Enfoque para la validación sintáctica de modelos organizacionales de Sistemas Multiagentes

Pedro Bernabé Araujo[1] y Sebastián Alberto Rodríguez[1]

Resumen

Durante las últimas décadas la tecnología orientada a agentes se ha convertido en una de las herramientas más importantes para el modelado de sistemas complejos, distribuidos y abiertos. Entre los sistemas multiagentes (SMA) el enfoque organizacional promueve una nueva forma de abordar problemas complejos inspirados de la metáfora social.

Las herramientas CASE son uno de los componentes fundamentales para la correcta adopción de una nueva tecnología en la industria; asistiendo al diseñador en el proceso de desarrollo.

En este articulo, presentamos un enfoque para la validación sintáctica automatizada de modelos para SMA basados en el metamodelo CRIO

Palabras clave: validación sintáctica, metamodelo, CRIO, sistemas multiagentes, reglas.

1 Universidad Tecnologica Nacional, Facultad Regional Tucumán, Argentina.

Abstract

Over the last decades, agent-oriented technology has become one of the main approaches to model complex, open and distributed systems.

Among the main approaches for Multiagent Systems, Organizational modeling promotes a new way of tackling complex problems based on the social metaphor.

In this context, CASE tools are a key component for a proper adaption of new technologies by the industry. These tools assist the designer throughout the design process.

In this article we present an automatic syntax validation approach for MAS models based on the CRIO Metamodel.

Key words: syntax validation, metamodel, CRIO, multiagent systems, rules.

I. Introducción

Durante las últimas décadas la tecnología orientada a agentes se ha convertido en una de las herramientas más importante para el modelado de sistemas complejos, distribuidos y abiertos. En este sentido los sistemas basados en agentes han demostrado exitosamente su potencial abordando una amplia variedad de problemas que van desde la robótica, resolución de problemas distribuidos, modelado y simulación, solo por mencionar algunas áreas (Wooldridge & Jennings, 1995; Wooldridge, 2009). El creciente interés se debe, entre otros, al concepto de agente como entidad autónoma. Un agente es una entidad física o virtual con un alto grado de autonomía, independiente, capaz de cooperar, competir, comunicarse, actuar flexiblemente y ejercer control sobre su comportamiento dentro del marco de sus objetivos. Un sistema multiagentes está compuesto por múltiples agentes inteligentes que interactúan entre sí para alcanzar objetivos que pueden ser compartidos o no entre los agentes (Weiss, 2013).

Entre los sistemas multiagentes (SMA de ahora en más) el enfoque organizacional promueve una nueva forma de abordar problemas complejos. Este enfoque consiste en descomponer los problemas en partes más pequeñas además de proveer el contexto de interacción entre los agentes de cada una de estas unidades. (J. Ferber & Gutknecht, 1998) define que dos niveles son posibles, nivel "organizacional" y nivel "agente". El nivel organizacional o social (también llamado "qué") es donde se puede observar los aspectos dinámicos y estructurales de una organización SMA. Este nivel describe la relación esperada y los patrones de actividad que deberían ocurrir en el nivel de agentes. También, es común encontrar conceptos tales como role, grupos, comunidades, tareas e interacción. En el nivel de agentes (denominado "como") describe el comportamiento del agente. En otras palabras, detalla la arquitectura interna del agente, sus estados mentales, creencias, deseos, intenciones y metas, y si es reactivo o intencional.

Además, adoptar el enfoque organizacional permite al diseñador tratar a los problemas a través de dos estrategias posibles: la descomposición vertical y la horizontal. La descomposición vertical permite que el comportamiento que representa a la organización sea fragmentado en un conjunto de sub-organizaciones de menor abstracción. En cambio, la descomposición horizontal, modela las interacciones existentes entre las entidades presentes en el mismo nivel de abstracción el cual es necesario para alcanzar los objetivos requeridos (Cossentino, 2010).

En la Ingeniería de Software Orientada a Agentes (ISOA o AOSE por sus siglas en inglés), requiere para el análisis, diseño e implementación de cuatro elementos fundamentales: el metamodelo y los lenguajes que se utilizarán para describir los modelos; la metodología que define la secuencia de pasos a seguir y los actores involucrados para la obtención de un diseño del producto; la plataforma

de implementación sobre la cual se ejecutarán estos modelos; y por último la herramienta CASE (Computer Aided Software Engineering) utilizada para asistir al diseñador en el proceso de desarrollo.

En este artículo presentamos los avances realizados en la herramienta CASE denominada Janeiro Studio (Araujo, 2013) para el metamodelo CRIO (Rodriguez, 2007) y la validación sintáctica de modelos.

El trabajo está estructurado de la siguiente manera: la sección 2 se realiza un breve resumen de los trabajos que existen en la literatura sobre herramientas que permiten el modelado de sistemas multiagentes. Sección 3 se presenta la adaptación del metamodelo CRIO y los diferentes diagramas asociados al mismo. Sección 4, introduce a la validación sintáctica de modelos. Sección 5, se presenta un conjunto de reglas de validación sintácticas que actualmente se están utilizando en la herramienta Janeiro Studio. Sección 6 es presentado un caso de estudio. Y por último, en la Sección 7, son realizadas las conclusiones.

