalmente en la Figura 34 se cuenta con un diagrama de secuencia que representa la acción de mover (move) usando un algoritmo que chequea la consistencia local de la malla, es decir, que triángulos vecinos no choquen ante el desplazamiento de cada uno de sus puntos. Move consiste en recorrer todos los arcos de la malla para verificar si las caras que lo comparten se superponen. Si hay colisión, se detecta la que ocurre más cerca de la superficie actual. Todos los puntos son movidos proporcionalmente un valor determinado por la distancia a la que se produce la primera inconsistencia.

El diagrama de secuencia de la Figura 34 realiza entonces primero las inicializaciones correspondientes y luego recorre todos los arcos para detectar la inconsistencia más cerca, si es que existe. Una vez determinado este valor, se recorren los puntos para moverlos proporcionalmente a esta distancia. En caso de no haber colisión, los puntos se mueven todo el desplazamiento especificado en el archivo de entrada.

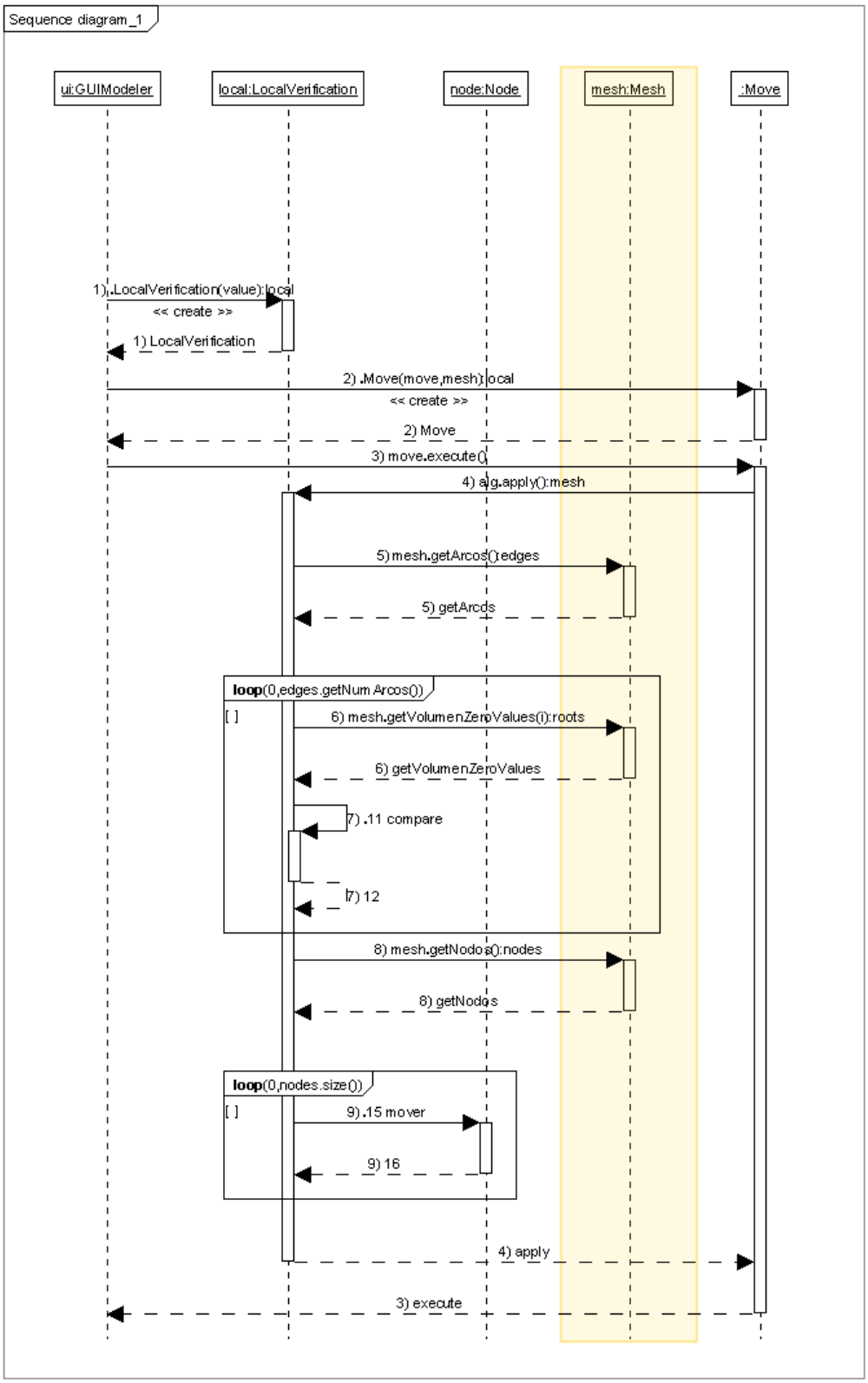

Figura 34 : Diagrama de secuencia de la acción mover (move) en ejemplo de validación.

4.2 Detección de inconsistencias

