Instituto de Computación
Facultad de Ingeniería
Universidad de la República

Lenguajes específicos de dominio para simulación multiagente

Informe de Proyecto de Grado

30 de junio de 2009
Montevideo - Uruguay

Autores:

A/C Horacio Blanco
A/C Diego Salido

Supervisores:

Ing. Daniel Calegari
Ing. Jorge Corral

Resumen

Los Sistemas Multiagente permiten modelar situaciones en las que, debido a la multiplicidad y heterogeneidad de actores (agentes) y su interacción en el tiempo, no es posible (o no es recomendable) una resolución algorítmica. Un campo de aplicación de este enfoque es la simulación de agroecosistemas para realizar análisis de los posibles desenlaces de un sistema al introducir cambios en el ambiente (como fluctuaciones de precios, sequías, etc.). En este contexto, la realidad es modelada utilizando el enfoque de sistemas multiagentes (representado mediante diagramas UML, principalmente de clases y actividad), para luego desarrollar una simulación con la información generada. Los problemas existentes en este enfoque son principalmente dos. En primer lugar, los diagramas utilizados no son lo suficientemente expresivos como para contemplar la particularidad del paradigma. En segundo lugar, la construcción de la simulación requiere de conocimientos informáticos debido a que se realiza manualmente, los cuales no suelen ser parte del perfil de quienes realizan este proceso.

El objetivo de este proyecto es explorar la posibilidad de utilizar un lenguaje especifico de dominio (lenguaje que ofrece mayor expresividad dentro de un dominio particular) para el modelado de las simulaciones multiagentes y simplificar el proceso de construcción de la simulación. Para lograr estos objetivos, se creó un lenguaje que ofrece mayor expresividad que UML en el dominio de simulaciones multiagentes. Ademas, se desarrolló una herramienta que toma modelos representados en dicho lenguaje y genera código para la plataforma CORMAS, que ofrece un marco tecnológico para realizar simulaciones bajo el paradigma de multiagentes. El enfoque se aplicó a un caso de estudio cuyo resultado fue el la generación de gran parte del código final de la simulación.

Índice general

Capítulo 1

Introducción

Los lenguajes de modelado, y en particular los lenguajes gráficos, han existido desde hace tiempo. Sin embargo, la complejidad que muchos de éstos presentaban (lo cual restringía su uso sólo a expertos), así como la multiplicidad de aspectos que pretendían cubrir en un solo diagrama, hacía que los mismos no tuvieran una aceptación general, ni se lograran consensos importantes.

Sin embargo, la complejidad de los sistemas actuales hace que los lenguajes de propósito general (como por ejemplo UML[40]) no tengan un soporte tecnológico adecuado, que permita aumentar el nivel de abstracción del desarrollo mediante su uso, y la posterior generación automática del código. Es en ese contexto, que dichos lenguajes se restringen a dominios más pequeños. De esta forma, el lenguaje maneja información propia de ese dominio, lo que facilita el modelado de esos sistemas, y su posterior interpretación, tanto por lo involucrados en el proyecto, como por generadores de código.

Un lenguaje específico de dominio[13] (DSL en inglés), es un lenguaje que ofrece, mediante notaciones y abstracciones adecuadas, mayor expresividad dentro de un dominio particular.

Por otro lado, los Sistemas Multiagente[41] (MAS en inglés), permiten modelar situaciones en las que, debido a la multiplicidad y heterogeneidad de actores (agentes) y su interacción en el tiempo, no es posible (o no es recomendable) una resolución algorítmica. Por tanto, se construye un sistema que modela el comportamiento (mediante reglas) de estos agentes heterogéneos, los recursos que éstos consumen o producen, así como las interrelaciones agente-agente y agente-recurso.

Actualmente existen líneas de investigación a nivel internacional (como en [6]) que combinan el modelado de una realidad (por ejemplo de una zona geográfica y sus habitantes) y la construcción de MAS, para luego realizar simulaciones de escenarios futuros. Es en esta línea que se vincula UML como herramienta de modelado y CORMAS[33] como plataforma de simulación de MAS.

Este proyecto tiene como objetivo profundizar en el conocimiento del uso de DSL para la construcción de sistemas de software. En particular, interesa explorar nuevas posibilidades de aplicación de DSL a la simulación mediante MAS. Para ello se definen como objetivos específicos:

- Definir un DSL que permita lograr mayor expresividad en el dominio.
- Desarrollar una herramienta capaz de generar código CORMAS a partir del DSL.
- Evaluar el DSL y la herramienta construida mediante el desarrollo de un caso de estudio.

El resto de este informe se divide de la siguiente manera: el Capítulo 2 contiene el estado del arte; presenta los conceptos de sistemas multiagentes, simulación y DSL sobre los cuales se basó el proyecto, así como también herramientas utilizadas en este domino. El Capítulo 3 comienza describiendo el contexto del trabajo y los problemas a resolver, para luego presentar el DSL propuesto y la herramienta construida. En el Capítulo 4 se desarrolla un caso de estudio para la evaluación del DSL y la herramienta construida. Finalmente, en el Capítulo 5 exponemos las conclusiones y enumeramos el trabajo futuro.

Capítulo 2

Estado del Arte

En este capítulo se presenta el estado del arte de los temas relacionados con los objetivos del proyecto. En primer lugar se introducen los sistemas multiagente (MAS, del inglés) como marco general del trabajo y en particular la simulación basada en multiagentes. A continuación se presenta el concepto de lenguaje especifico de dominio, y finalmente se exponen un conjunto de herramientas para el desarrollo de simulaciones basadas en multiagentes. Algunas de las mismas, se basan en lenguajes específicos de dominio. Este marco básico nos permitirá trabajar sobre un lenguaje especifico de dominio para la simulación basada en multiagentes.

2.1. Sistemas Multiagente

Si bien no hemos encontrado una única definición formal de qué son los sistemas multiagentes, la mayoría de ellas coinciden en gran parte. Tomaremos como referencia las definiciones publicadas en [41], [26] y [12], ya que son citadas en casi la totalidad de los trabajos encontrados.

Un sistema multiagentes (MAS por sus siglas en inglés), es un sistema compuesto por múltiples entidades, llamadas agentes, que conviven en un ambiente y que interactúan entre sí para cumplir sus objetivos individuales.

Un agente es un componente de software que reside en un ambiente y opera en un ciclo continuo de Percepción – Razonamiento – Acción. En la percepción, el agente recibe algún estímulo del ambiente, razona con su conocimiento interno y decide que acción tomar. Esta acción puede modificar el estado del ambiente, generando nuevos estímulos para el próximo ciclo.

Además, el agente es una entidad autónoma y flexible, donde flexible implica que el agente es reactivo (responde a cambios en el ambiente), pro-activo u orientado a objetivos (satisfaciendo sus objetivos o maximizando su utilidad) y social (para cumplir sus objetivos puede comunicarse con otros agentes).

Algunos autores hablan que los agentes cumplen Roles. Si bien no existe una definición sencilla de rol, éste se puede ver como la función que cumple un agente en un proceso (con otros roles / agentes). Las principales características de un rol son las siguientes:

1. Si un agente cumple con un rol, se tienen ciertas expectativas sobre su comportamiento. Se puede ver un rol como un conjunto de acciones.
2. Pueden existir dependencia entre roles. Por ejemplo, el rol “profesor” no tiene sentido sin la existencia del rol “alumno”.
3. Un agente puede cumplir varios roles al mismo tiempo.

Hemos dicho que los agentes interactúan con el ambiente. El ambiente puede cumplir con las siguientes propiedades:

- Accesible o No accesible: es accesible si la información que los agentes extraen de él es exacta (completa, actualizada y sin errores), no accesible en caso contrario.
- Determinista o No Determinista: es determinista si una acción de un agente sobre el ambiente tiene un único resultado conocido, no determinista en caso contrario.
- Episódico o no Episódico: es episódico si los agentes determinan la acción a realizar, basados solamente en el estado actual. no episódico en caso contrario.
- Estático o Dinámico: es estático si el ambiente es afectado exclusivamente por las acciones del agente, es dinámico si se encuentran otros procesos actuando sobre el mismo y modificándolo.
- Discreto o Continuo: es discreto si existe un número acotado de acciones y percepciones que los agentes pueden obtener, continuo en caso contrario.

Para la construcción de los sistemas multiagentes existen tres arquitecturas abstractas básicas para agentes: reactivos, deliberativos e híbridos. En una arquitectura deliberativa, los agentes cuentan con un sistema central de razonamiento donde se generan planes para conseguir sus objetivos (agentes orientados a objetivos). En una arquitectura de agentes reactivos, los agentes no mantienen un modelo interno, seleccionan la acción utilizando el estado actual del mundo como índice en una tabla de acciones. En una arquitectura híbrida, cada agente posee un subcomponente de agente deliberativo y uno reactivo, con un componente de control que decide cuando usar uno u otro.

Aplicaciones

Algunas empresas han adoptado la tecnología multiagentes para el desarrollado de sus soluciones. Un ejemplo de ellas es Whitestein Technologies ([37]), cuyos productos basados en MAS brindan soluciones para telecomunicaciones, logística y líneas de producción y manufactura. Por ejemplo, el sistema LS/ATN (Living Sytems® Adaptive Transportation Networks) ([38]) provee optimización automática y soporte de expedición para FTL (Full Truck Load), LTL (Less than Truck Load) y otras redes de transporte.

El modelado basado en agentes puede ser utilizada, entre otras cosa, como una herramienta para el estudio de sistemas sociales complejos como en [19]. Esta es utilizada cuando interesa observar cómo a través del comportamiento de agentes individuales (micro) emergen propiedades a nivel del sistema (macro). Al usar esta tecnología se puede comprobar diferentes hipótesis sobre los atributos, comportamientos y tipos de interacciones entre agentes, al igual que su efecto a nivel macro.

Además de estos escenarios, la tecnología de agentes se puede encontrar en robótica, comercio electrónico, inteligencia artificial (de donde ha surgido buena parte de la teoría), internet y simulación. BotBox ([1]), por ejemplo, comercializa un producto que ofrece un conjunto de agentes de monitoreo de información que automatizan las tareas de obtener y filtrar información relevante. Al producto se le especifican las fuentes de información y los agentes las monitorean constantemente.

Otros sistemas basados en agentes para internet son aquellos destinados a realizar subastas ([2]). Estos agentes representan a los usuarios, realizando ofertas automáticamente según las preferencias de estos.

También existen agentes con el objetivo de comparar precios de un mismo producto en varios sitios, filtrando la información por el usuario.

En el área de inteligencia artificial distribuida también se encuentra un área que utiliza sistemas multiagentes. Aquí varios agentes se coordinan para razonar sobre el proceso de resolución de un problema.

La tecnología multiagentes también ha sido adoptada en el área de simulación de agroecosistemas. Un ejemplo de esto se puede ver en [6], donde se utilizan agentes para modelar las estrategias de productores en Uruguay, desarrollando un modelo que permite la simulación de la evolución

de los distintos tipos de productores y el uso de tierras, realizando una simulación exploratoria y prospectiva.

Investigación y Desarrollo

Los sistemas multiagentes han sido objeto de estudio en distintas disciplinas y por diferentes organizaciones durante años. Dentro de la gran variedad de empresas que vienen realizando investigaciones en la materia, se encuentran HP Labs ([23]) e IBM. HP Labs publicó el primer reporte en 1993, entre los años 1996 y2005 han publicando informes de forma continua, con una mayor intensidad del 2000 al 2003. En algunas de estas publicaciones ha participado Michael Wooldridge, un investigador muy recurrente entre las referencias de reportes técnicos sobre Multiagentes. Wooldridge ha publicado más de 200 artículos y 13 libros sobre la teoría y práctica de sistemas basados en multiagentes.

Una de las áreas de investigación ha sido el desarrollo de metodologías para construir sistemas multiagentes. Dentro de las metodologías que surgen al comenzar a investigar en el área se encuentran [42], [18], [27], [5], [28], entre otras.

Investigaciones en el desarrollo de sistemas Multiagentes han llevado a la construcción de patrones, es decir, la identificación de problemas comunes y la elaboración de soluciones elegantes para ellos. En [21] se presenta un patrón para arquitecturas de agentes en capas.

También se puede encontrar bibliografía de buenas prácticas para el desarrollo de sistemas multiagentes, como en [20], donde se provee una colección de ellas.

Desde el punto de vista del modelado, también surgió la necesidad de contar con una notación estándar para representar la realidad, tanto el modelo como las interacciones entre agentes. Existen soluciones a este problema basados en UML, como por ejemplo AUML, MAS-ML, AML, entre otras. Describiremos las tres mencionadas, ya que son las más significativas para nuestro proyecto. AUML por ser propuesta por FIPA, MAS-ML por los conceptos introducidos y AML por evolucionar de éstos.

AUML (Agent Unified Modeling Language) ([24], [16]) es un Profile[1] UML propuesto por FIPA ([36]) que extiende UML agregando restricciones y estereotipos necesarios para MAS. Inicialmente se profundizaron dos especificaciones: diagramas de clases y diagramas de interacción. Los diagramas de interacción fueron los primeros tipos de diagramas que se diseñaron en AUML y se basaron en los diagramas de secuencia de UML. Los diagramas de clases utilizan una abstracción de la definición de agente para cubrir todos los tipos de agentes.