II. Trabajos relacionados

Una CASE es un conjunto de herramientas informáticas que asisten al diseñador en algunas de las actividades relacionadas con el desarrollo de un sistema (requerimientos, análisis, diseño, codificación y pruebas). Estas herramientas permiten transmitir lo más rápido posible las intenciones de los desarrolladores mediante una combinación de notaciones gráficas y textuales que representan algún lenguaje específico compartido por el equipo de desarrollo. Además, incrementan la productividad de los equipos de desarrollo reduciendo tiempo y costos a su vez que mejora la calidad del software entregado (Sommerville, 2010). Existen diferentes herramientas presentes en la literatura para el modelado de sistemas multiagentes, muchas de estas cuentan con módulos de validación que permiten asegurar que los modelos sean conceptualmente válidos. En este trabajo nos enfocamos en aquellas que se encuentran centradas en los conceptos organizacionales (Organization-Centred MultiAgent System). Muchas de estas herramientas son para metamodelos específicos y brindan soporte visual para algunos o la mayoría de sus diagramas más relevantes. La tabla 1 ofrece un resumen de las principales herramientas CASE para el modelado de SMA.

Tabla 1 – Herramientas y sus características

	Metodología	Diagramas Cubiertos	Validación de modelos	Validación entre modelos	Generación de código	Activo
AgentTool III (Garcia-Ojeda, 2009)	OMASE	Todos	Si	Si	Si	Si
GAIA4E (Cernuzzi, 2009)	GAIA	Todos	N/E	N/E	No	N/E
IDK (Gomez-Sanz, 2008)	Ingenias	Principales	No	No	Si	Si
PDT (Padgham, 2005)	Prometheus	N/E	Si	Si	Si	N/E
Metameth/PTK (Cossentino, 2005)	PASSI	Todos	Si	Si	Si	No/Si
OpenTool (Picard, 2004)	Adelfe	Principales	Si	No	N/E	No
Rebel (Al-Hashel, 2007)	ROADMAP	Principales	No	No	No	No
T-Tool (Susi, 2005)	Tropos	Principales	Si	No	No	No
N/E - No especifica.						

III. Visión General del Enfoque

Durante los últimos 30 años la tecnología agentes representa una alternativa para abordar problemas complejos. Dentro de los SMA, el enfoque organizacional ha ganado especial importancia. Este está inspirado en la metáfora social y es usado tanto en metodologías (GAIA (Wooldridge, 2000), MESSAGE (Caire, 2002), ASPECS (Cossentino, 2010)) como en metamodelos (AGR (Ferber, 2004), MOCA (M. Amiguet, 2003), CRIO (Rodriguez, 2007)). Conceptos como "Organización", "Grupo", "Comunidad", "Roles", "Protocolos" son términos comunes en este tipo de enfoque. En el presente artículo nos centraremos en el metamodelo CRIO el cual extiende de RIO (Hilaire, 2000).

CRIO es un metamodelo basado en los conceptos organizacionales. Su nombre resulta del acrónimo formado por sus cuatros conceptos principales:

Una *Capacidad* es la descripción de un know-how/servicio. En otras palabras es la especificación de una transformación de una parte de un sistema o de su ambiente. Es una abstracción de alto nivel que promueve la reusabilidad y modularidad y en este sentido puede ser considerado como un componente básico de diseño. Además, el concepto de capacidad permite la definición de un rol sin hacer ninguna suposición de la arquitectura interna de éste.

Un *Role* es definido como un comportamiento esperado (un conjunto de tareas del rol ordenados por un plan) y un conjunto de derechos y obligaciones dentro del contexto de la organización. El objetivo de cada rol es contribuir en alcanzar los requerimientos de la organización dentro del cual está definido.

Una *Interacción* es una secuencia de eventos dinámica intercambiada entre roles (una especificación de alguna ocurrencia que puede potencialmente disparar efectos en el sistema) o entre roles y entidades fuera del sistema.

Por último, una *Organización* se define como una colección de roles y sus interacciones dentro de un determinado contexto. La figura 1 muestra el diagrama UML del metamodelo.

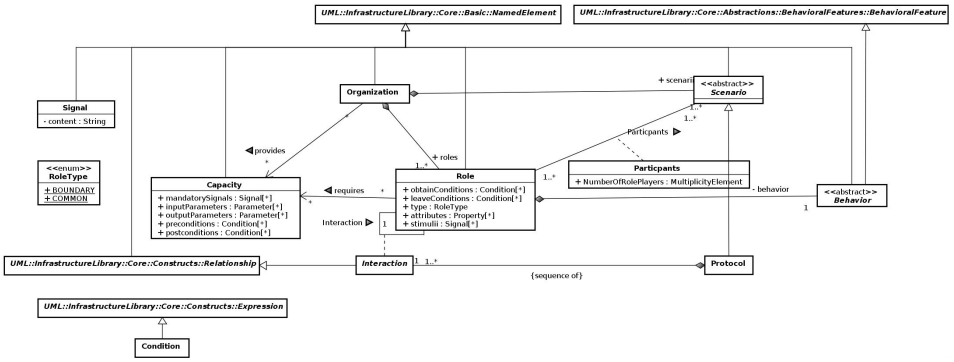

Figura 1 – Metamodelo CRIO (Cossentino, 2010)

a. Representación informática de CRIO

El metamodelo descripto en la sección anterior no es fácilmente manipulable por herramientas informáticas. Por este motivo es necesario adaptarlo para su manipulación y validación. En este contexto se ha seleccionado el Eclipse Modeling Framework (EMF)[2] para esta representación.