El ejemplo representado por las figuras: Figura 32, Figura 33 y Figura 34 representa tan solo una porción del modelo total que se puede presentar para la aplicación. La presencia de inconsistencias en estos diagramas claramente puede resultar en diversas confusiones posteriores durante el desarrollo de la aplicación o incluso errores futuros.

La carga de este modelo en MCC toma alrededor de 10 segundos, tras los chequeos se realizan en no más de 3 segundos. Claramente estos resultados son dependientes del desempeño del equipo en donde se encuentra instalada la aplicación, pero dan luces de tiempos razonables de espera, sobretodo pensando en que estos tiempos pueden ahorrar largas horas de desarrollo perdidas por la existencia de inconsistencias en los modelos.

La aplicación fue sometida a los chequeos de consistencia implementados. Los resultados muestran que los tres chequeos más básicos resultaron en inexistencia de inconsistencias. Sin embargo la aplicación del chequeo “Comportamiento incompatible” arrojó el resultado observable en la Figura 35.

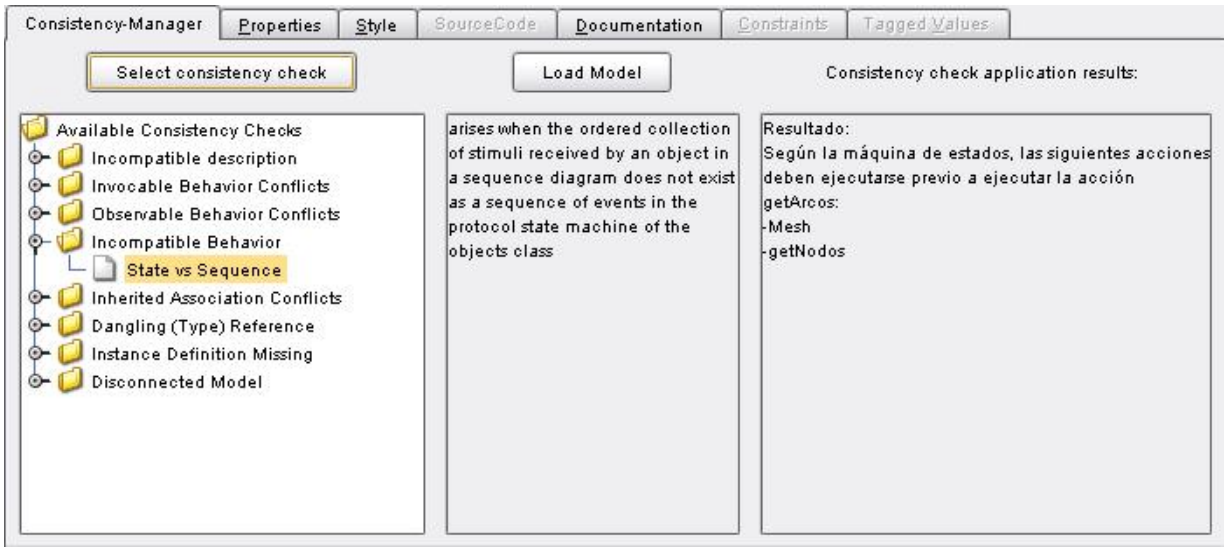

Figura 35 : Aplicación del chequeo “Comportamiento incompatible” en el modelo representado por las figuras: Figura 32, Figura 33 y Figura 34.