MAS-ML (Multi-Agent System Modeling Language) ([7]) extiende el meta-modelo UML describiendo nuevas meta-clases y estereotipos, extendiendo los diagramas de clase y secuencia, agregando también nuevos diagramas. En este lenguaje, los Agentes, Roles y Ambiente no son representados con estereotipos, sino que pertenecen a nuevos elementos del meta-modelo. A causa de algunas particularidades en la definición del meta-modelo, es difícil implementar una herramienta gráfica basada el UML para MAS-ML.

AML (Agent Modeling Language) [39] es un lenguaje ideado para especificar, modelar y documentar sistemas basados en multiagentes. Se encuentra especificado como una extensión de UML 2.0, donde se extiende el meta-modelo UML con clases particulares para multiagentes. AML es el lenguaje más moderno encontrado, e incorpora conceptos existentes de lenguajes anteriores. Se permiten modelar tanto los aspectos estáticos como dinámicos del sistema, ofreciendo para ambos extensiones y estereotipos, definiendo también iconos para las principales entidades (como agente o recurso).

Un punto que parece no estar del todo claro entre los distintos enfoques y herramientas para sistemas multiagentes es la representación del ambiente. Si bien existe un consenso general que el

[1] Un Profile UML es una especificación que personaliza UML para un dominio específico.

ambiente es esencial para este tipo de sistemas, algunos no integran el ambiente como un concepto primario en los modelos y herramientas para MAS. Por ejemplo Jade [14], una plataforma popular para MAS, reduce el ambiente a un sistema de mensajería entre agentes.

2.2. Simulación

Dado que el foco de nuestro proyecto se centra en simulaciones basadas en multiagentes, introduciremos parte de los conceptos de simulación sobre los cuales nos hemos basado.

Algunos autores definen la simulación como la imitación de un proceso o sistema del mundo real a través del tiempo ([30]). Es experimentar con un modelo del sistema real bajo estudio y medir a éste en vez del sistema en sí. Otros lo definen como el proceso de experimentar con el modelo del sistema programándolo en un computador. Estos autores se refieren con modelo a una representación del sistema a simular. El tipo de simulación depende de la forma en la que avance el tiempo, la granularidad de los elementos, los objetivos del modelo, etc.

Si clasificamos las simulaciones respecto al avance del tiempo, existen tres categorías a saber: simulación continua, simulación basada en eventos discretos y time-step.

Simulación continua refiere a las simulaciones donde el paso del tiempo puede ser reducido a intervalos arbitrariamente pequeños. Este tipo de simulaciones utiliza un modelo basado en ecuaciones diferenciales, las cuales describen el comportamiento del tiempo del sistema.

Simulación basada en eventos discretos es un tipo de simulación donde el entorno de simulación administra una lista de eventos a ejecutar. El ambiente remueve el primer elemento de la cola y lo ejecuta (el cual puede generar nuevos eventos a encolar). Luego de la ejecución se mueve el tiempo al siguiente evento de la lista.

Time-stepped es un caso particular de eventos discretos, donde el modelo se ejecuta cada cierta cantidad fija de tiempo.

La simulación también se puede clasificar según el nivel de detalle del modelo en micro simulación o macro simulación, aunque existen también aproximaciones intermedias.

Macro simulación modela el sistema en su totalidad. Aquí se utiliza la perspectiva del sistema en su totalidad.

Micro simulación modela cada una de las entidades "pequeñas" por las cuales está compuesto el sistema. Cada una de las entidades puede tener estado y comportamiento diferentes.

Los sistemas de simulación, bajo cualquiera de las anteriores clasificaciones, pueden ser utilizados para distintos objetivos, entre ellos:

- Para predecir eventos: mostrar uno de los posibles escenarios en base a la teoría.
- Explicar eventos: intentar saber porqué pasó algo.
- Observar respuestas del sistema: en un sistema se introduce un evento para ver la manera en que éste responde. Por ejemplo se podría simular el sistema de emergencias frente a un evento de catástrofe.
- Observar resultados: cuando se quiere observar el resultado de un proceso que en la vida real toma demasiado tiempo o es muy costoso. Un ejemplo podría ser la evolución de la vida de una estrella (a diferencia del punto anterior, no podemos ponerlo a prueba para ver el resultado).

- Modelar sistemas complejos: cuando el comportamiento de un sistema es demasiado complejo para modelar matemáticamente, éste es simulado para poder predecir su comportamiento. Si existiera un modelo matemático, se podrían ver los escenarios futuros cambiando el valor de las variables.

Simulación Basada en Multiagentes

La simulación basada en multiagentes (MASS) es un sistema multiagente en un ambiente simulado con tiempo virtual, donde las entidades (de simulación) son los agentes, el ambiente para los agentes consiste de otros agentes y otras entidades (recursos, ambiente, etc). Las interacciones entre los agentes son el punto central de la simulación ([22]). Una simulación multiagentes es una forma especial de micro simulación, donde se puede utilizar cualquier forma de avance del tiempo. También se puede ver como una simulación donde se modeló el sistema con el enfoque de multiagentes.

La simulación multiagentes ofrece un nivel de abstracción mayor que la simulación procedural (PASCAL-SIM). En este nivel de abstracción aparecen conceptos bien definidos como los Agentes y Recursos que se acercan más a la realidad, haciendo más sencillo el entendimiento y desarrollo.

Toda simulación implementada bajo otro paradigma también se puede implementar bajo la simulación multiagente. Sin embargo, se logran mayores beneficios en las áreas donde hayan múltiples tipos de agentes interactuantes, o donde se quiere experimentar con una gran cantidad agentes sabiendo el comportamiento individual. El objetivo es ver el comportamiento de la sociedad.

Su objetivo puede ser para predecir o explicar sucesos. Por tal motivo ha sido utilizada para la investigación en distintas áreas como ser: administración de recursos (agua, madera, etc), utilización del terreno y cambios sociales (Land Use and Land Cover Change), difusión de enfermedades epidémicas, noticias, etc. En particular, existe una línea de investigación que modela una realidad como un sistema multiagente, utilizando UML como lenguaje de modelado, para finalmente implementar dicho sistema sobre la plataforma para simulación multiagente CORMAS y ejecutarla con distintos parámetros de entrada.

2.3. Lenguajes Específicos de Dominio

Según Fowler ([13]), los lenguajes específicos de dominio (DSL - Domain Specific Language) son lenguajes de programación con expresividad limitada enfocados a un dominio particular. También se los puede encontrar en la literatura bajo los nombres de pequeños lenguajes o micro lenguajes. Se crean para resolver problemas puntuales de un área o dominio específico, como ser, seleccionar lineas de texto (como grep[2]), definir interacciones celulares [15], despliegue de contenido en una página web (HTML o CSS), o realizar consultas sobre una base de datos relacional (SQL). Su poder de expresión esta limitado al dominio para el que fueron creados. Notar que esta definición permite que una biblioteca de un lenguaje de propósito general sea considerada un lenguaje especifico de dominio.

Se distinguen entre lenguajes internos y externos. En los lenguajes internos se utiliza un lenguaje base y se lo hace parecer a otro lenguaje más específico. Un ejemplo podría ser una biblioteca en Java. Los lenguajes externos tienen su propia sintaxis y se crean compiladores específicos para ellos. Esta es una practica común en la comunidad Linux. Los DSLs externos pueden ser implementados para ser interpretados o generar código. Usualmente el código que generan es de un lenguaje de alto nivel como Java.

Los DSL visuales son aquellos que son representados gráficamente. Permiten a expertos del dominio (médicos, agrónomos, etc) validar los diagramas o modelos realizados sin necesidad de aprender o entender un lenguaje ajeno a su realidad. Los DSL textuales son aquellos que se representan utilizando una sintaxis particular en un archivo de texto. Están más orientado a progra-

[2]Grep es un comando para encontrar lineas de texto, originalmente escrito para el sistema operativo Unix.

madores, donde se les ofrece construcciones concretas del dominio de la aplicación, generalmente disminuyendo la cantidad de líneas de código a escribir.

Las ventajas de este tipo de lenguajes son:

- Ofrece mayor expresividad dentro de un dominio.
- Aumenta la productividad.
- Permiten a expertos del dominio participar del desarrollo sin necesidad de aprender un lenguaje ajeno a su realidad.

2.4. Herramientas

Ahora veremos las herramientas disponibles para atacar los distintos enfoques del proyecto, verificando la existencia de una que abarque parte o la totalidad de nuestro objetivo. La búsqueda se enfoca a una herramienta de software capaz de facilitar el desarrollo de simulaciones utilizando la tecnología de multiagentes, que sea adecuada para agroecosistemas (se necesitan agentes, recursos naturales y el ambiente como grilla).

Luego de listar las herramientas encontradas, las clasificaremos según algunos criterios de interés con el objetivo de observar claramente qué herramienta satisface qué parte del proyecto, y decidir si se extenderá alguna de ellas u obtener ideas para la construcción de una nueva.

2.4.1. Listado

Para realizar la categorización de las herramientas, dividimos sus propiedades en tres partes: una sobre la herramienta, otra sobre su forma de realizar el modelado multiagente y por último sus capacidades con respecto a la simulación.

De las características de la herramienta, tomamos nota de las siguientes propiedades:

Tipo Indica si es una herramienta para realizar simulación o si es una biblioteca para la construcción de sistemas multiagente.

Extensibilidad Aquí podríamos encontrar algunas con puntos de extensión definidos (API, biblioteca), y otras cuya extensibilidad no está planeada.

Licencia El tipo de licencia (Open Source, propietaria, etc.).

Documentación Cantidad y calidad de la documentación.

Para el modelado multiagente nos interesa:

Modelado Forma en que se especifica el modelo (código, diagramas, etc.).

Soporte Gráfico Si presenta soporte gráfico para el mismo (por ejemplo, si gráficamente se puede acelerar la manera de realizar el modelado).

Representación La manera en que se representa el modelo o las capacidades de exportación (por ejemplo XML).

En caso de tener soporte para simulación, nos interesa:

Tiempo Forma de avance del tiempo (tiempo continuo, discreto o time-step).

Construcción Cómo se construye la simulación a partir del modelo (código, construcciones gráficas, etc.).

Soporte Gráfico Si existe representación gráfica de la corrida.

Salida Los distintos tipos de salida que se pueden obtener (como gráficas, archivos de datos, etc.).

Soporte Si da soporte para reproducir una misma simulación ya ejecutada, aunque exista decisiones tomadas de manera aleatoria.

De todas las herramientas evaluadas, las más destacadas son CORMAS, Repast y SeSAm. Describiremos brevemente cada una de ellas, mencionando qué características las hacen sobresalir.

CORMAS ([33]) está enfocado hacia los modelos de administración de recursos naturales renovables. Es una plataforma de simulación multiagente que ofrece un conjunto de clases base para dar el soporte al paradigma de multiagentes (agentes, recursos, ambiente). Provee un ambiente gráfico para correr las simulaciones. Es utilizada como plataforma de simulación en la línea de investigación que dio origen al proyecto.

Repast ([3]) es una herramienta para el modelado, implementación, y ejecución de simulaciones basadas en multiagentes. Presenta un entorno gráfico para la especificación del modelo y el comportamiento de los agentes. Para el comportamiento, además de construcciones gráficas, se deben especificar detalles de implementación en código, escrito en una tabla de propiedades, haciéndolo incómodo de utilizar. El código de la simulación es generado automáticamente a partir del modelo ante cualquier cambio del mismo.

SeSAm ([11]) provee un ambiente genérico para el modelado y experimentación con simulaciones basadas en agentes. Provee un ambiente gráfico donde se define el modelo y el comportamiento de las entidades. La implementación de la simulación es puramente gráfica (no se escribe código), sin embargo esto hace que el desarrollo de una simulación sea muy engorroso, ya que los detalles de implementación se realiza con muchas construcciones gráficas.

2.4.2. Categorización

En la tabla 2.1 resumimos todas las herramientas encontradas, de tal forma que sea más sencillo encontrar la que mejor se adecue al proyecto.

Por cada herramienta indicamos si tiene elementos del dominio de multiagentes. Para el caso que lo sean, se indica si el modelado de la realidad se realiza gráficamente. También indicamos si se trata de una herramienta de simulación, y si posee soporte gráfico para el mismo. Algunos proveen una API (u otro mecanismo) que ayudan al desarrollador a programar la interfaz gráfica, estos se denotan con 'prog'. Otro aspecto destacado es la extensibilidad, algunos la ofrecen explícitamente, pero para otros es indirecto (indicados por 'ind'), donde requiere por ejemplo modificación del código fuente. Ninguna presenta soporte directo para agroecosistemas, pero una de ellas da soporte indirectamente siendo adecuada para recursos naturales (denotada con 'rn'). Los casilleros en blanco significan que la herramienta no posee dicha característica o no aplica.