EMF es parte importante de una gran familia de proyectos llamado Eclipse Modeling Project (EMP)[3]. Es una librería que permite a los expertos construir sus propias aplicaciones basadas en un metamodelo de datos estructurados denominado Ecore. Un Ecore simplifica la definición de un metamodelo facilitando la programación requerida para la implementación de un modelo específico del dominio. Por otra parte, el uso de EMF permite el modelado de jerarquías donde un modelo es el metamodelo de otro.

Los conceptos utilizados en este framework son más simples que aquellos definidos por la OMG[4]. Esto se debe a que la intención es obtener rápidamente un

[2] Eclipse Modeling Framework, www.eclipse.org/emf/
[3] Eclipse Modeling Project, www.eclipse.org/modeling/
[4] Object Management Group, www.omg.org

código ejecutable. Los conceptos principales son EPackage (paquete), EClass (clase), EDataType (tipo de datos), EEnum (enumeración) y EObject (objeto) además de la posibilidad de representar a través del concepto de EReference las relaciones de asociación y agregación. El metamodelo CRIO fue desarrollado completamente usando este framework y además es el núcleo principal de nuestro enfoque.

Para el soporte de la modelización basado en CRIO y ASPECS, se ha desarrollado la CASE Janeiro Studio. Para Janeiro Studio se utilizó Eclipse Rich Client Platform (RCP) (McAffer, 2010) como base dada las ventajas que ofrece dicha plataforma para el desarrollo de herramientas integradas. Provee un entorno rico en opciones y debido a su popularidad posee una comunidad importante que le da soporte. También destacamos su desarrollo basado en el concepto de plugins que permite una alta modularización del sistema. Un plugin es definido como una unidad de modularidad en Eclipse, de hecho todo en Eclipse es desarrollado bajo este concepto. Básicamente, un plugin es autodescriptivo y explicita una lista de los otros plugins a los cuales depende para un funcionamiento adecuado. RCP nos permite una manera clara y sencilla de gestionar los plugins para construir aplicaciones complejas y funcionales. Otro de los framework involucrados es Graphical Modeling Framework (GMF) útil para establecer como los conceptos serán representados gráficamente en el entorno de desarrollo propuesto.

Actualmente, y para cubrir las etapas tempranas de requerimientos, Janeiro cuenta con tres Ecore. (i) DRDMetamodel, (ii) PODMetamodel, (iii) ProblemMetamodel. Los dos primeros metamodelos son las bases de los Diagramas de Requerimientos del Dominio y de Ontología. El tercer metamodelo, ProblemMetamodel, surgen los Diagramas de Organización, Interacción y Comportamiento. Estos diagramas pueden considerarse como vistas del metamodelo y es la forma en que será manipulado el metamodelo. En la tabla 2, presentada más adelante, se puede ver en detalle los Ecore y sus diagramas asociados. En este artículo, y por cuestiones de espacio, nos centraremos en el Ecore principal de la herramienta, ProblemMetamodel.

Tabla 2 – Ecore y sus diferentes diagramas

Ecore	Diagramas	Descripción
DRDMetamodel	Diagrama de Requerimientos	Con diagramas similares a los usados en los CASOS de Usos de UML, es posible documentar las expectativas y necesidades de los interesados. Estos diagramas permiten capturar los requerimientos funcionales y no funcionales utilizando un lenguaje específico del dominio provisto por los usuarios.

PODMetamodel	Diagrama de Ontologías	Permite identificar, modelado como un diagrama de clases, los conceptos, predicados, y acciones relevantes del dominio. La idea principal es conceptualizar los requerimientos descriptos en el DRD o en cualquier documento que describa el sistema a desarrollarse.
ProblemMetamodel	Diagrama Organizacional	Permite representar gráficamente la estructura organizacional. La idea detrás de este diagrama es modelar la organización que represente el comportamiento global. Esta representación es realizada a través de un conjunto de roles presentes en la organización y de la interacción entre ellos. La capacidad define el comportamiento que es requerido por el agente para que pueda tomar el rol. Los conceptos usados en este diagrama son : organización, rol, capacidad, protocolo entre otros.
	Diagrama de Interacción	Describe la secuencia de intercambio de información entre los roles que tienen lugar en un protocolo. Los conceptos más relevantes en este diagrama son: protocolo, rol, interacción, callCapacity, señales, atributos, etc.
	Diagrama de Comportamiento	Describe el comportamiento de un rol (la dinámica de la instancia). En otras palabras, permite representa los estados posibles y las transiciones de un rol. Este diagrama es un perfil del diagrama de estados de UML.

En la figura 2 se muestra la adaptación de CRIO en EMF. En el mismo se distingue claramente los conceptos centrales que conforman dicho metamodelo tales como capacidad, organización, rol, protocolo, entre otros. Para que esta representación sea posible fue necesaria la realización de algunas adaptaciones específicas que sirven para cumplir con las especificaciones inherentes del metamodelo CRIO. El Diagrama Organizacional tiene como elementos principales a los conceptos de organización, compuesta por rol, protocolo y capacidad. Del concepto de rol se desprende *behaviour* que es el elemento raíz para el Diagrama de Comportamiento y en donde se describe los estados y las transiciones posibles que pueden darse en un rol. El concepto de *Participant* nos permite enlazar el *rol* con *protocolo*. Por último, *protocol* es el elemento principal del Diagrama de Interacción, el mismo está compuesto de *interacciones*, *RoleSet*, *callCapacity* y *onSignal.* El primero, interacción, es el intercambio de información entre roles. El concepto de *RoleSet* es el que nos permite representar al rol en el Diagrama de Interacción dado que el framework utilizado no es posible la utilización de un mismo concepto en dos diagramas diferentes. Y por último, tanto *callCapacity* como *onSignal* son dos tipos

de operaciones especiales que puede realizar un rol y que indica el llamado a una capacidad o el evento disparado por un agente indicando la finalización de alguna actividad, respectivamente.