Los resultados observados indican la necesidad de primero inicializar la instancia “mesh”, esto dado que se indica que debe primero ejecutarse la acción “Mesh”, o sea el constructor de la clase del mismo nombre.

La inicialización previa de la instancia de nombre “mesh” resulta en el diagrama de secuencia representado por la Figura 36.

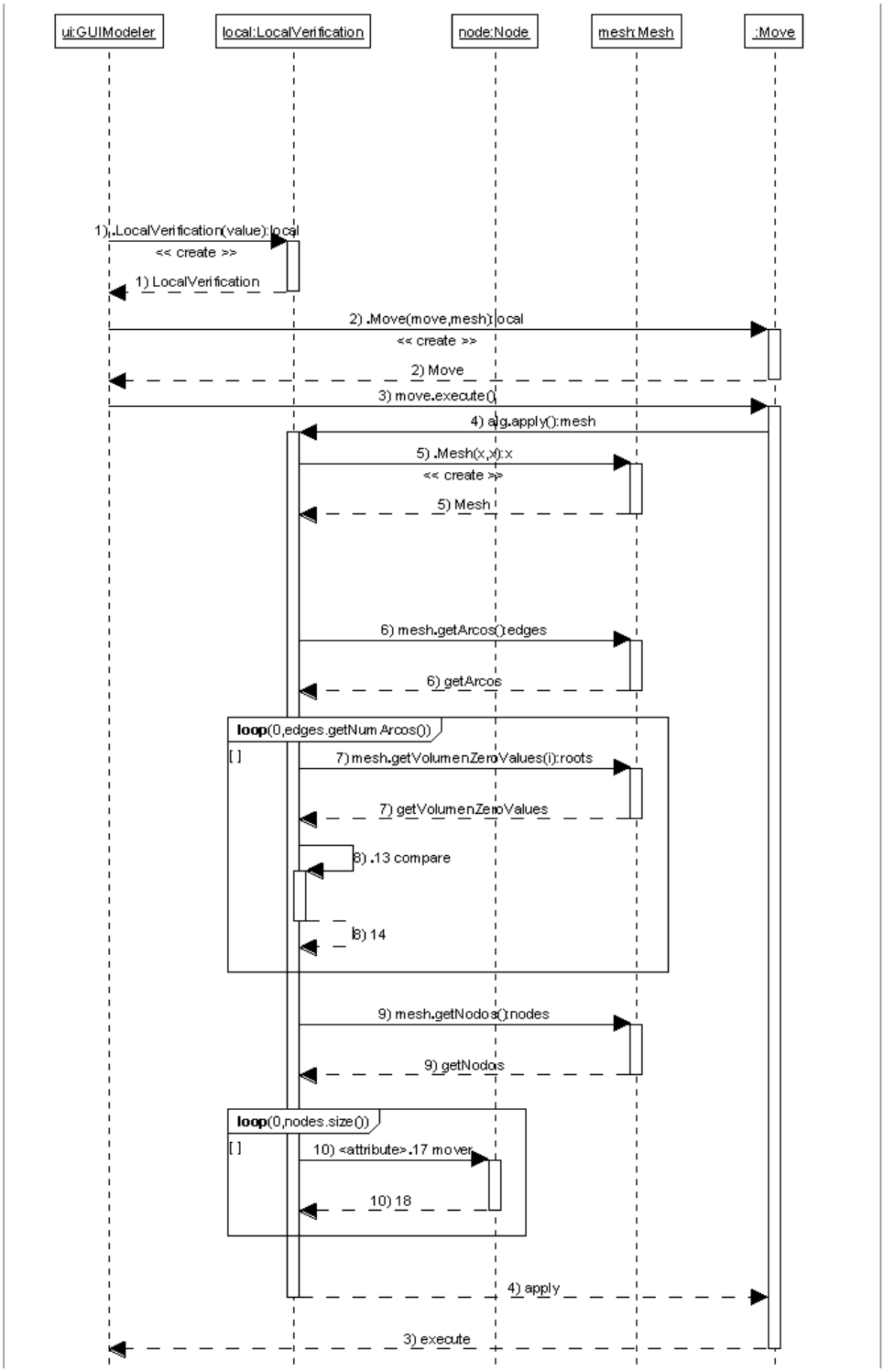

Figura 36 : Diagrama de secuencia corregido de la acción mover (move).

Nuevamente se realiza un chequeo en búsqueda de posibles nuevas inconsistencia. Los resultados son presentados en la Figura 37