Nombre	Multiagentes	Modelado Gráfico	Simulación	Simulación Gráfica	Extensibilidad	Agroecosistemas
CORMAS	SI		SI	SI	(ind)	(rn)
SeSAm	SI	SI	SI	SI	(ind)	
Repast	SI	SI	SI	SI	SI	
Mimosa		SI	SI	SI	SI	
Manson	SI		SI	SI	SI	
Breve	SI		SI	SI	SI	
Jade	SI				SI	
Ingenias	SI				SI	
Swarm	SI		SI	(prog)	SI	
MAML	SI		SI	(prog)	(ind)	
Netlogo	SI		SI	(prog)	SI	

Cuadro 2.1: Categorización de herramientas evaluadas

En la tabla se puede ver claramente que ninguna de las herramientas analizadas permite asistir en el desarrollo de simulaciones basadas en agentes para agroecosistemas de manera gráfica. Sin embargo, CORMAS posee un dominio que coincide en gran parte con el dominio de la simulación de sistemas agrícolas. Por dicho motivo será la base para implementar nuestra herramienta, pudiendo ser intercambiada por otra en el futuro.

Capítulo 3

Simulación Multiagente de Agroecosistemas

El presente capítulo se divide en cuatro secciones. En la primera presentamos el área de trabajo del proyecto (simulación multiagente de agroecosistemas), las herramientas que se utilizan, y cómo se lleva adelante la construcción de las simulaciones. Luego introducimos las características generales del lenguaje a construir. Continuamos con la descripción del lenguaje propuesto y por último la herramienta construida.

3.1. Contexto del Trabajo

Actualmente existen líneas de investigación a nivel internacional [6] que combinan el modelado de una realidad (por ejemplo de una zona geográfica y sus habitantes) y la construcción de una o varias simulaciones de sistemas multiagentes para luego analizar posibles escenarios futuros. Es en esta línea que se vincula UML como herramienta de modelado y CORMAS como herramienta de simulación.

El procedimiento habitual en esta área es comenzar con la elaboración de un modelo conceptual UML. En él se muestran las entidades principales del dominio y sus relaciones. Luego se generan diagramas de actividad UML para la especificación del comportamiento de los agentes involucrados. Ambos tipos de diagramas sirven en la comunicación entre el programador y los expertos del dominio. Luego de establecido el modelo y el comportamiento de los agentes, el programador debe traducir los diagramas a la plataforma CORMAS.

3.2. Problema a Resolver

La metodología utilizada para construir las simulaciones en este contexto tiene algunas deficiencias. Entre ellas se destaca que el modelado no es específico de dominio. Se utiliza un lenguaje (UML) que no posee construcciones para el modelado ni de simulaciones ni de MAS. Otro problema es la necesidad de poseer conocimientos informáticos para la construcción de la simulación.

Este proyecto tiene como objetivo profundizar en el conocimiento del uso de DSL para la construcción de sistemas de software. En particular, interesa explorar nuevas posibilidades de aplicación de DSL a la simulación mediante MAS. Es de interés construir un lenguaje específico de dominio para las simulaciones basadas en multiagentes. Deberá ser adecuado para simulaciones de agroecosistemas, lo que implica que se requiere construcciones para agentes, recursos y ambiente.

Los agentes de nuestro dominio llevan a cabo sus acciones para alcanzar su objetivo en cada paso de la simulación. A éste método se le llama 'step' y se le invoca una sola vez por agente por

paso de la simulación (time-step). El resultado final de la simulación puede diferir según el orden en que actúen los agentes. Quien determina el orden en que se hace esto se le llama 'scheduler' de la simulación. Como toda simulación, en las simulaciones multiagente interesa ver la evolución de determinados atributos. En CORMAS a estos atributos se le llama 'probe'.

Además deberá presentar construcciones más expresivas en el dominio. En un diagrama de clases es difícil ver si una entidad es un agente, un recurso o una celda del ambiente. Se busca poder representar dichas entidades de tal forma que su interpretación sea más directa.

En una simulación de agroecosistemas, a diferencia de otro tipo de simulaciones, el ambiente de simulación puede ser representado como un plano 2D representando un terreno (grilla 2D formado por NxM celdas). En él habitan agentes y recursos, los cuales pueden ser localizados (es decir, están presentes en una celda del ambiente) o no localizados (no tienen ubicación).

Se deberá crear una herramienta, que en base a nuestro lenguaje, minimice el trabajo de un programador en el pasaje desde el modelado gráfico a la construcción de la simulación generando código ejecutable en CORMAS.

3.3. DSL4MASS

3.3.1. Presentación del Lenguaje

El lenguaje esta pensado para modelar las simulaciones basadas en multiagentes. Permite representar tanto la estructura estática como la componente dinámica de la realidad, así como también incluir información útil para la posterior simulación. Este DSL no tiene como objetivo especificar la totalidad de la simulación, no es apto para detallar minuciosamente el comportamiento de los agentes.

UML es utilizado en el área como lenguaje de modelado, por lo que decidimos realizar una extensión de él para restringirlo a nuestro dominio. Por otro lado ya existen lenguajes que extienden UML para el modelado de sistemas multiagentes (no simulación). AML es el que mejor satisface las necesidades de modelado multiagente, dado que ofrece construcciones específicas para agentes, recursos y ambiente. Por tal motivo modificamos AML para incluir las restricciones de simulaciones.

De esta manera, generamos un lenguaje gráfico específico para nuestro dominio basado en una extensión de UML ampliamente utilizada para sistemas multiagentes, al cual le llamamos DSL4MASS (DSL para MASS). No es nuestro objetivo que este lenguaje permita realizar la especificación total de la simulación. Nos enfocamos en un lenguaje visual que no abrume al programador con detalles de implementación a la hora del análisis y diseño.

Además de la representación gráfica del lenguaje, diseñamos una representación textual, ofreciendo las mismas construcciones, que permitirá a la herramienta generar el esqueleto de código de la simulación.

3.3.2. Diagramas Definidos

De los diagramas ofrecidos por AML, nuestro lenguaje agrega construcciones a dos de ellos. El primero, el diagrama de clases, sirve para representar la estructura estática de la realidad, el cual llamaremos diagrama de estructura. El otro para representar el comportamiento, basado en los diagramas de actividad UML.

El diagrama de estructura nos permite representar las diferentes entidades de la realidad, sus atributos, métodos y relaciones. También permite declarar información específica de simulaciones como ser probes, step, y el scheduler. Este tipo de diagrama sólo permite incluir entidades de tipo agente, recurso o ambiente (una celda). Los distintos tipos de entidades se identifican utilizando estereotipos.

Los diagramas de actividad permiten especificar el comportamiento de las entidades, como ser, el step de un agente, el scheduler de un modelo o un método cualquiera de un agente, recurso,

ambiente (celda) o del modelo. Aquí no se introducen grandes cambios al diagrama de actividad UML.

3.3.3. Construcciones del DSL

Estructura Estática

El lenguaje distingue entre diferentes tipos de entidades. Las principales son Agentes, Agentes Localizados, Recursos, Recursos Localizados, Celda y Modelo, los cuales se especifican mediante estereotipos UML. Que un agente o recurso sea localizado, implica que se encuentra en una celda de la grilla. Tanto los agentes como los recursos pueden movilizarse entre las celdas según avanza el tiempo.

Una celda es una componente de la grilla bidimensional. Dado que en una grilla no pueden existir distintos tipos de celda, en un modelo sólo puede existir una entidad con dicho estereotipo. Dado que se modifica el concepto respecto a AML, también se cambia el icono que lo representa. Se puede ver un ejemplo en la figura 3.1 observando la entidad 'MiCelda'.

Los agentes corresponden al concepto de Agente introducido anteriormente. En AML no existe el concepto de agente localizado, por lo que se agregó en nuestro lenguaje. Los agentes perciben, razonan y actúan en cada paso de la simulación (step). Se modifica la especificación de AML para incluir los elementos de simulación step y probe. En nuestro lenguaje, los agentes pueden especificar el método step, el cual se anota con el estereotipo 'step'. La firma de un método step no puede tener ni parámetros ni valor de retorno, ya que es invocada por el 'scheduler', que no necesariamente se tiene control sobre él. Un ejemplo se puede ver en la figura 3.1 con las entidades 'MiAgente' y 'MiAgenteLocalizado'.

Los recursos son entidades pertenecientes a la realidad, pero que no persiguen ningún objetivo. Al igual que con agentes, se agregó a AML el concepto de recurso localizado. Por esta razón, este tipo de entidades no tienen step, pero sí pueden contener 'probes'. También se modificó el icono de representación para que sea más adecuado para agroecosistemas. En el ejemplo de la figura 3.1 se puede observar 'MiRecurso' y 'MiRecursoLocalizado'.

Una entidad Modelo se utiliza para mostrar cuestiones generales a toda la realidad, como variables globales, así como también aspectos de simulación como el Scheduler (también indicado con un estereotipo en un método) y 'probe'.

Estructura Dinámica

Para los diagramas de actividad se agregó la construcción de ciclos (loop), la cual permite representar de manera explícita la ocurrencia de un ciclo controlado por una condición booleana, una iteración sobre una colección de elementos o una cantidad fija de pasos. Además, se restringieron las condiciones booleanas, las cuales pueden ser del tipo: invocación a método (que retorna verdadero o falso), operaciones sobre elementos booleanos (AND, OR, NOT) ó comparación de expresiones aritméticas. Las expresiones aritméticas pueden ser comparaciones (menor, menor o igual, mayor, mayor o igual, distinto, igual) entre: constantes, atributos (de tipo numérico), llamadas a métodos o resultados de operaciones aritméticas sobre los anteriores.

Para facilitar el entendimiento de los diagramas y la traducción, se restringió la construcción de las decisiones a sólo dos posibles salidas según una condición booleana (las mismas utilizadas para los ciclos).

El envío de mensajes entre los diferentes elementos del modelo se realiza a través de la misma construcción que en AML. Por ejemplo, el envío de un mensaje de un agente a un recurso asociado a él, se traduce a la invocación de un método del recurso.

3.3.4. Representación de las Construcciones

En la siguiente tabla se resumen los distintos estereotipos permitidos en un diagrama de estructura estática y a qué son aplicables.

Entidad	Estereotipo	Aplica a:
Agente	<<Agent>>	clases
Agente Localizado	<<LocatedAgent>>	clases
Recurso	<<Resource>>	clases
Recurso Localizado	<<LocatedResource>>	clases
Celda	<<Cell>>	clases (una por realidad)
Modelo	<<Model>>	clases (una por realidad)
Step	<<step>>	métodos (de agente, sin parámetros ni retorno)
Probe	<<probe>>	métodos y atributos
Scheduler	<<scheduler>>	métodos (de Modelo)

Cuadro 3.1: Estereotipos definidos

La figura 3.1 es un ejemplo de estructura estática, conteniendo algunas de las construcciones soportadas por el DSL. En la figura 3.2 se muestra un ejemplo de estructura dinámica, utilizando las construcciones de ciclos, condición e invocación, junto a las clásicas de UML.

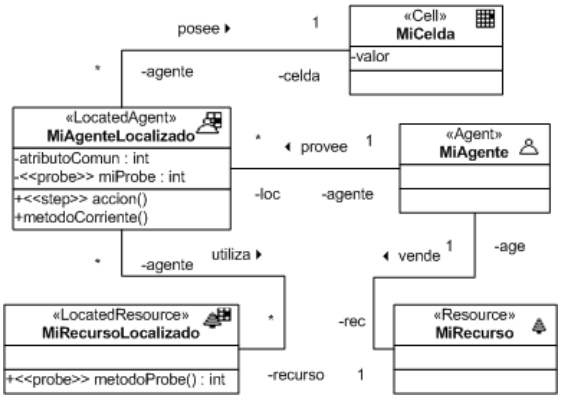

Figura 3.1: Estructura estática.

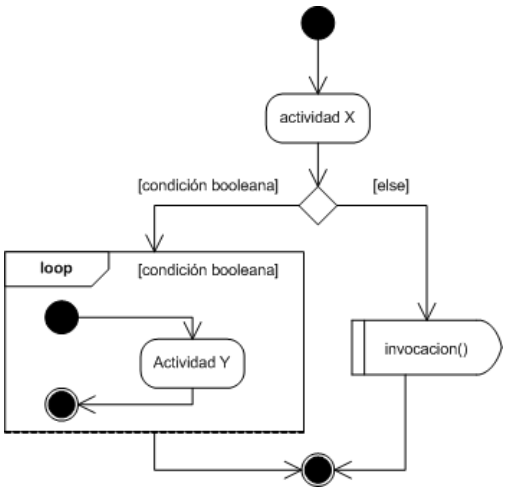

Figura 3.2: Estructura dinámica.

3.3.5. Representación Textual

Todas las construcciones de los diagramas descritos anteriormente tienen una representación textual, que sirve como lenguaje intermedio entre una herramienta gráfica y el generador de código. Mostraremos la sintaxis mediante ejemplos.

Se puede ver el elemento 'MiAgenteLocalizado' de la figura 3.1en el extracto de la estructura estática.

Como se ve, un agente se especifica con la palabra clave 'Agent'. De manera similar, los recursos utilizan 'Resource' y las celdas 'Cell'. Los recursos y agentes localizados llevan la palabra 'Located' antes de la especificación.

Extracto 1 Estructura estática textual

```
Located Agent MiAgenteLocalizado {
   association posee toMany MiCelda celda;
   association provee toOne MiAgente agente;
   association utiliza toMany MiRecursoLocalizado recurso;
   instance var int atributoComun;
   instance var int miProbe isProbe;
   instance method void step() isStep { }
   instance method void metodoCorriente() { }
}
```

Dentro del extracto de la estructura dinámica se puede ver especificada el diagrama de la figura 3.2, donde se muestran todas las construcciones.