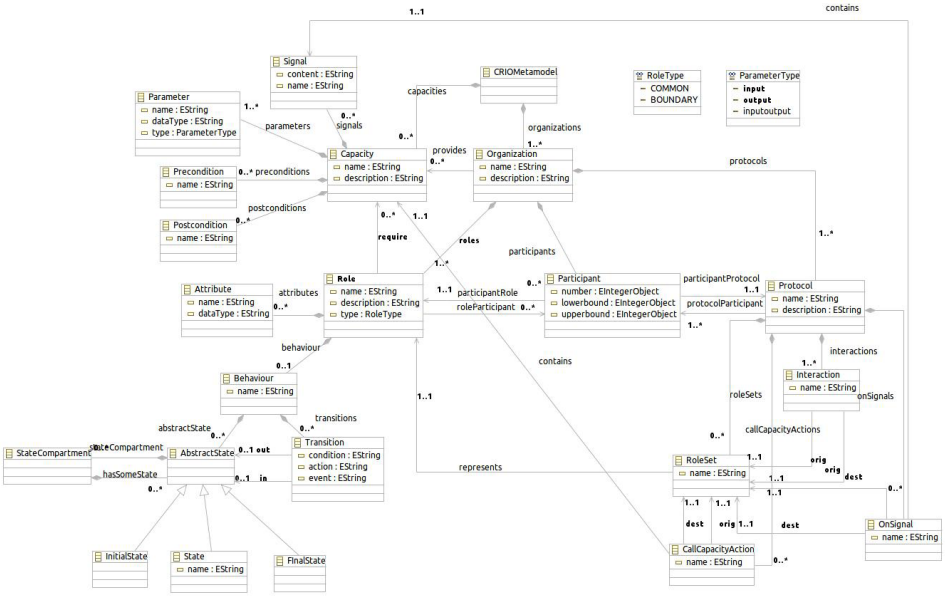

Figura 2 – Adaptación del metamodelo CRIO a EMF

b. Manipulación del Modelo

A medida que un proyecto itera a través de los ciclos de desarrollo y pasado un determinado tiempo, los diseñadores pueden comenzar a validar y verificar que los modelos respeten determinadas reglas. El paradigma orientado a objetos cuenta con un gran número de herramientas de validación y verificación, en cambio en la tecnología agente no podemos decir lo mismo; dado que en la actualidad existen muy pocas herramientas que permitan la validación de modelos. Es más, ese trabajo en muchos casos es realizado por personas y de forma manual con las desventajas que trae aparejado (Fagan, 1976).

En la figura 3 se muestra una vista general de los conceptos utilizados para la validación sintáctica. Utilizando la definición del metamodelo en EMF podemos construir editores que brindarán a los usuarios de una metáfora gráfica de los conceptos además de hacer respetar ciertas restricciones que están presentes en el metamodelo (Manipulación del modelo). Basándonos en estos diagramas podemos generar un modelo del sistema real. Las vistas principales del metamodelo están detalladas en la Tabla 2. Si bien determinados elementos pueden estar validados a través del metamodelo de forma automática por las herramientas, existen un

conjunto de características sintácticas que no pueden ser validadas o la necesidad de validarlas hacen impráctica la utilización de la herramienta. Es por ello que sería interesante contar con una herramienta que nos permita definir nuestras propias reglas de validación además de hacerlo de forma automatizada.

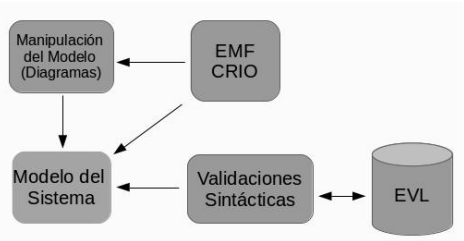

Figura 3 – Vista general

IV. Validación Sintáctica

Es posible realizar validaciones de diferentes maneras y abordando diferentes aspectos de un modelo. El objetivo de la definición de reglas de validación es la de detectar, de manera temprana, los defectos del modelos para que puedan ser tratadas de manera inmediata.

Detectar un error en etapas tempranas y de forma automática trae aparejado un importante número de ventajas, entre ellas permitirá reducir drásticamente los costos de desarrollo del sistema al mismo tiempo que el producto se acerca a los estándares de calidad deseada por la organización. Es por ello que contar con un mecanismo de validación automatizada nos permitirá responder rápidamente a los errores detectados (Unhelkar, 2005).