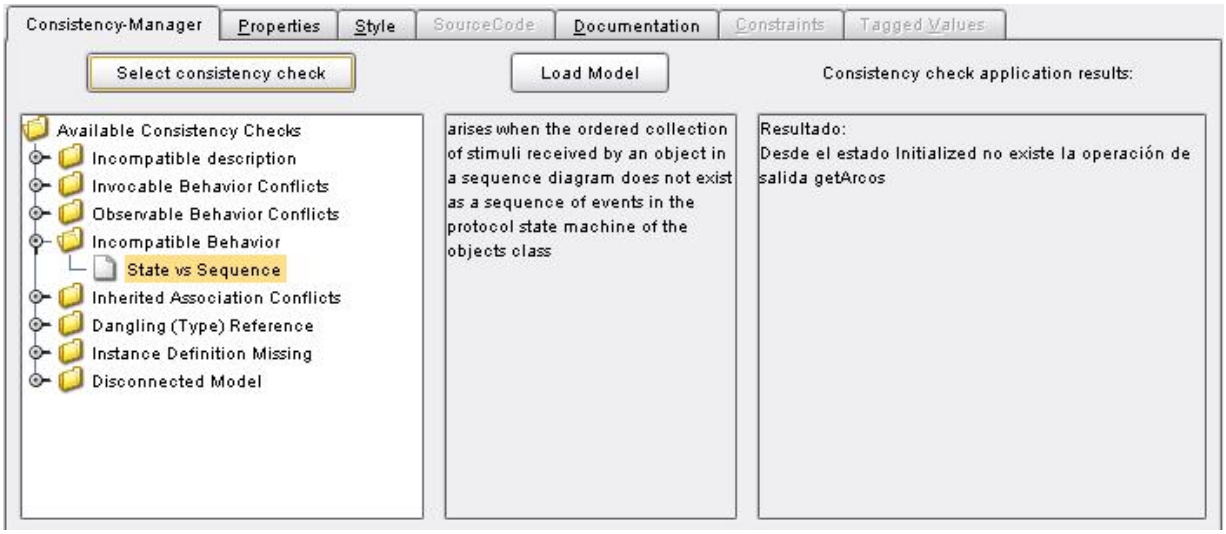

Figura 37 : Aplicación del chequeo “Comportamiento incompatible” en el modelo representado por las figuras: Figura 32, Figura 33 y Figura 36.

La nueva aplicación del chequeo nos revela que aún existen inconsistencias, esta vez debido a que según el diagrama de secuencia, tras instanciar la clase “Mesh”, se pasa al estado “Initialized” que no tiene como mensaje de salida la acción “getArcos”. Finalmente es posible deducir tras examinar ambos diagramas que el problema se debe a que previo a llamar al método “getArcos” debe llamarse a “getNodos”.