Extracto 2 Estructura dinámica textual

```
ActivityFlow for method step of Entity UnAgente {
  start destination actividadX;
  activity actividadX destination boolCond;
  condition boolCond
    trueDestination loopElem
    falseDestination callElem
    with (bool) condicionBooleana();
  call callElem destination END
    target invocacion();
  loop loopElem destination END
    with (bool) condicionBooleana() {
      start destination actividadY;
      activity actividadY destination END;
  };
}
```

3.4. Herramienta Construida

Además del lenguaje propuesto, se desarrolló una herramienta para la traducción del lenguaje intermedio a código ejecutable en el entorno CORMAS.

En la figura 3.3 se muestran el flujo de trabajo para el desarrollo de una simulación con DSL4MASS. A diferencia de la metodología actual, mostrada en la figura 3.4, cada paso genera artefactos que son utilizados en la siguiente fase.

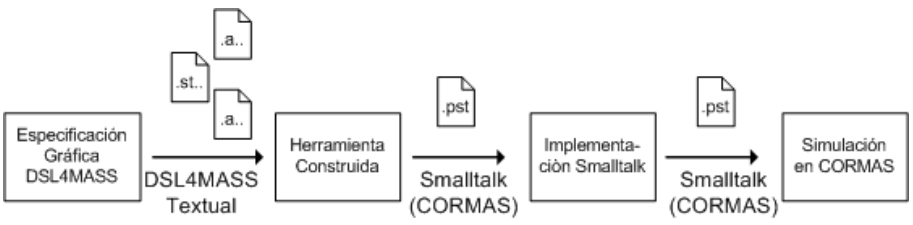

Figura 3.3: Implementación con DSL4MASS

Esta herramienta toma como entrada una realidad especificada en el lenguaje DSL4MASS. Este modelo sufre una serie de transformaciones y validaciones para luego generar un esqueleto ejecutable por una plataforma de simulación específica. Si bien la herramienta se diseño para ser fácilmente adaptable a cualquier otra plataforma, se implementó la salida sólo para CORMAS. Esta salida sirve de entrada al programador para generar la simulación final.

Figura 3.4: Implementación tradicional

El modelo que se toma como entrada debe estar especificado utilizando el lenguaje textual. Éste debe contener la especificación estructural de la realidad (agentes, recursos, relaciones, etc.) y opcionalmente la especificación del comportamiento de las distintas entidades, como los step de los agentes, el scheduler del modelo o cualquier otro método (no necesariamente razonamiento de agentes).

La especificación de estructura no necesariamente debe contener la definiciones de todos los métodos utilizados en las actividades. La herramienta se encarga de generar los esqueletos de métodos utilizados que no se hayan definido.

Las principales actividades que se pueden especificar son las acciones de los agentes en cada paso de simulación (step), inicializaciones y el orden en que actúan las diferentes entidades en cada paso.

La entrada pasa por diferentes transformaciones y validaciones, como se muestra en la figura 3.5, para llegar al resultado final. Estos pasos son:

1. Interpretación de los diagramas. Se genera una estructura básica del modelo de entrada, haciendo sólo validaciones sintácticas.

2. Validaciones semánticas. Se genera una nueva estructura intermedia más rica que la anterior, con la cual se realiza una serie de validaciones independientes de la plataforma de salida. Estas incluyen la validación de las transiciones, que exista el destino de una actividad; que se encuentren accesibles, visibilidad dentro de los ciclos; existencia de métodos invocados, creados automáticamente si no existen; entre otros.

3. Generación de código. Se genera el esqueleto de código para una plataforma de simulación específica. Dicha generación puede incluir un proceso de validación extra según la plataforma. En nuestro proyecto, esta fase es la encargada de generar comportamiento por defecto para las entidades en caso de no estar especificada en la entrada. Esto se debe a que CORMAS exige que todos los agentes tengan método de inicialización y de step.

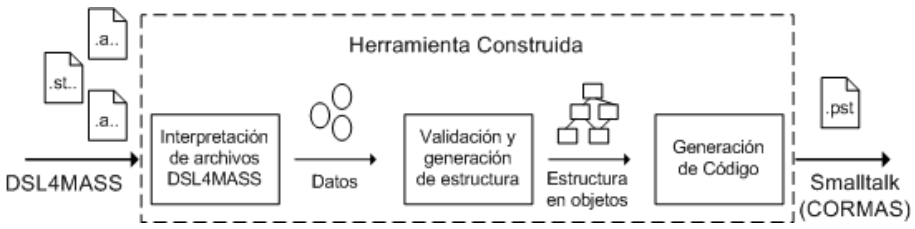

Figura 3.5: Diseño de la herramienta

El generador de código nunca podrá generar la totalidad de la simulación debido a que el DSL no tiene las construcciones necesarias para hacerlo (por ejemplo no hay forma de modificar el estado interno de las entidades). Por esta razón siempre se requiere de un programador para completar la simulación. Esto se debe a que se buscó disminuir el trabajo de un programador, automatizando el pasaje entre la fase de modelado y la de implementación. Se vio en el estado del arte que herramientas que permiten especificar la simulación completamente de manera gráfica, resultan ser engorrosas.

Capítulo 4

Caso de Estudio

A los efectos de evaluar el lenguaje DSL4MASS así como también la herramienta construida, desarrollamos un caso de estudio y lo comparamos con una simulación ya implementada con el método tradicional.

El capítulo comenzará explicando la realidad del caso de estudio. Luego se muestra el modelo en UML y en DSL4MASS para observar las diferencias. Finalmente se muestra el procedimiento empleado para llegar a la simulación ejecutable en ambos enfoques, mostrando aquí también las diferencias.

4.1. Descripción

El caso de estudio se encuentra definido en [6], donde se toma como realidad los sistemas agrarios de Uruguay y Argentina. El objetivo del trabajo original consistía en:

- Identificar y modelar las estrategias seguidas por los diferentes tipos de productores en Uruguay.
- Desarrollar un modelo que permita simular la evolución de los diferentes tipos de productores y el uso de las tierras.
- Realizar una simulación exploratoria prospectiva utilizando el enfoque multiagentes, con CORMAS como plataforma de simulación.

La metodología utilizada para el desarrollo de dicho proyecto consistía en tres etapas. En la primera de ellas, se formó un grupo interdisciplinario encargado de entender la realidad. En la segunda etapa se generó el modelo utilizando UML. La utilización de éste lenguaje estándar ayudó en el entendimiento y análisis de la realidad, así como también la comunicación entre los participantes. Finalmente se implementó la simulación en computadora y se ejecutaron varias corridas con diferentes tipos de entrada.

4.2. Estructura estática

En la figura 4.1 vemos el modelo conceptual de la realidad descrita en UML. A la derecha del diagrama se representan las tierras disponibles, o parcelas (Plot), y la utilización de las mismas (LandUse, Cattle, SoyBean y Empty). Cada parcela tiene una única utilización por vez. Cada una de las mismas puede ser alquilada a un empresario (InvestmentFundManager o IFM) o vendida a otro productor tradicional (Traditional). Las parcelas pueden ser utilizadas para ganado (Cattle)

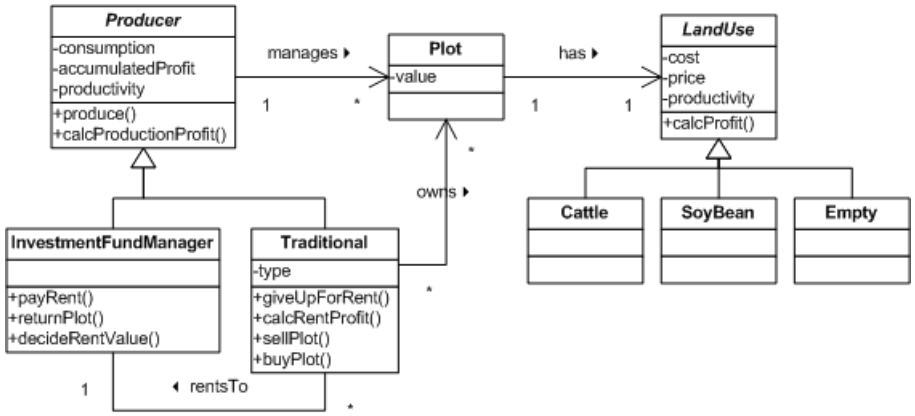

Figura 4.1: Realidad en UML

o soja (Soybean), o pueden estar sin utilizar (Empty). Cada cobertura tiene diferentes costos y precios que evolucionan de acuerdo al mercado.

En la parte izquierda se muestran los agentes del modelo: el productor tradicional (Traditional) y el empresarial (InvestmentFundManager). Cualquier tipo de productor puede administrar un conjunto parcelas, pero sólo los tradicionales pueden ser dueños. Según la cantidad de parcelas de las cuales es propietario un productor tradicional, se pueden clasificar en pequeños, medianos y grandes productores.

Los productores empresariales únicamente siembran soja sobre las parcelas alquiladas. Los tradicionales pueden utilizar sus parcelas para sembrar soja, engorde de ganado, alquilarlas al IFM o dejarlas sin utilizar. La decisión de la actividad a realizar depende de cual es la más rentable para los productores tradicionales.

Podemos notar que en la figura 4.1 no se utiliza una nomenclatura especial para describir los elementos del dominio. Por ejemplo, no sabemos si 'Producer' es un agente, un recurso o parte del ambiente. Si bien un humano podría inferir que se trata de un agente, una herramienta de software no lo puede hacer ya que no cuenta con información apropiada.

Con el lenguaje propuesto, el modelo se puede dibujar como en la figura 4.2.

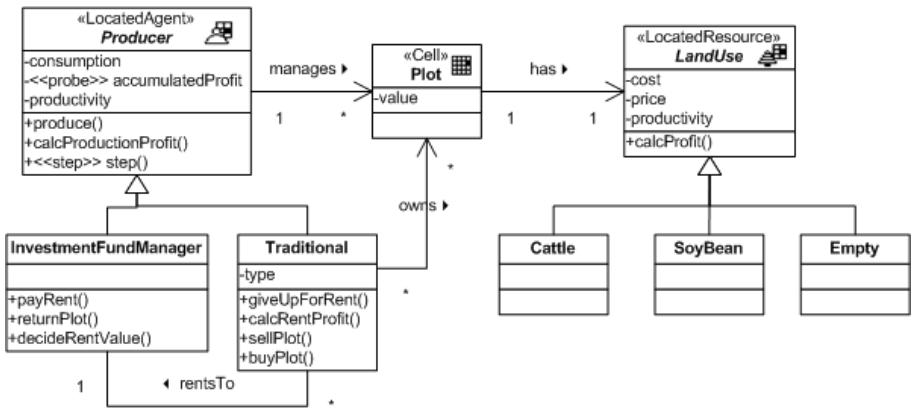

Figura 4.2: Realidad en DSL4MASS

Aquí se ve de forma explícita la naturaleza de cada elemento de nuestra realidad. En este caso, una herramienta de software cuenta con la información adecuada para saber que 'Producer' es un

agente localizado.

Además de la naturaleza de los elementos, se incluye información específica de simulación. Se puede ver claramente qué método corresponde a un 'step' de simulación y las variables que se pretenden monitorear ('probes').

4.3. Estructura dinámica

Además del modelo conceptual especificando la componente estática del modelo, se cuenta con un diagrama de actividad que describe el comportamiento del productor tradicional (versión simplificada) en cada paso de simulación (un año). Este diagrama se puede ver en la figura 4.3.

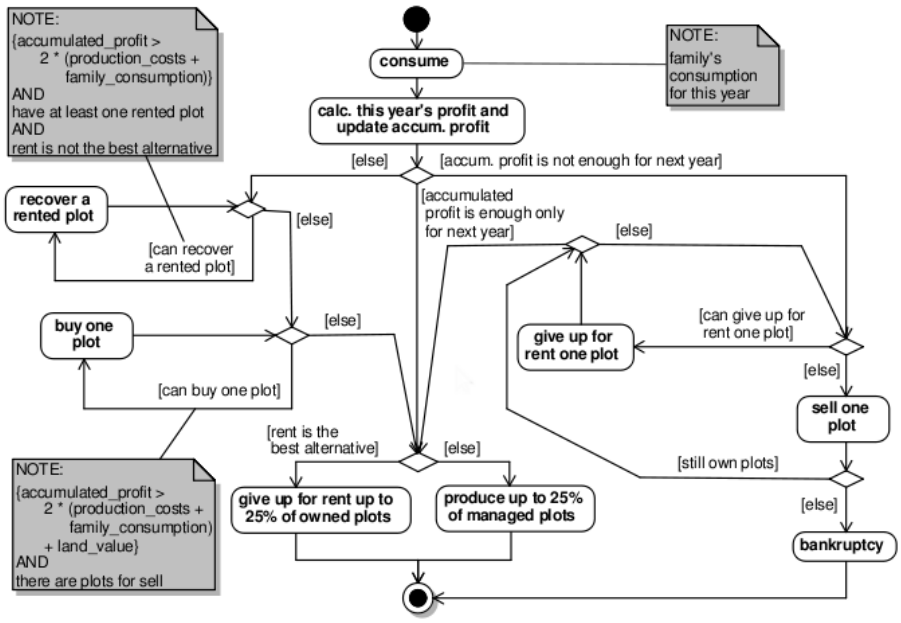

Figura 4.3: Modelo dinámico en UML

El diagrama se puede dividir en tres secciones: izquierda, centro y derecha. La izquierda corresponde a la situación en la que el productor tradicional tiene buenas ganancias acumuladas. En este caso intentará recuperar las parcelas que dio en alquiler y comprar más. El caso contrario se puede apreciar del lado derecho donde la ganancia acumulada no es suficiente para sobrevivir el año siguiente. El productor dará en alquiler y/o venderá sus parcelas hasta obtener la ganancia suficiente para sobrevivir. De no lograrlo se declarará en bancarrota. El centro corresponde a la situación en la cual el productor solo tiene suficiente dinero para el año siguiente. Finalmente, para todos los casos excepto que el productor quede en bancarrota, decidirá o la producción para el año siguiente o alquilar parte de sus parcelas.