El lenguaje seleccionado para la validación de los modelos es Epsilon Validation Language (EVL)[5]. Utilizamos EVL en vez de Object Constraint Language (OCL) [6]que es el lenguaje más utilizado), dado que posee múltiples ventajas que permiten mayor especificidad de las propiedades de los modelos. La combinación de los modelos y EVL provee una alta expresividad que habilita a los desarrolladores a formular propiedades adicionales que no pueden ser expresados en la notación gráfica. Algunas de las ventajas que ofrece EVL son: (i) la posibilidad de definir restricciones que son independientes de otras, (ii) modularizar o descomponer las consultas complejas en otras más sencillas, (iii) permite especificar mensajes personalizados cuando no se satisface un invariante, (iv) permite solucionar dinámicamente las inconsistencias encontradas, (v) soporta operaciones de entrada y salida del usuario, entre otras características.

[5] Epsilon Validation Language, www.eclipse.org/epsilon/doc/evl
[6] Object Constraint Language, www.omg.org/spec/OCL/

V. Reglas de Validación Sintáctica de modelos

En esta sección se detallan algunas de las reglas de validación para modelos basados en CRIO. Las reglas presentadas en este artículo representan una parte del conjunto de restricciones obligatorias que son definidas por el metamodelo CRIO. Por limitaciones de espacio, solo se presentarán las algunas reglas por cada concepto a forma de ilustración del enfoque.

Reglas asociadas a Rol

(1) Es obligatorio que los roles posean un nombre.

```
 Role {
    constraint HasName {
        check : self.name.isDefined()
        message : 'Role must have a name'
    }
}
```

Como hemos mencionado anteriormente el rol representa una parte del comportamiento de la organización. La restricción HasName definida en el contexto del rol permite validar a través de su función isDefined() que todos los roles definidos en la organización posean un nombre.

(2) El nombre de un rol debe ser único dentro de la organización.

```
context Role {
    constraint DuplicatedName {
        guard : self.satisfies("HasName")
        check {
            var roleList : Set( Role );
            roleList.addAll( self.eContainer().roles );
            roleList.remove( self );
            return not roleList->exists( c | c.name = self.name );
        }
        message : 'Duplicated role name: ' + self.name
    }
}
```

Además de la validación presentada en el punto anterior, también resulta importante que en un modelo no existan duplicaciones de nombres dado que puede crear confusión para aquellos que utilizan el modelo. Esta regla posee una restricción DuplicatedName que permite detectar dos nombre iguales de roles en la organización. Antes de ejecutar check, la restricción verifica primero que a los roles se les haya asignado un nombre. En el cuerpo de la función se declara una lista donde se recuperan todos los roles presentes en la organización (menos el rol sobre el cual está situado la regla) y se verifica, con la función exists(), si hay coincidencia de nombre. Si existe una coincidencia la función devolverá verdadero por lo que se antepone una negación para que la condición retorne falso y sea violada la restricción.

Reglas asociadas a Capacidad

(3) Una capacidad debe estar referenciada al menos una vez.

```
context Capacity {
    constraint InsolatedCapacity {
        check {
            var capacityList : Set( Capacity );
            for ( r in Role.allInstances ) {
                capacityList.addAll( r.require );
            }
            return  capacityList.includes( self );
        }
        message : 'All capacities must be referenced'
    }
}
```

Una capacidad es definir el comportamiento genérico del rol identificado cuales son las competencias necesarias que debe tener el agente para poder tomar el rol. Un enlace entre el rol y la capacidad indica que la capacidad es requerida por el rol. La regla de validación permite detectar aquellas capacidades que no son enlazadas con ningún roles. Usando como contexto la capacidad, primero se define un conjunto capacityList del tipo Capacity. A su vez, el lazo for permite recorrer todas las instancias que existen de Rol en el modelo y agregarla a las lista recién definida. Una vez fuera del lazo, el algoritmo verifica si la capacidad sobre la que estoy situado está presente en la lista capacityList. En caso de que la función includes() retorne false, se violaría la restricción informando al usuario que la capacidad no es referenciada o requerida por ningún rol.

(4) Las capacidades deben contener al menos una señal.

```
context Capacity {
    constraint HasALeastOneSignal {
        check : self.signals.size() > 0
        message : 'The Capacity ' + self.name + ' must have at least one signal'
    }
}
```

En el contexto agente, los estímulos de un rol están definidos como un conjunto de señales a los cuales el rol reacciona. Los agentes pueden disparar señales que son eventos definidos en la capacidad y que permiten comunicar resultados o el estado de la ejecución de las tareas realizadas por la capacidad. Usando Capacidad como contexto, la restricción HasALeastOneSignal recupera a través de la función size() el número de señales asociados a la capacidad. Si el número es igual que cero la restricción es falsa por lo que el usuario es notificado del error.

Reglas asociadas a Organización

(5) Las organizaciones deben estar compuestas por al menos un rol.

```
context Organization {
    constraint HasALeastOneRole {
        check : self.roles.size() > 0
        message : 'The Organization ' + self.eClass().name + ' must have at least
one role'
    }
}
```

Una vez definido el comportamiento global y representado a través de una organización, el siguiente paso es descomponer ese comportamiento es unidades más pequeñas. Cado uno de estos fragmentos identificados serán los comportamientos exhibidos por el rol. Esta regla valida que en la organización existe al menos un rol. Utilizando Organización como contexto de la regla, la restricción HasALeastOneRole recupera, a través de la función size() el número de roles existentes en la organización. Si la función es mayor que cero la organización cumple con la restricción, ahora, si el número es igual a cero la restricciones es violada y por tanto se notifica al usuario.