La corrección permite obtener el diagrama de secuencia representado por la Figura 38 que nuevamente es sometido al mismo chequeo de consistencia. El resultado mostrado en la finalmente indica que no se encuentran este tipo de inconsistencias.

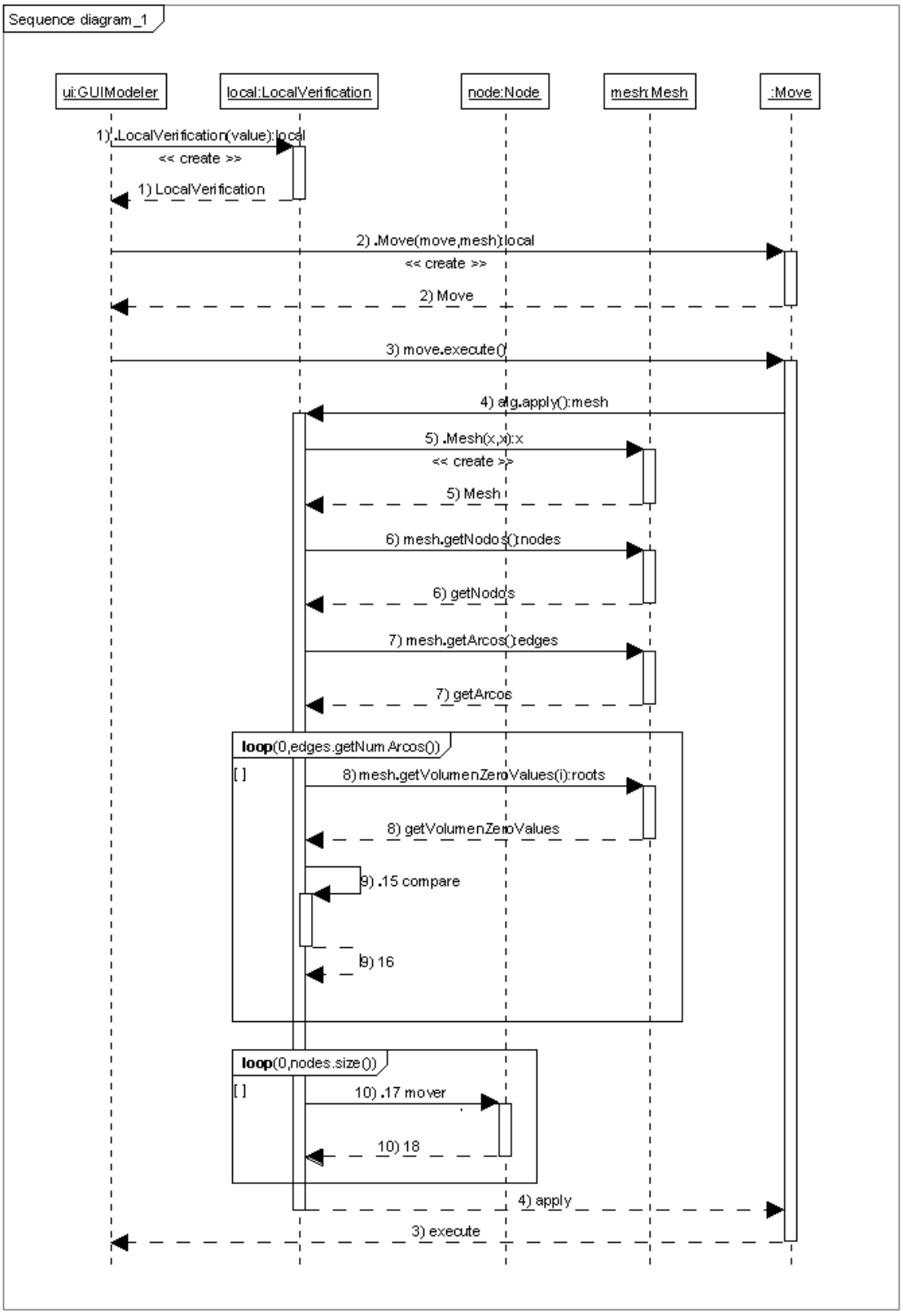

Figura 38 : Diagrama de secuencia final de la acción mover (move).