El mismo diagrama representado en el lenguaje propuesto se presenta en la figura 4.4.

Si bien el diagrama de actividad propuesto no incluye primitivas específicas del dominio, se agregó una construcción para los ciclos (existentes en los diagramas de secuencia), otra para invocación de métodos y se restringió el elemento de decisión. Todos para facilitar el entendimiento y la generación de código.

En el diagrama original, las decisiones podían tener múltiples condiciones de salida no necesariamente disjuntas. Restringimos las decisiones a una única condición. Con esto evitamos aumentar la complejidad del dibujo para especificar el orden en el que se deben evaluar las condiciones.

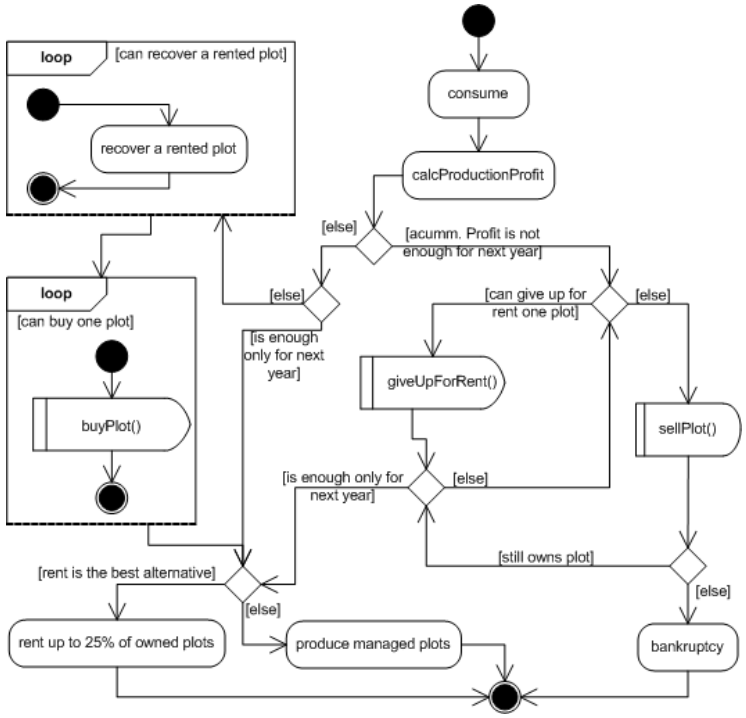

Figura 4.4: Modelo dinámico en DSL4MASS

La construcción de ciclos nos permiten identificarlos claramente en el dibujo y ayuda al generador de código construir estructuras más adecuadas para ellos. Un ciclo puede verse como un subdiagrama de actividad. Ninguna actividad dentro del ciclo puede ir hacia una actividad fuera de él, como tampoco una actividad externa ir hacia una interna.

Las invocaciones de métodos son útiles cuando se quiere mostrar la invocación de un método sobre el propio agente o uno asociado, permitiendo al generador crear el fragmento de código necesario. En nuestro diagrama se pueden ver las invocaciones a los métodos 'sellPlot', 'buyPlot' y 'giveUpForRent'.

4.4. Implementación

En el trabajo original se utilizaron los diagramas UML como entrada a la etapa de implementación de la simulación en CORMAS. Estas entradas son a nivel conceptual. No se utilizó una herramienta que tome los modelos y asista en la implementación final. El programador debió escribir el cien por ciento del código de la simulación.

Para el lenguaje propuesto, las figuras 4.2 y 4.4 tienen su correspondiente representación textual, la cual es utilizada por la herramienta desarrollada para generar un esqueleto de código de la simulación, disminuyendo la cantidad de líneas de código que debe escribir un programador.

No hemos desarrollamos ninguna herramienta gráfica para dar soporte a los diagramas. Por tal motivo, la traducción al lenguaje tipo texto se realizó de forma manual. Sin embargo, existen herramientas, como [32] entre otras, que pueden ser extendidas para lograr esta funcionalidad.

En el siguiente fragmento de código mostramos la especificación de 'Producer' y 'IFM' de la figura 4.2 traducida al lenguaje textual.

Extracto 3 Fragmento de estructura estática en DSL4MASS textual

```
...
Located Agent Producer {
  association manages toMany Plot managedPlots;
  instance var int consumption;
  instance var int accumulatedProfit isProbe;
  instance var int productivity;
  instance method void step() isStep { }
  instance method void produce() { }
  instance method void calcProductionProfit() { }
}
...
InvestmentFundManager extends Agent Producer {
  instance method void payRent() { }
  instance method void returnPlot() { }
  instance method void decideRentValue() { }
}
...
```

Como se puede apreciar, esta representación además de ser simple, es específica de dominio ya que se explicitan los diferentes agentes, recursos, y el ambiente. Por ejemplo, para definir el agente localizado 'Producer' utilizamos la primitiva 'Located Agent' seguida del nombre. También se ve que la variable de instancia 'accumulatedProfit' es un valor que nos interesa muestrear en el transcurso de la simulación ('isProbe') y que el método de instancia 'step' es el método a ejecutar en cada paso de simulación ('isStep').

De manera similar se generó la representación textual de la figura 4.4. A continuación se presenta un fragmento mostrando el ciclo de compra de parcelas y la condición luego de esta.

Extracto 4 Fragmento de estructura dinámica en DSL4MASS textual

```
ActivityFlow for method step of Entity Traditional {
...
  loop buyPlotLoop destination isRentBestAlternativeCond
    with (bool) canBuyPlot() {
      start destination buyPlotCall;
      call buyPlotCall destination END
        target buyPlot();
  };
  condition isRentBestAlternativeCond
    trueDestination rentOwnedPlots
    falseDestination produceManagedPlots
    with (bool) isRentBestAlternative();
...
}
```

En ambos diagramas la traducción es directa, ya que todas las primitivas ofrecidas en el lenguaje gráfico tienen su correspondiente en el lenguaje textual.

La herramienta construida toma la representación textual y genera una estructura de código para CORMAS. Todo el código se debe generar en un archivo XML, utilizado por Cincom Smalltalk ([35]). El siguiente fragmento muestra parte del XML generado, mostrando la implementación del ciclo de compra de parcelas.

Extracto 5 Fragmento de código Smalltalk para Cincom Visual Works

```
...
<class-id>CormasNS.Models.Parcelaria.Traditional</class-id>
<category>step_activity</category>
...
<body package="Parcelaria" selector="buyPlotLoop">buyPlotLoop
""
[self canBuyPlot] whileTrue: [
        self buyPlotLoopLoopInit
].
self isRentBestAlternativeCond</body>
</methods>
...
```

En la figura 4.5 se puede ver el mismo ciclo en Cincom Visual Works.

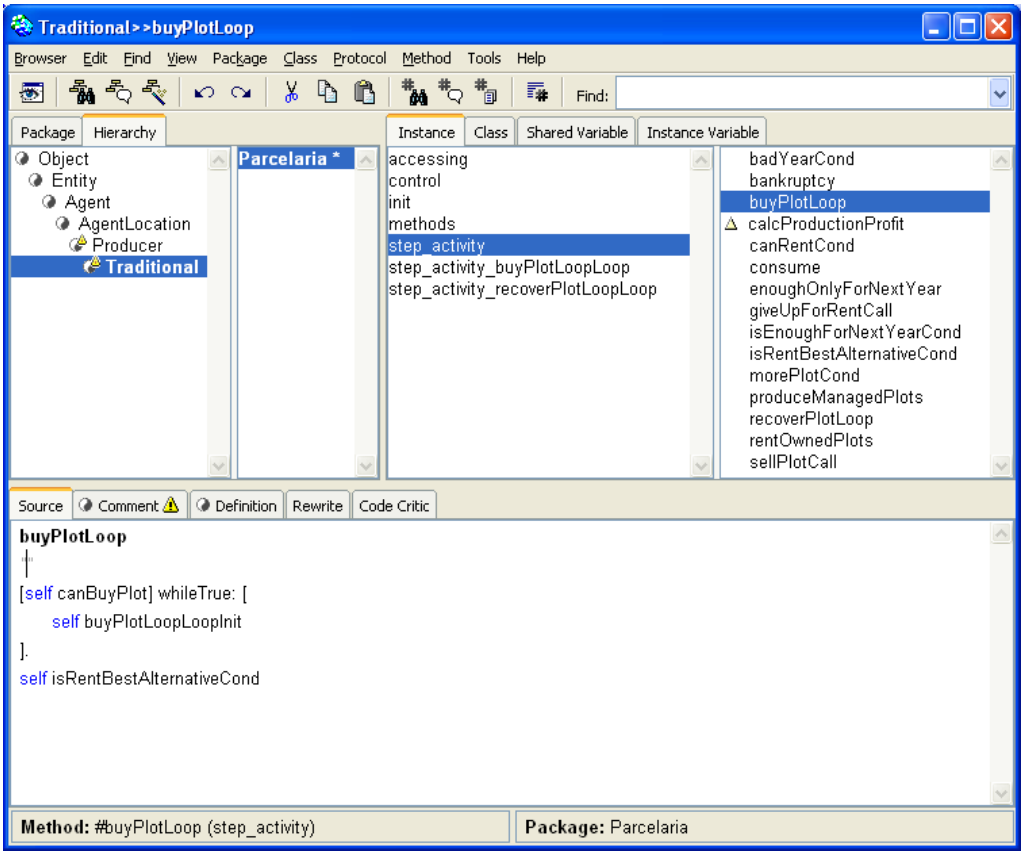

Figura 4.5: Imagen de código generado en CORMAS

Un programador deberá completar la estructura generada, realizando un esfuerzo menor que la implementación completa de la simulación. El trabajo de completar varía según el detalle con que se realizan los diagramas de actividad.

4.5. Conclusiones

Como conclusión de la implementación del caso de estudio, se desprende que la utilización del lenguaje propuesto ofrece mayor expresividad a la hora de modelar un sistema para su simulación basada en multiagentes.

La herramienta construida acelera el proceso de desarrollo y disminuye la probabilidad de errores de implementación, ya que se puede generar un esqueleto del código final a partir de los diagramas. En la metodología tradicional, esto lo debe realizar el programador.

Un punto para medir el esfuerzo del programador se desprende de las líneas de código escritas. La simulación final requirió 800 líneas de código[1], de las cuales herramienta generó 700 aproximadamente. Es decir, el programador escribió solamente el 12,5 % de las líneas.

[1] Consideramos líneas de código, la cantidad de líneas en el archivo XML que utiliza Cincom Smalltalk como código de programa.

Capítulo 5

Conclusiones

Actualmente existen líneas de investigación en el campo de agroecosistemas que utilizan simulaciones en computadora para realizar análisis de los posibles desenlaces de un sistema al introducir cambios en el ambiente (como fluctuaciones de precios, sequías, etc.).

Generalmente, el modelado de la realidad (con el objetivo de ser simulado) es llevado a cabo por un grupo interdisciplinario, donde intervienen expertos del dominio (como agrónomos) y programadores. El enfoque utilizado para esto es el de multiagentes, donde se modela el comportamiento individual de cada una de las partes del sistema (divididos en agentes, recursos y ambiente). UML, por ser un lenguaje gráfico sencillo de entender por no informáticos, es utilizado para representarlo, además de ayudar en la comunicación y cooperación. Permite modelar la componente estática (o estructura) de la realidad, para representar los elementos y sus relaciones, y la componente dinámica, para el comportamiento o razonamiento de un agente de la realidad . Luego de completada la fase de modelado conceptual se implementa la simulación en computadora. En el contexto del proyecto, la implementación es realizada por un programador utilizando la plataforma CORMAS. Ésta es una herramienta para implementar simulaciones utilizando la tecnología multiagentes enfocada en simulaciones de utilización de recursos naturales, lo cual la hace adecuada para agroecosistemas.

El proyecto comenzó con la búsqueda de un lenguaje especifico para esta área y una herramienta que ayude más al programador (por ejemplo generando código a partir de los diagramas UML). Se encontraron varias herramientas y lenguajes interesantes, pero no satisfacían completamente nuestras expectativas. Esto nos llevó a crear nuestro propio lenguaje para simulaciones multiagentes que es adecuado para agroecosistemas (aunque no exclusivo para ellos). Construimos una herramienta que toma modelos representados en dicho lenguaje y genera código Smalltalk para CORMAS. Finalmente pusimos a prueba tanto el lenguaje como la herramienta con el objetivo de verificar si realmente se hizo un aporte. Se tomó una simulación existente con el enfoque tradicional y se desarrolló nuevamente, observando en cada paso de la construcción las diferencias.