(6) Las organizaciones deben contener al menos un protocolo.

```
context Organization {
   constraint HasALeastOneProtocol {
     check : self.protocols.size() > 0
        message : 'The Organization ' + self.eClass().name + ' must have at least
one protocol'
   }
}
```

El concepto del protocolo permite al diseñador definir como es la interacción entre los agentes. En otras palabras permitirá detallar la secuencia de paso de información entre los roles participantes. La regla definida permite validar que en la organización exista al menos un protocolo. Usando la organización como contexto la regla navega a través del atributo *protocols* para recuperar el número de instancias presente en dicha organización. Si el número devuelto por la función *size*() es mayor que cero la restricción evalúa a verdadero, en cambio si es igual a cero la restricción es falso por lo tanto notifica al usuario sobre la necesidad de corregir este error.

Reglas asociadas a Interacción

(7) Debe existir al menos una interacción en los protocolos.

```
context Protocol {
    constraint HasALeastOneInteraction {
        check : self.interactions.size() > 0
        message : 'This protocol must have a least one interaction'
    }
}
```

Una interacción en un Diagrama de Interacción representa un intercambio de información entre dos roles. Esta información puede ser o un mensaje o un acto de lenguaje, entre otras posibilidades. La regla presentada en este inciso verifica que en el protocolo exista al menos una interacción. La restricción HasALeastOneInteraction recupera a través de la función size() el número de instancias de interacciones. Si el número de instancias es igual a cero el usuario es notificado de que en el protocolo no existen interacciones.

(8) En una señal el origen y el destino deben ser el mismo rol.

```
context OnSignal {
   constraint IsTheSameRole {
     check {
     var band : Boolean = false;
     if (self.orig = self.dest) {
       band = true;
     }
     return band;
      }
      message: 'Origin a destination role must be the same'
    }
}
```

Tanto las señales como los llamados a capacidad son tipos especiales de operaciones que puede realizar un rol. Estos son representados como auto-mensajes que pueden o no llevar parámetros de entrada o salida sea o un llamado a capacidad o una señal, respectivamente. En la condición if la restricción IsTheSameRole se compara que el origen como el destino sea la misma instancia del rol.

VI. Aplicación práctica

El ejemplo utilizado para mostrar la efectividad de las reglas de validación semánticas fue presentado en (Basso, 2011). A continuación introduciremos brevemente algunos conceptos para un mejor entendimiento del ejemplo.

En el contexto actual de la demanda eléctrica, las actuales redes eléctricas tradicionales se vieron superadas debido al incremento del consumo eléctrico además del surgimiento de conceptos y tendencias tales como generación distribuida, aplicación y uso de energías renovables, almacenamiento de energía, y gestión eficiente del consumo. Todo esto trajo aparejado la necesidad de administrar de manera más eficiente la red, dando la introducción al concepto de Smart-Grid (Cirrincione, 2009; Fang, 2012). Las Smart-Grid son, en pocas palabras, un sistema eléctrico funcionando en paralelo con uno sistema informático. Estos sistemas deben ser capaces de monitorear y controlar el estado de la red a través de diferentes dispositivos, monitores y actuadores con el objetivo de lograr una administración más eficiente el uso de la energía.

Otro concepto asociado a las Smart-Grid son las denominadas MicroGrid, las cuales hacen referencia a una porción o parte de una red que puede separarse de la

red de abastecimiento principal de energía. Una Microgrid está conformada por un sistema informático que debe ser capaz de gestionarla, un sistema de generación de energía distribuida, un sistema de almacenamiento y por supuesto el sistema eléctrico que interconecte los consumidores (viviendas, negocios, edificios públicos, etc) que conforman la MicroGrid. Las Microgrid tienen la capacidad de autoabastecerse y gestionar el consumo, de tal manera que puedan funcionar independientemente de la red principal, aunque de ser necesario, pueden llegar a conectarse a ésta ultima para satisfacer su demanda o también para proveer de energía.

Teniendo en cuenta que la tecnología de sistemas multiagentes es utilizada mayormente para el modelado de sistemas complejos es habitual que los diagramas cuenten con un alto número de conceptos definidos en los mismos. Por cuestiones de espacio, en la Figura 4 se presenta la organización MicroGrid que está constituida por cuatro roles, tres de los cuales se encuentran asociados a la simulación del funcionamiento de una MicroGrid (EnergyDevice, Regulator, Transmiter), mientras que el restante (TimeManager) se utiliza para administrar el tiempo de la simulación. En más detalle, el rol EnergyDevice representa un dispositivo que consume energía, genera o consume/genera energía en un determinado momento, tal como sucede con los sistemas de almacenamiento de energía, dado que estos consumen al momento de la carga y proveen energía en la descarga, según la red los necesite. Por otro lado, el Transmiter cumple la función de simular el conductor o red a la cual se encuentra conectados los diferentes dispositivos o EnergyDevices. El Regulator tiene la función de regular y estabilizar la energía que se encuentra "fluyendo" por el Transmiter. Es un dispositivo del tipo monitor y regulador. Por último, el TimeManager cumple con el role de ser el encargado del manejar el tiempo o velocidad actual de la simulación que se desee ejecutar.