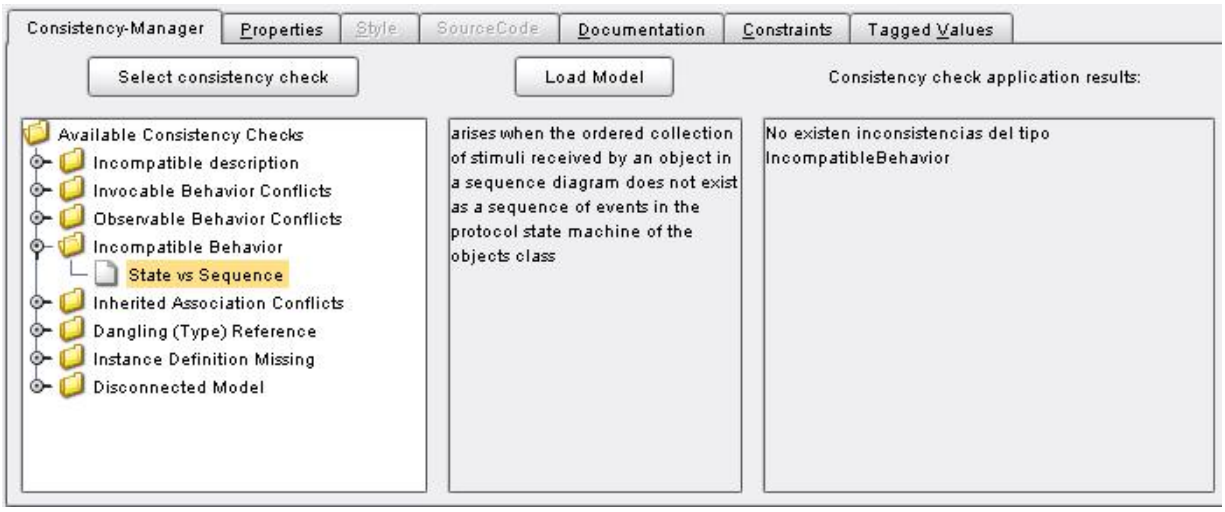

Figura 39 : Aplicación del chequeo “Comportamiento incompatible” en el modelo representado por las figuras: Figura 32, Figura 33 y Figura 38.

5. Conclusión y Discusiones

UML 2.0 [11] es actualmente el estándar de facto en el modelado de sistemas de software. La especificación desarrollada por la OMG indica qué se puede modelar y como debe hacerse, sin embargo no obliga a que los diagramas del modelo sean consistentes. Si bien esto puede resultar positivo al momento del modelamiento, la permanencia de dichas inconsistencias en las etapas posteriores puede provocar un impacto negativo en los tiempos de desarrollo, o incluso en la estabilidad o calidad del sistema final. Lo anterior hace patente la necesidad de chequear los modelos y de contar para esto, con una herramienta que permita corregir a tiempo los posibles errores.

MCC [16] se creó inicialmente con el objetivo de detectar inconsistencias, sin embargo la falta de actualizaciones lo ha convertido en una herramienta obsoleta. Por otra parte, su arquitectura contempla la existencia de dependencias externas, lo hace una aplicación difícil de utilizar y que no cumple debidamente con ocultar al usuario final la existencia de un motor de inferencia lógica. Aún peor es que dichas dependencias externas se han vuelto aplicaciones comerciales, lo que desfavorece aún más el uso de MCC.

La existencia de nuevos motores de inferencia lógica desarrollados en Java, tales como Jena [9], permiten desarrollar una nueva versión de MCC más autocontenido, ya que elimina la necesidad de conexiones externas con un motor de Description Logic [3]. Además, la existencia de nuevos estándares o recomendaciones W3C en la industria como OWL-DL [23] y SPARQL [27], que fueron incluidos en la nueva aplicación, permiten asegurar una mejor mantenibilidad a futuro.