Durante la búsqueda hallamos herramientas que soportan todo el proceso, desde modelado hasta simulación. SeSAm, por ejemplo, fue una de las mejores herramientas encontradas. Presenta los conceptos de agentes, recursos y ambiente. Permite realizar toda la especificación en forma gráfica, aunque en el modelo estático no se pueden especificar las relaciones entre los elementos. La especificación de comportamiento por más que sea gráfica, requiere las habilidades de un programador, el cual debe especificar minuciosamente detalles de implementación que serían mucho más cómodos y rápidos de escribir en un lenguaje de programación. Lejos de ayudar, se hace muy pesado el trabajo de implementar el comportamiento. Esta herramienta fue descartada por las carencias en el modelo estático y por ser poco práctico.

Repast mejora este último aspecto, dejando agregar líneas de código en forma de propiedades en el comportamiento, aunque incómodo de escribir. Otro punto importante es que genera código accesible al programador. Sin embargo, éste no conviene editarlo directamente ya que se regenera completamente frente a cualquier cambio en los diagramas. Esta herramienta también fue descar-

tada, principalmente porque no da soporte al concepto de recurso (haciéndolo poco adecuada para agroecosistemas), y luego por su usabilidad.

En cuanto a lenguajes, encontramos perfiles de UML que sirven para modelado de sistemas multiagente. Los de mayor consideración fueron AML y AUML. AUML fue una de las primeras aproximaciones para modelar sistemas multiagentes, propuesto por FIPA (organización encargada de realizar estándares para la tecnología multiagentes). Sin embargo, debido a actualizaciones en otros estándares relacionados, su desarrollo está detenido. AML es un lenguaje más moderno, mejor formalizado y toma referencias de AUML y otros lenguajes. A nuestro criterio es el mejor y más evolucionado de los lenguajes encontrados para MAS. Sin embargo, ninguno cubre conceptos de simulación por lo que no son adecuados para nuestras necesidades.

Dado que no encontramos nada que satisfaga las características deseadas, decidimos construir nuestro propio lenguaje basado en AML (por tener muchas características deseables). El lenguaje obtenido tiene las siguientes características:

- Ayuda en todo el proceso al programador. En las primeras etapas, para la comunicación con los expertos del dominio y como documentación. En etapas posteriores, como entrada para el generador de código, adelantando gran parte del trabajo de programación.

- Mayor expresividad en el dominio de simulaciones basadas en multiagentes. En el diagrama de clases queda explícito quiénes son los agentes, cuáles son los recursos, el ambiente, las relaciones entre ellos y aspectos de simulación como el step y variables a observar. En los diagramas de actividad se agregó la representación de ciclos y llamadas al mismo u otros agentes.

- Adecuado para agroecosistemas. Los conceptos de agentes, recursos y ambiente son importantes en esta área y están soportados de forma explícita por el lenguaje. No todas las herramientas de simulación multiagente lo hacen con recursos y ambiente.

- No exige mucha especificación gráfica. El programador sólo especifica el modelo y rasgos generales del comportamiento. Los detalles de programación se deberán implementar en un lenguaje de programación.

Junto con el lenguaje desarrollamos una herramienta que permite tomar el modelo especificado y traducirlo a lenguaje ejecutable. Sin embargo, el código generado es simplemente un esqueleto de la estructura y el comportamiento. Exige que el programador complete dicho esqueleto. Las características principales son:

- Genera código a partir del modelo. El modelo especificado en nuestro lenguaje, además de servir de guía, es una entrada procesable a la etapa de programación.

- Ahorra tiempo de programación. En la prueba realizada, el esqueleto significaba casi un 90 % del código final de la simulación.

Si bien la herramienta genera código Smalltalk para CORMAS, tanto el lenguaje como la herramienta se diseñaron en base a los conceptos generales y no sobre ésta plataforma. Esto nos agrega la ventaja que el generador puede ser adaptado a otra plataforma de simulación. El lenguaje no se vería afectado y gran parte de la herramienta puede ser reutilizada.

Luego de implementado el caso de estudio, se observó que el lenguaje propuesto junto a la herramienta mejora la metodología tradicional. Esto es debido a que se logra mayor expresividad en los diagramas del modelo y se genera una gran parte del código final a partir de los mismos.

5.1. Trabajo a Futuro

Si bien alcanzamos los objetivos fijados, detectamos oportunidades de mejora que enumeraremos a continuación.

Nuevos diagramas Nuestro DSL actualmente extiende los diagramas de clases y de actividad de UML. Quizá sea de utilidad extender otros diagramas (tanto de AML como de UML) con el objetivo de abarcar más apropiadamente otros aspectos. Por ejemplo, los diagrama de comunicación podrían ser más apropiados que los de actividad para especificar el scheduler de la simulación.

Mejorar construcciones Dentro de las limitantes de la versión actual de DSL4MASS, se encuentra la restricción en las condiciones de los diagramas de actividad. Podría ser de utilidad modificarlos para soportar varias alternativas, definiendo el orden en que deben evaluarse. En los diagramas de estructura, sería conveniente permitir la definición de clases comunes (para representar datos). También se podría permitir especificar más detalladamente las cardinalidades en el lenguaje textual (actualmente sólo se soporta 1 o N). El lenguaje hoy no soporta ni entidades ni métodos abstractos, queda como trabajo a futuro darle soporte a esto.

Formalizar meta-modelo DSL4MASS es una extensión informal de AML y UML. Resta formalizar dicha extensión a través del meta-modelo de UML.

Herramienta gráfica Si bien el lenguaje propuesto es un lenguaje gráfico basado en UML, no construimos una herramienta que permita realizar el modelado gráfico. Hemos trabajado con una representación textual del lenguaje.

Ingeniería inversa Si se conoce la plataforma de simulación, también se puede dar soporte bidireccional al proceso. Es decir, poder realizar ingeniería inversa cuando se modifica el código fuente y actualizar los diagramas.

Motor de simulación CORMAS es una plataforma para ser utilizada con Smalltalk, lenguaje que ha perdido bastante terreno frente a lenguajes modernos como Java o .NET. Un trabajo a futuro es generar un motor de simulación más apropiado para el dominio. Incluso que sea específico para agroecosistemas, lo cual daría lugar también para agregar más elementos de agroecosistemas al lenguaje.

IDE Por último, restaría crear una herramienta del estilo de Repast o SeSAm, que asista en todo el proceso de construcción y ejecución de la simulación dentro de una misma herramienta basado en un lenguaje moderno, como por ejemplo Java. Esto requeriría construir (o integrar) un componente que permita realizar los diagramas utilizando el lenguaje propuesto y generando la representación textual. Se debería construir una plataforma de simulación basada en Java para soportar todos los aspectos de una simulación multiagente. Habría que modificar el generador de código construido para que su salida sea con el formato y construcciones de la nueva plataforma. Construir (o integrar) un editor de código Java adecuado que permita al programador completar la estructura creada por el generador de código. Para la última fase se requiere un módulo que permita ejecutar la simulación de manera gráfica, soportando distintos tipos de salida (similar a casi todas las herramientas gráficas de simulación).

Bibliografía

[1] BOTBOX. Personal Internet Agents. http://www.botbox.com/products.php.

[2] P. Anthony and N. R. Jennings. Developing a bidding agent for multiple heterogeneous auctions. *ACM Trans. Internet Tech*, 3(3):185–217, 2003.

[3] Repast Simphony. Agent based modeling toolkit. http://repast.sourceforge.net/.

[4] Breve. The breve Simulation Environment. http://www.spiderland.org/.

[5] Giovanni Caire, Francisco Garijo Jorge, Gomez Juan Pavon, and Emilio Vargas. Agent Oriented Analysis using MESSAGE/UML. pages 119–135. Springer-Verlag, 2001.

[6] Jorge Corral, Pedro Arbeletche, Julio César Burges, Hermes Morales, Guadalupe Continanza, Jorge Couderc, Virginia Courdin, and Pierre Bommel. Multi-agent systems applied to land use and social changes in río de la plata basin (south america).

[7] Viviane Torres da Silva, Ricardo Choren, and Carlos J. P. de Lucena. A uml based approach for modeling and implementing multi-agent system. Technical report, Pontifícia Universidade Católica do Rio de Janeiro.

[8] ANTLR. Generador de compiladores. http://www.antlr.org/.

[9] MIMOSA. Plataforma de modelado y simulación. http://sourceforge.net/projects/mimosa.

[10] JAVA Standard Edition. http://java.sun.com/javase/.

[11] SeSAm. Multi-Agent Simulation Environment. http://www.simsesam.de/.

[12] Jacques Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Professional, February 1999.

[13] Martin Fowler. http://www.martinfowler.com/bliki/DomainSpecificLanguage.html.

[14] Jade. Java Agent Development Framework. http://jade.tilab.com/.

[15] William Harrison and Robert Harrison. Domain Specific Languages for Cellular Interactions. pages 3019–3022. IEEE, 2004.

[16] Michael N. Huhns. Agent uml notation for multiagent system design. 2004.

[17] Eclipse IDE. http://www.eclipse.org/.

[18] Mauro Jansen and Rosario Girardi. *GENMADEM: A Methodology for Generative Multi-agent Domain Engineering*, volume 4039 of *Lecture Notes in Computer Science*, pages 399–402. Springer Berlin / Heidelberg, 2006.

[19] Marco A. Janssen. Agent-based modelling. Technical report, Indiana University, 2005.

[20] Nicholas R. Jennings and Michael Woolridge. Pitfalls of agent-oriented development. *Proceedings 2nd International Conference on Autonomous Agents*, pages 385–391, 1998.

[21] E. A. Kendall, C. V. Pathak, P. M. Krishna, and C. Suresh. The layered agent pattern language. *Proceedings of the Conference on Pattern Languages of Programs*, 1997.

[22] Franziska Klüg. Multi-agent simulation. Technical report, Universität Würzburg, 2004.

[23] HP Labs. http://www.hpl.hp.com/techreports/.

[24] Agent UML. Multi-Agent Modeling Language. http://www.auml.org/.

[25] MAML. Multi-Agent Modeling Language. http://www.maml.hu/maml/initiative/index.html.

[26] Dr. Jürgen Lind. General concepts of agents and multiagent systems. http://www.agentlab.de/.

[27] Dr. Jürgen Lind. *The MASSIVE Method*, volume 1994 of *Lecture Notes in Computer Science*. Springer, 2001.

[28] INGENIAS Development Kit. Desarrollo MAS. http://ingenias.sourceforge.net/.

[29] NetLogo. Ambiente multiplataforma programable para modelado multiagente. http://ccl.northwestern.edu/netlogo/.

[30] Mohammad S Obaidat and Georgios I Papadimitriou. *Applied System Simulation: Methodologies and Applications*. 2003.

[31] Swarm. Plataforma para modelado basado en agentes. http://www.swarm.org/.

[32] StarUML. The Open Source UML/MDA Platform. http://staruml.sourceforge.net.

[33] Cormas. Natural Resources and Multi-Agent Simulations. http://cormas.cirad.fr/.

[34] MASON. Multiagent simulation library. http://www.cs.gmu.edu/~eclab/projects/mason/.

[35] Cincom Smalltalk. http://www.cincomsmalltalk.com/.

[36] FIPA. Multi-Agent IEEE Computer Society standards organization. http://www.fipa.org/.

[37] Whitestein Technologies. http://www.whitestein.com/.

[38] Whitestein Technologies. LS/ATN - Living Systems® Adaptive Transportation Networks. http://www.whitestein.com/autonomic-business-solutions/ls-atn-living-systems-adaptive-transportation-networks.

[39] Ivan Trencansky and Radovan Cervenka. *AML - The Agent Modelling Language: A comprehensive approach to modelling Multi-Agent Systems*. 2007.

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

[41] Michael Wooldridge. *Introduction to MultiAgent Systems*. John Wiley & Sons, June 2002.

[42] Franco Zambonelli, Nicholas R. Jennings, and Michael Woolridge. Developing multiagent systems: The gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, July 2003.

Glosario

Agroecosistemas Ecosistema sometido por el hombre a continuas modificaciones de sus componentes bióticos y abióticos.

AML Agent Modeling Language.

API Interfáz de programación de aplicaciones (Application Programming Interface). Conjunto de funciones y procedimientos que ofrece un sistema para ser utilizado por otro software.

AUML Agent Modeling Language.

CORMAS Plataforma para la simulación multiagente enfocado a la utilización de recursos naturales.

DSL Lenguaje específico de dominio (Domain Specific Languaje).

DSL4MASS Lenguage propuesto. DSL para MASS.

FIPA Organización de estándares de la IEEE que promueve la tecnología multiagentes.

Framework Componente reutilizable por otros sistemas de software.

IDE Ambiente integrado de desarrollo.

Java Lenguaje de programación orientado a objetos.

MAS Sistema Multiagente (Multi-Agent System).

MASS Simulación multiagente (Multi-Agent System Simulation).