Las reglas de validación sintácticas, presentadas en el apartado anterior, pueden ejecutarse en cualquier momento que el diseñador crea conveniente. Para nuestro ejemplo, que es un refinamiento del modelo presentado en (Basso, 2011), se detectaron en las etapa inicial de diseño un serie de errores en las distintas organizaciones que conforman el modelo general. Los errores capturados por el módulo de validación son presentados con el formato que muestra la Figura 4, en la misma se puede ver un círculo a las esquina superior derecha de cada concepto vertido en el diagrama (para mayor claridad se introdujeron en la figura letras por cada error), además de una ventana que los muestra en formato de lista.

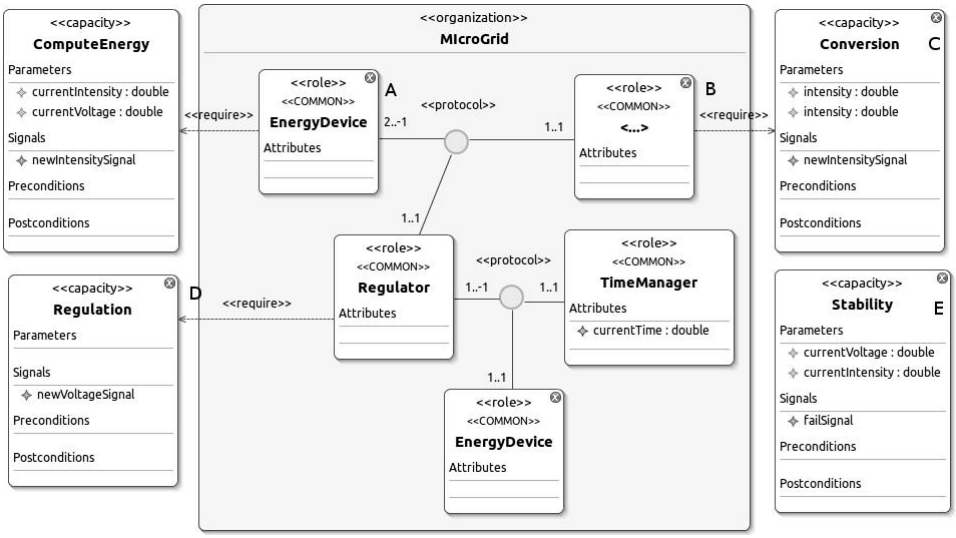

Figura 4 – Microgrid – Etapa inicial de diseño

Tabla 3 – Errores detectados en la Organización MicroGrid

	Error detectado
A	Se detectó una duplicación de nombres en los roles (EnergyDevice)
B	Falta nombre en un rol de la organización (Debería ser Transmitter)
C	Parámetros de entrada duplicado en nombre y tipo en la capacidad Conversion
D	Debe existir al menos un parámetro de entrada en la capacidad Regulation
E	La capacidad Stability no es requerida por ningún rol.

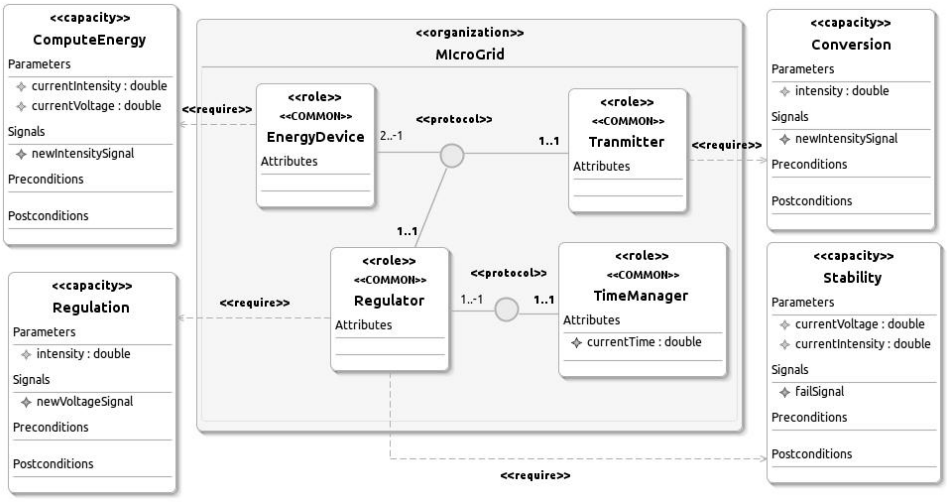

Figura 5 – Microgrid – Después de aplicar las correcciones

Una vez aplicadas las reglas de validaciones sintácticas y llevado a cabo las correcciones pertinentes la organización MicroGrid queda tal como se muestra en la Figura 5.

VII. Conclusiones

La finalidad de este artículo es presentar un mecanismo para la validación sintáctica de modelos de sistemas multiagentes basados en el enfoque organizacional utilizando Janeiro Studio. Detectar un error en etapas tempranas y de forma automática permitirá reducir drásticamente los costos de desarrollo del sistema al mismo tiempo que se incrementa la calidad del software entregado. Este mecanismo está compuesto por un conjunto de reglas de validación sintáctica que pueden ser definidas por el diseñador. Además, estas reglas serán ejecutadas sobre los diferentes diagramas de manera automatizada por la herramienta de validación.

Trabajos futuros buscarán ampliar las validaciones realizadas sobre los modelos, contribuyendo más aún a la asistencia del diseñador en el análisis, conceptualización e implementación de SMA.