La nueva herramienta desarrollada cuenta con mayor facilidad de uso, dado que se eliminó por completo la necesidad de configuraciones complejas. Además la instalación se simplificó a solo agregar MCC como Plug-In de Poseidon, e incluir una biblioteca externa en las variables de entorno de la aplicación, eliminando la necesidad de instalar aplicaciones adicionales. Finalmente, si bien no se realizaron pruebas exhaustivas, el rendimiento de la aplicación resulta razonable y los tiempos de espera no debieran superar los 15 a 20 segundos en la carga de modelos de tamaño similar al ejemplo.

La utilidad de la herramienta fue probada mediante un ejemplo de uso en el que fue posible apreciar errores no evidentes a simple vista y que más tarde fueron corregidos de manera

de lograr diagramas consistentes. A través del ejemplo es posible ver la potencialidad de la herramienta, dado que la permanencia de las inconsistencias detectadas pudieron haber repercutido en diferentes problemas tales como mal entendimiento de las clases por parte de los desarrolladores, con las consecuentes perdidas de tiempo en el desarrollo, o el posible comportamiento inadecuado de la aplicación dado un orden incorrecto en la ejecución de los métodos.

Si bien el foco principal de trabajo se concentró en mejorar los problemas existentes en la integración con el motor de inferencia lógica Racer [7], la dependencia con Poseidon [6] es actualmente también fuente de problemas, sobretodo debido a que se ha convertido en una herramienta comercial y no ofrece ayudas en cuanto a documentación para el desarrollo. Sin embargo es una buena herramienta para el modelamiento UML, por lo que deben analizarse los beneficios de un posible cambio, así como también los costos que esto implicaría. En este sentido deben tomarse en cuenta herramientas como argoUML [21] predecesora de Poseidon, de código libre, pero lamentablemente con soporte solo para UML 1.4. Otra alternativa para tomar en cuenta es Visual Paradigm [22] que si bien es comercial, cuenta con una versión comunitaria gratuita, también permite la creación de Plug-Ins y cuenta con una mejor documentación que Poseidon.

Otro punto importante y que queda pendiente es que las inconsistencias implementadas por MCC fueron determinadas por un trabajo de investigación desarrollado el año 2003 [14]. Si bien el trabajo fue concluido con la determinación de 18 chequeos de inconsistencias, que no han sido implementados de forma completa en MCC, es posible investigar la existencia de nuevos tipos de inconsistencia utilizando mayor cantidad de información como la proveniente de diagramas de componentes omitidos hasta ahora. Por último, resulta necesario analizar más en detalle el rendimiento de la aplicación, sobretodo pensando en la posibilidad de incluir más información o de considerar modelos mayores en tamaño y en número de diagramas.

6. Bibliografía

[1] BAADER, Franz, SATTLER, Ulrike. An Overview of Tableau Algorithms for Description Logics [en línea]. http://citeseer.ist.psu.edu/baader00overview.html. [Consulta: noviembre de 2006]

[2] Clark & Parsia, LLC. Pellet: An OWL DL Reasoner [en línea]. http://pellet.owldl.com/. [Consulta: noviembre de 2006]

[3] Description Logic (DL) [en línea]. http://dl.kr.org/. [Consulta: noviembre de 2006]

[4] DL Implementation Group (DIG). DIG Interface [en línea]. http://dig.sourceforge.net/. [Consulta: noviembre de 2006]

[5] Frame Logic (F-Logic) [en línea]. http://flora.sourceforge.net/aboutFlogic.php. [Consulta: noviembre de 2006]

[6] Gentleware. Poseidon for UML [en línea]. Consultada en abril de 2006 http://gentleware.com/