Meta-modelo Define lenguaje y procesos con los cuales construir un modelo.

Plataforma Componente reutilizable por otros sistemas de software.

Probe Variable monitoreada dentro de una simulación.

Repast Herramienta para el modelado, implementación, y ejecución de simulaciones basadas en multiagentes.

Scheduler Determina el orden en que los diferentes agentes ejecutan su step en una simulación time-step.

SeSAm Ambiente genérico para el modelado y experimentación con simulaciones basadas en agentes.

Smalltalk Lenguaje de programación orientado a objetos.

Step En simulación de tipo time-step, un step es un turno de una entidad. Es el evento que se ejecuta cada cierta cantidad fija de tiempo.

Time-step Tipo de simulación donde el modelo se ejecuta cada cierta cantidad fija de tiempo.

UML Unified Modeling Language.

XML XML (eXtensible Markup Language) es un metalenguaje extensible basado en texto desarrollado por el World Wide Web Consortium (W3C).

Apéndice A

Diseño de la Herramienta

En este anexo explicaremos en detalle dos aspectos importantes de la herramienta: cómo se utiliza y cómo puede ser extendida por un programador.

A.1. Guía de Usuario

La herramienta construida es una aplicación de para línea de comandos implementada en Java SE 5 [10], por lo que se necesita al menos Java Runtime Enviroment 5 instalado para su ejecución. Además de Java estándar, se utilizó una librería externa llamada ANTLR [8] en su versión 3.1.1, debiendo estar ésta accesible al momento de la ejecución.

A la herramienta se le debe proveer un modelo especificado en DSL4MASS textual. Debe existir un archivo con extensión `structure` conteniendo la estructura estática, y opcionalmente uno o varios archivos con extensión `activity` con los diagramas de actividad, también en formato textual. Todos estos archivos deben estar contenidos en un mismo directorio. La ruta de éste se debe pasar por parámetro al ejecutar.

Un ejemplo de entrada se puede ver en la figura A.1, donde el archivo `basicgrammar.strucure` contiene la estructura estática del modelo y los archivos `model_scheduler.activity` y `productor_step.activity` contiene la especificación de comportamiento. La invocación en este caso, suponiendo que se encuentra en la carpeta anterior a `examples`, se vería de la siguiente forma:

```
java uy.edu.fing.dsl4mass.generator.DSL4MASSMain ./examples/firstmodel
```

Figura A.1: Ejemplo de entrada

La salida de la herramienta es dependiente de la plataforma para la cual se está generando. La salida implementada para CORMAS es un único archivo de extensión `pst` y de nombre igual al nombre del modelo (si fue especificado). El archivo se ubica dentro de una carpeta con el mismo nombre del archivo, dentro de una carpeta `Models` bajo el directorio de ejecución. Para el ejemplo anterior, el modelo `BasicGrammar`, la salida se vería como en la figura A.2.

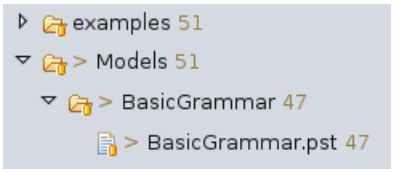

Figura A.2: Ejemplo de salida

Para importar la salida en CORMAS, la carpeta generada dentro de `Models`, se debe copiar dentro de la carpeta `Models` de CORMAS.

A.2. Guía del Programador

La herramienta fue implementada en Java 5 [10]. La única librería utilizada fue ANTLR 3.1.1 [8] para construir el traductor a partir de la gramática del lenguaje textual.

Para la generación del código final a partir de la especificación, se pasa por diferentes fases, transformando en cada paso la entrada de la etapa anterior. Se podría resumir el proceso en las siguientes fases:

1. Interpretación de los diagramas. Se genera una estructura básica del modelo de entrada, haciendo sólo validaciones sintácticas. En esta fase es donde se utiliza ANTLR.

2. Representación interna. Se genera una nueva estructura intermedia más rica que la anterior, con la cual se realiza una serie de validaciones independientes de la plataforma de salida.

3. Generación de código. Se genera el esqueleto de código para una plataforma de simulación específica. Dicha generación puede incluir un proceso de validación extra según la plataforma. En nuestro proyecto, esta fase es la encargada de generar comportamiento por defecto para las entidades en caso de no estar especificada en la entrada. Esto se debe a que CORMAS exige que todos los agentes tengan método de inicialización y de step.

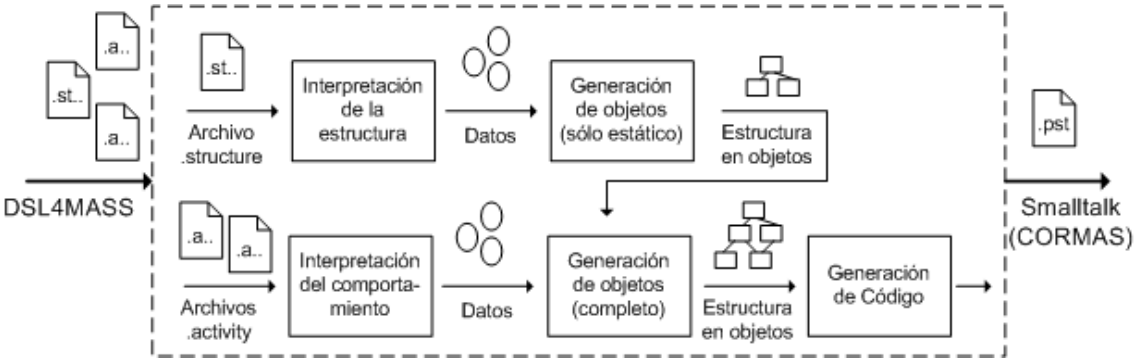

Figura A.3: Fases de procesamiento

En realidad las fases 1 y 2 se repiten varias veces, como se muestra en la figura A.3. La primera vez se ejecuta para el diagrama con la estructura estática del modelo y luego para cada archivo de actividad, enriqueciendo en cada iteración la representación interna (en objetos) del modelo.

Finalmente, el proceso de generación de código toma toda la información para producir el código Smalltalk para CORMAS. Es en esta fase donde se puede implementar la salida a otra plataforma de simulación, sin modificar el código de las etapas anteriores.

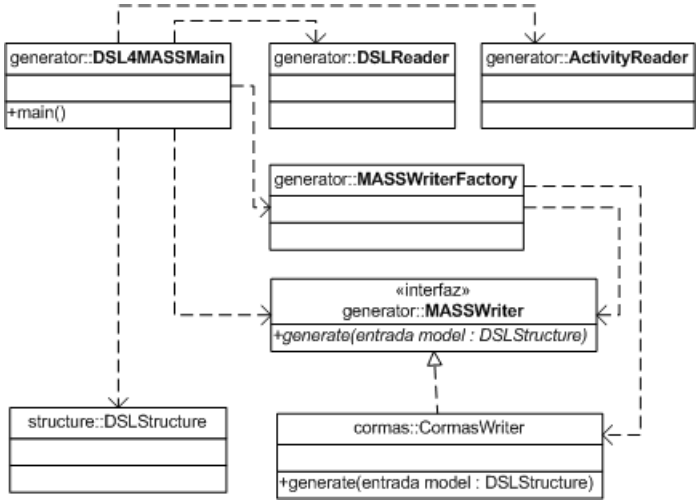

Figura A.4: Diseño de la Herramienta

En la figura A.4 se puede observar las clases que participan en las fases mencionadas. DSL4-MASSMain es el punto de entrada de la herramienta, quién implementa el método invocado al iniciarla. Utiliza las clases DSLReader y ActivityReader para interpretar la estructura y el comportamiento respectivamente y generar los objetos de negocio a partir de archivos con diagramas de clase y de actividad. Esta información es almacenada en un objeto de la clase DSLStructure, que es utilizada por el MASSWriter apropiado para generar el código final. La clase MASSWriterFactory es la encargada de conocer qué implementación de la interfaz MASSWriter se utilizará para generar el código. Como construimos un generador de código Smalltalk para CORMAS, implementamos la interfaz en la clase CormasWriter. En caso de querer modificar la herramienta para generar código de otra plataforma de simulación, se debe implementar la interfaz MASSWriter, además de modificar la clase MASSWriterFactory para que cree una instancia de la nueva clase.

El generador implementado, CormasWriter, agrega ciertas particularidades a la salida; parte por la plataforma y parte por el lenguaje Smalltalk. En CORMAS, los métodos step sólo pueden llamarse 'step' (porque sobrecargan un método de una superclase). Como DSL4MASS permite cualquier nombre, al generar el código de un método marcado como step se renombra dicho método al nombre requerido. Además es obligatoria la existencia del step, por lo que se genera un esqueleto vacío en caso que un agente no lo tenga definido. Un caso análogo se presenta con el método scheduler del modelo.

Smalltalk requiere que en la invocación a métodos con dos o más parámetros, los valores pasados incluyan el nombre del parámetro en la llamada. Si se debe generar una llamada de éste tipo y el método no se encuentra especificado en el modelo (porque es un método nativo o no interesaba modelarlo), el generador desplegará una advertencia, no genera la invocación y continúa con el resto.

En la figura A.5 se presenta un diagrama de clases de diseño con la representación interna de la componente estática de la realidad. Tanto las entidades de MAS (Agentes, Recursos y Ambiente) como el modelo (DSLStructure), tienen asociados definiciones de atributos y métodos. Los métodos por su parte, pueden o no tener asociado una especificación de comportamiento (MethodSpec), la que se obtiene luego de interpretar la estructura dinámica. Las entidades MAS son las únicas que pueden tener asociaciones con otras entidades.

En la figura A.6 se muestra la representación interna de la estructura dinámica. Actualmente sólo se implementa los diagramas de actividad, por lo que MethodSpec sólo tiene una extensión, ActivityMethodSpec. ActivityFlow es quién representa todo un diagrama de actividad. Todas las

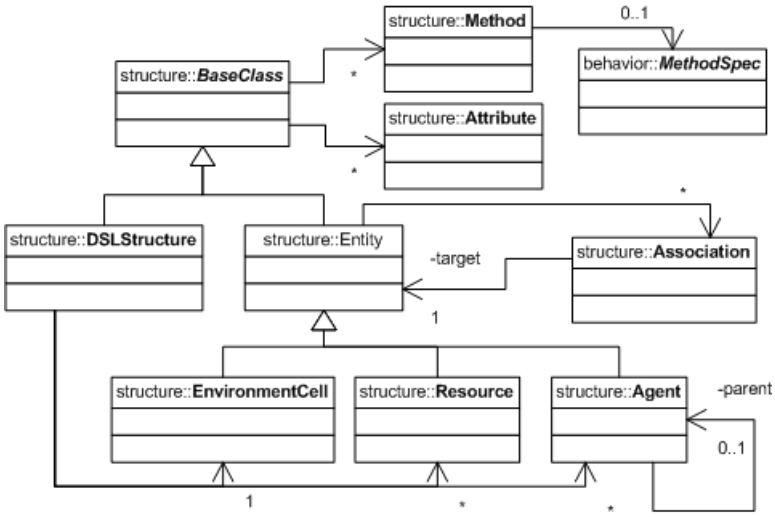

Figura A.5: Representación interna para estructura estática

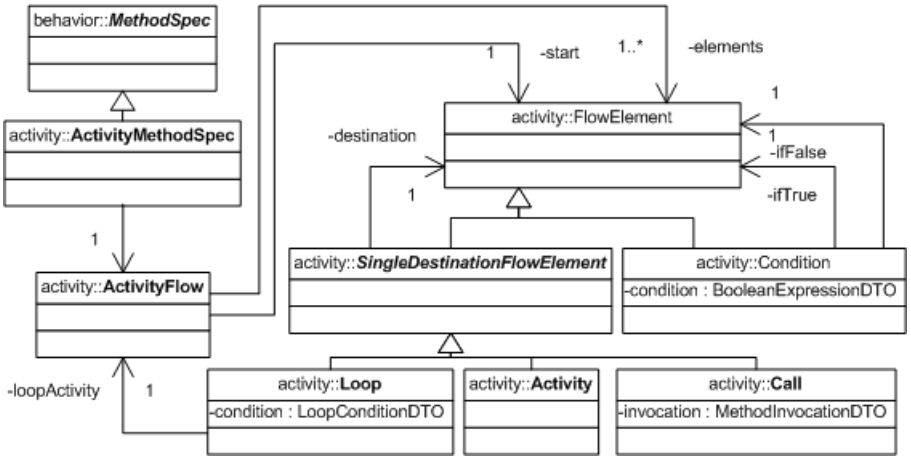

Figura A.6: Representación interna para estructura dinámica

construcciones permitidas heredan de FlowElement, que representa un elemento de la actividad donde pueden ser destino otro elemento. Los ciclos (Loop), a demás de tener un elemento destino, tienen un diagrama de actividad interno (lo que se repite). Aquí existe la restricción de que los elementos de un ciclo no pueden acceder a elementos fuera de él, así como tampoco elementos externos pueden acceder a elementos dentro de él. Si se permite la anidación de ciclos.