Referencias

Al-Hashel, E., Balachandran, B. M., & Sharma, D. (2007). A Comparison of Three Agent-Oriented Software Development Methodologies: ROADMAP, Prometheus, and MaSE. In B. Apolloni, R. J. Howlett, & L. Jain (Eds.), *Knowledge-Based Intelligent Information and Engineering Systems* (pp. 909–916). Springer Berlin Heidelberg.

Basso, G., Hilaire, V., Lauri, F., ROCHE, R., & Cossentino, M. (2011). A MAS-based simulator for the prototyping of Smart Grids. In *9th European Workshop on Multiagent Systems* (EUMAS11).

Caire, G., Coulier, W., Garijo, F. J., Gomez, J., Pavon Mestras, J., Leal, F., … Massonet, P. (2002). Agent Oriented Analysis Using Message/UML. In M. Wooldridge, G. Wei\s s, & P. Ciancarini (Eds.), *Agent-Oriented Software Engineering II, Second International Workshop, AOSE 2001, Montreal, Canada, May 29, 2001, Revised Papers and Invited Contributions* (Vol. 2222, pp. 119–135). Springer.

Cernuzzi, L., & Zambonelli, F. (2009). Gaia4E: A Tool Supporting the Design of MAS using Gaia. In *ICEIS (4)* (pp. 82–88). Citeseer.

Cirrincione, M., Cossentino, M., Gaglio, S., Hilaire, V., Koukam, A., Pucci, M., … Vitale, G. (2009). Intelligent Energy Management System. In *Proceedings of the IEEE indin conference*.

Cossentino, M. (2005). From requirements to code with the PASSI methodology. *Agent-Oriented Methodologies*, 3690, 79–106.

Cossentino, M., Gaud, N., Hilaire, V., Galland, S., & Koukam, A. (2010). ASPECS: an agent-oriented software process for engineering complex systems. *Autonomous Agents and Multi-Agent Systems*, 20(2), 260–304.

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. I*BM Systems Journal*, 15(3), 182–211.

Fang, X., Misra, S., Xue, G., & Yang, D. (2012). Smart Grid - The New and Improved Power Grid: A Survey. *Communications Surveys Tutorials, IEEE, 14*(4), 944–980.

Ferber, J., & Gutknecht, O. (1998). A meta-model for the analysis and design of organizations in multi-agent systems. In *International Conference on Multi Agent Systems, 1998. Proceedings* (pp. 128–135).

Ferber, J., Gutknecht, O., & Michel, F. (2004). From Agents to Organizations: An Organizational View of Multi-agent Systems. In P. Giorgini, J. P. Müller, & J. Odell (Eds.), *Agent-Oriented Software Engineering IV* (pp. 214–230). Springer Berlin Heidelberg.

Gomez-Sanz, J. J., Fuentes, R., Pavón, J., & García-Magariño, I. (2008). INGENIAS Development Kit: A Visual Multi-agent System Development Environment. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems: Demo Papers* (pp. 1675–1676). Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems. Retrieved from

Hilaire, V., Koukam, A., Gruer, P., & Müller, J.-P. (2000). Formal Specification and Prototyping of Multi-agent Systems. In A. Omicini, R. Tolksdorf, & F. Zambonelli (Eds.), *Engineering Societies in the Agents World* (pp. 114–127). Springer Berlin Heidelberg.

M. Amiguet. (2003). *MOCA: un modèle componentiel dynamique pour les systèmes multi-agents organisationnels*. Université de Neuchâtel.

McAffer, J., Lemieux, J.-M., Aniszczyk, C., & more, & 0. (2010). *Eclipse Rich Client Platform* (2 edition.). Upper Saddle River, NJ: Addison-Wesley Professional.

Padgham, L., Thangarajah, J., & Winikoff, M. (2005). Tool support for agent development using the Prometheus methodology. In *Fifth International Conference on Quality Software, 2005. (QSIC 2005)* (pp. 383–388).

Pedro Araujo, Sebastían. Rodríguez. (2013). Janeiro Studio. Presented at the CONAIISI, Córdoba, Argentina.

Picard, G., & Gleizes, M.-P. (2004). The ADELFE Methodology. In F. Bergenti, M.-P. Gleizes, & F. Zambonelli (Eds.), *Methodologies and Software Engineering for Agent Systems* (pp. 157–175). Springer US. Retrieved from

Rodriguez, S., Gaud, N., Hilaire, V., Galland, S., & Koukam, A. (2007). An Analysis and Design Concept for Self-organization in Holonic Multi-agent Systems. In S. A. Brueckner, S. Hassas, M. Jelasity, & D. Yamins (Eds.), *Engineering Self-Organising Systems* (pp. 15–27). Springer Berlin Heidelberg.

Sommerville, I. (2010). *Software Engineering* (9 edition.). Boston: Addison-Wesley.

Susi, A., Perini, A., Mylopoulos, J., & Giorgini, P. (2005). The Tropos Metamodel and its Use. *Informatica* (03505596), 29(4).

Unhelkar, B. (2005). *Verification and Validation for Quality of UML 2.0 Models* (1 edition.). Hoboken, NJ: Wiley-Interscience.

Weiss, G. (2013). *Multiagent Systems* (second edition edition.). Cambridge, Massachusetts: The MIT Press.

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems* (2nd edition.). Chichester, U.K: Wiley.

Wooldridge, M., & Jennings, N. R. (1995). Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(02), 115–152.

Wooldridge, M., Jennings, N. R., & Kinny, D. (2000). The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3(3), 285–312.