[7] HAARSLEV, Volker, MÖLLER, Ralf. RACER [en línea]. http://www.racer-systems.com/. [Consulta: noviembre de 2006]

[8] HAARSLEV, Volker. The New Racer Query Language – nRQL [en línea]. http://www.cs.concordia.ca/~haarslev/racer/racer-queries.pdf. [Consulta: noviembre de 2006]

[9] HP Labs Semantic Web Programme. Jena – A Semantic Web Framework for Java [en línea]. http://jena.sourceforge.net/. [Consulta: noviembre de 2006]

[10] Information Process Engineering (IPE). KAON2 [en línea]. http://kaon2.semanticweb.org/. [Consulta: noviembre de 2006]

[11] Object Management Group. Unified Modeling Language [en línea]. http://www.uml.org. [Consulta: noviembre de 2006]

[12] Racer-Systems. RacerPorter, the native, graphical user interface for RacerPro [en línea]. http://www.racer-systems.com/products/porter.phtml. [Consulta: noviembre de 2006]

[13] Racer-Systems. Release notes for RacerPro Version 1.9 [en línea]. http://www.racer-systems.com/products/racerpro/release.phtml. [Consulta: noviembre de 2006]

[14] SIMMONDS WAGEMANN, Jocelyn. Consistency Maintenance of UML Models with Description Logics. Belgium: Vrije Universiteit Brussel. France: Ecole des Mines de Nantes Department of Computer Science, 2003. Master's thesis.

[15] SIMMONDS WAGEMANN, Jocelyn. Un Framework para el manejo de consistencia en diseños UML 2.0. Santiago: Universidad de Chile, Departamento de Ciencias de la Computación, 2005. Memoria para optar al título de Ingeniero Civil en Computación.

[16] SIMMONDS WAGEMANN, Jocelyn. MCC [en línea]. http://www.dcc.uchile.cl/~jsimmond/mcc/mcc.html. [Consulta: noviembre de 2006]

[17] Stanford Medical Informatics. Protégé [en línea]. http://protege.stanford.edu/. [Consulta: noviembre de 2006]

[18] Sun Microsystems. Java 2 EE 1.4 SDK [en línea]. http://java.sun.com/javaee/downloads/previous/. [Consulta: noviembre de 2006]

[19] Sun Microsystems. Java EE 5 SDK [en línea]. http://java.sun.com/javaee/downloads/index.jsp. [Consulta: noviembre de 2006]

[20] Sun Microsystems. Java Platform, Standard Edition 6 release [en línea]. http://java.sun.com/javase/6/. [Consulta: noviembre de 2006]

[21] Tigris.org. ArgoUML [en línea]. http://argouml.tigris.org/. [Consulta: noviembre de 2006]

[22] Visual Paradigm. Visual Paradigm for UML [en línea]. http://www.visual-paradigm.com/. [Consulta: noviembre de 2006]

[23] World Wide Web Consortium (W3C). OWL Web Ontology Language Guide [en línea]. http://www.w3.org/TR/owl-guide/. [Consulta: noviembre de 2006]

[24] World Wide Web Consortium (W3C). Resource Description Framework, RDF [en línea]. http://www.w3.org/RDF/. [Consulta: noviembre de 2006]

[25] World Wide Web Consortium (W3C). RDF Data Access Working Group [en línea]. http://www.w3.org/2001/sw/DataAccess/. [Consulta: noviembre de 2006]

[26] World Wide Web Consortium (W3C). Semantic Web [en línea]. http://www.w3.org/2001/sw/. [Consulta: noviembre de 2006]

[27] World Wide Web Consortium (W3C). SPARQL Query Language for RDF [en línea]. http://www.w3.org/TR/rdf-sparql-query/. [Consulta: noviembre de 2006]

[28] World Wide Web Consortium (W3C). SWRL: A Semantic Web Rule Language Combining OWL and RuleML [en línea]. http://www.w3.org/Submission/SWRL/. [Consulta: diciembre de 2006]