El diagrama de actividad debe generarse en código como una máquina de estados. La alternativa utilizada para el generador de CORMAS es hacer corresponder cada elemento del flujo con un método. Luego, para codificar la transición simplemente se genera al final del método una invocación al método correspondiente al elemento destino. En caso de que el destino sea el estado de fin, no se genera ninguna llamada. Esto provoca que la actividad se ejecute recursivamente.

Se intentó escribir un código claro y con comentarios, por lo que cualquier extensión o modificación de los componentes implementados, así como el estudio del código fuente no debería ser una tarea difícil de realizar si se tiene en cuenta el diseño presentado aquí.

Apéndice B

Lista extendida de Herramientas

Aquí se muestran en detalle todas las herramientas evaluadas en el transcurso del estado del arte.

CORMAS

CORMAS [33] está enfocado hacia los modelos de administración de recursos naturales renovables, orientado a la representación de las interacciones entre los interesados en el uso de los recursos.

Es una plataforma de simulación multiagente integrada al ambiente de programación Cincom VisualWorks ([35]). Para la implementación de las simulaciones utiliza el lenguaje Smalltalk, el cual es orientado a objetos y tipado dinámico. La plataforma ofrece un conjunto de clases base para dar el soporte al paradigma de multiagentes (agentes, recursos, ambiente).

Provee un ambiente gráfico para correr las simulaciones. Permite, entre otras cosas, asignar variables iniciales y configurar la forma de desplegar los valores observados.

Herramienta

Tipo: Simulación multiagente para Recursos Naturales.

Extensibilidad: Modificando código fuente (Smalltalk).

Licencia: Open Source.

Documentación: Escasa.

Modelo Multiagentes

Modelado: Código Smalltalk.

Soporte gráfico: No.

Representación: XML (código Cincom Smalltalk).

Simulación

Tiempo: Timestep.

Construcción: Código Smalltalk.

Soporte gráfico: Grilla 2D.

Salida: Gráficas personalizables, se puede exportar a CSV[1].

Soporte: Repetible (asignación de variables iniciales y semilla).

Mimosa

Mimosa [9] es una plataforma para el modelado y simulación. Su objetivo es cubrir todo el proceso desde construir los modelos conceptuales hasta ejecutar la simulación.

La especificación del modelo se realiza a través de ontologías y un conjunto extensible de formalismos, como por ejemplo código Java, que permiten especificar el comportamiento de las entidades. El motor de la simulación está basado en DEVS[2].

Su desarrollo esta siendo llevado a cabo por el mismo equipo de desarrollo de CORMAS. Mimosa se plantea como el sucesor del mismo.

A la hora de evaluar esta herramienta, se encontraba en una versión beta con muchos fallos y funcionalidades sin implementar.

Herramienta

Tipo: Simulación.

Extensibilidad: Tipo Framework.

Licencia: Open Source.

Documentación: Buena.

Modelo Multiagentes

Modelado: Ontologías.

Soporte gráfico: Si.

Representación: XML.

Simulación

Tiempo: Timestep.

Construcción: Gráfico para la estructura. Para el comportamiento se puede utilizar: DEVS, código Java, etc.

Soporte gráfico: Grilla 2D.

Salida: Gráficas personalizable, otros programables por el desarrollador.

Soporte: Repetible (asignación de variables iniciales y semilla).

[1] Tipo de documento sencillo para representar datos en forma de tabla, donde las columnas se separan por comas.

[2] DEVS es un formalismo para modelar y analizar sistemas de diversos tipos, en particular, sistemas de eventos discretos.

Repast

Repast [3] es una herramienta libre y de código abierto para el modelado, implementación, y ejecución de simulaciones basadas en multiagentes. Da soporte a simulaciones basadas en eventos discretos en ambientes 2D y 3D.

Presenta un entorno gráfico para la especificación del modelo y el comportamiento de los agentes integrado a Eclipse IDE [17]. No utiliza un lenguaje gráfico estándar.

El comportamiento se define en base a varias construcciones como ser iteraciones, decisiones y tareas, entre otras. Para las tareas, como también para otros elementos, se debe especificar el código a ejecutar. Éste se escribe en una tabla de propiedades (una línea de código por propiedad), haciendo que la especificación de cada tarea (escribir código) se vuelva incomoda.

El código de la simulación es generado automáticamente a partir de los modelos. A pesar de estar disponible, no conviene modificarlo ya que es regenerado ante cualquier cambio en los diagramas.

Herramienta

Tipo: Simulación multiagente.

Extensibilidad: Tipo Framework.

Licencia: Open Source.

Documentación: Buena.

Modelo Multiagentes

Modelado: Gráfico.

Soporte gráfico: Si. También se puede incluir código Groovy[3].

Representación: XML.

Simulación

Tiempo: Eventos discretos.

Construcción: Gráfico y código Groovy.

Soporte gráfico: Grilla 2D y 3D.

Salida: Gráficas personalizables, CSV.

Soporte: Repetible (asignación de variables iniciales y semilla), Almacenable (en disco, se puede detener y continuar luego).

Jade

Jade [14] es una plataforma que simplifica el desarrollo de sistemas multiagente. Cumple con las especificaciones FIPA y ofrece herramientas gráficas para dar soporte a las fases de desarrollo y depuración.

Pudimos apreciar que es un proyecto activo, ya que en el transcurso de este proyecto se liberaron nuevas versiones.

Parece ser uno de las mejores plataformas disponibles para el desarrollo de sistemas multiagente. No fue pensado para implementar simulaciones.

[3]Groovy es un lenguaje de programación orientado a objetos implementado sobre la plataforma Java.

Herramienta

Tipo: Desarrollo de MAS.

Extensibilidad: Extendiendo API[4].

Licencia: Open Source.

Documentación: Buena.

Modelo Multiagentes

Modelado: Código Java.

Soporte gráfico: No.

Representación: Código fuente.

SeSAm

SeSAm [11] provee un ambiente genérico para el modelado y experimentación con simulación basada en agentes. Provee un ambiente gráfico donde se define el modelo y el comportamiento de las entidades.

La implementación de la simulación es puramente gráfica (no se escribe código), esto hace que el desarrollo sea engorroso, ya que se debe ser muy minucioso en la especificación gráfica. Para nuestro gusto, los detalles de implementación se especifican más fácilmente escribiendo código.

Esta es una de las mejores herramientas de simulación basada en multiagentes que evaluamos.

Herramienta

Tipo: Simulación multiagente.

Extensibilidad: Modificando código fuente.

Licencia: Open Source.

Documentación: Buena.

Modelo Multiagentes

Modelado: Gráfico.

Soporte gráfico: Si.

Representación: XML.

Simulación

Tiempo: Timestep.

Construcción: Especificación gráfica mediante diagramas.

Soporte gráfico: Grilla 2D.

Salida: Archivos CSV, gráficas.

Soporte: Repetible, mismos valores iniciales.

[4]Conjunto de funciones y procedimientos que ofrece cierta biblioteca o sistema para ser utilizado por otro software como una capa de abstracción.

Mason

Mason [34] es una biblioteca escrita en Java para la simulación multiagente basada en eventos discretos. Provee un conjunto de herramientas para visualización 2D y 3D.

Herramienta

Tipo: Biblioteca.

Extensibilidad: Tipo Framework.

Licencia: Open Source.

Documentación: Buena.

Modelo Multiagentes

Modelado: Código Java.

Soporte gráfico: No.

Representación: Código Java.

Simulación

Tiempo: Eventos Discretos.

Construcción: Código Java.

Soporte gráfico: Grilla 2D, 3D

Salida: Imágenes, archivo de datos, películas Quicktime y más.

Soporte: Repetible.

Breve

Brave [4] es un ambiente de simulación 3D diseñado para la simulación de sistemas descentralizados y vida artificial. Incluye simulación física y detección de colisiones para poder simular criaturas realistas. Ya viene con varias clases y comportamientos (en forma de plantillas). Se le permite al usuario interactuar con la simulación.

Herramienta

Tipo: Simulación multiagente.

Extensibilidad: Tipo Framework.

Licencia: Open Source.

Documentación: Buena.

Modelo Multiagentes

Modelado: Código Python[5] o Steve[6].

Soporte gráfico: No.

Representación: Código fuente.

Simulación

Tiempo: Timestep.

Construcción: Código Python o Steve.

Soporte gráfico: 3D.

Salida: Imágenes (PNG[7]) y animaciones (MPEG[8]).

Soporte: Repetible.

Ingenias

Ingenias Development Kit (IDK [28]) es una plataforma que intenta facilitar el desarrollo de MAS soportando el proceso de desarrollo INGENIAS. Es un conjunto de herramientas para la especificación, verificación e implementación de sistemas multiagente.

Las funcionalidades de IDK pueden ser extendidas agregando módulos para generación de código y verificación del modelo. Esto es posible a través de una API basada en un meta-modelo de MAS.

Herramienta

Tipo: Desarrollo MAS.

Extensibilidad: API.

Licencia: Open Source.

Documentación: Buena.

Modelo Multiagentes

Modelado: Código Java.

Soporte gráfico: Si.

Representación: XML. Exporta código Jade.

Swarm

Swarm [31] es una plataforma de modelado basado en agentes que incluye una plataforma para diseñar, describir y ejecutar experimentos sobre modelos. Es una colección de librerías que provee soporte para la programación de simulaciones.

[5]Python es un lenguaje de programación interpretado, orientado a objetos y multiplataforma.

[6]Steve es un lenguaje de programación interpretado, orientado a objetos, especialmente diseñado para implementar simulaciones 3D.

[7]Formato gráfico basado en un algoritmo de compresión sin pérdida.

[8]Estándares de codificación de audio y vídeo.

Herramienta

Tipo: Simulación multiagente.

Extensibilidad: API.

Licencia: Open Source.

Documentación: Buena.

Modelo Multiagentes

Modelado: Código Objective-C[9] o Java.

Soporte gráfico: No.

Representación: Código fuente.

Simulación

Tiempo: Timestep.

Construcción: Código Objective-C o Java.

Soporte gráfico: Programable.

Salida: Programable.

Soporte: Repetible.

MAML

La versión actual de MAML [25] es básicamente una extensión de Objective-C utilizando librerías Swarm. Consiste en varios macros que definen la estructura general de una simulación. Luego se debe completar con código Swarm.

Es un lenguaje para modelado multiagente pensado para no programadores (estudiantes de ciencias sociales) para crear simulaciones multiagente.

No es un lenguaje gráfico (hay que escribir código), su compilador al momento de evaluarlo se encuentra en la versión 0.03 ALPHA (desarrollado entre 1998 y 1999).

Herramienta

Tipo: Simulación multiagente.

Extensibilidad: Modificando código base.

Licencia: Open Source.

Documentación: Escasa y confusa.

Modelo Multiagentes

Modelado: Código MAML, Swarm, Objective-C.

Soporte gráfico: No.

Representación: Código fuente.

[9]Objective-C es un lenguaje de programación orientado a objetos creado como un superconjunto de C. Implementa un modelo de objetos parecido al de Smalltalk.

Simulación

Tiempo: Timestep.

Construcción: Código.

Soporte gráfico: Programable.

Salida: Programable.

Soporte: Repetible.

Netlogo

Netlogo [29] es un ambiente de modelado programable para la simulación de fenómenos sociales y naturales.

Es un proyecto activo en el transcurso del proyecto. Desde la búsqueda de herramientas hasta las últimas revisiones dadas al documento se liberaron varias versiones del producto.

Herramienta

Tipo: Simulación multiagente.

Extensibilidad: API.

Licencia: Freeware.

Documentación: Buena.

Modelo Multiagentes

Modelado: Código LOGO.

Soporte gráfico: No.

Representación: Código fuente.

Simulación

Tiempo: Timestep.

Construcción: Código LOGO.

Soporte gráfico: Programable. API para 2D y 3D.

Salida: Programable, API para texto y animaciones Quicktime[10].

Soporte: Repetible.

[10]QuickTime es un framework multimedia capaz de manipular varios formatos de video digital, sonido, texto, etc.

Apéndice C

Referencia DSL4MASS

En este apéndice brindaremos detalles de las construcciones del lenguaje gráfico y textual. Comenzamos con una descripción en mayor detalle de las construcciones gráficas que se ofrecen y terminamos especificando la gramática de lenguaje textual propuesto.

C.1. DSL4MASS Gráfico

El lenguaje gráfico propuesto es una extensión informal de AML[39], que a su vez es una extensión de UML. Se extendieron tanto los diagramas de clase (para modelar la estructura estática) como los de actividad (para modelar la estructura dinámica), agregando nuevas construcciones y limitando otras existentes.

C.1.1. Estructura Estática

En la figura C.1 se muestran todas las construcciones ofrecidas por DSL4MASS para especificar la estructura estática. Se pueden observar todos los tipos de entidades con las anotaciones e iconos definidos.

Las únicas entidades soportadas son: modelo, ambiente (celda), agentes y recursos; las cuales deben ser concretas. Todas ellas se representan como clases y por lo tanto pueden tener métodos y atributos, tanto de instancia como estáticos. Los tipos primitivos soportados son: void, int, float, char, string, list, set, object y boolean. Las asociaciones pueden tener cardinalidad 1 o *, y opcionalmente tienen un nombre.

Cualquier entidad puede contener atributos o métodos probe, los cuales se anotan con 'probe'. Para el caso de los métodos debe tener valor de retorno y no recibir parámetros.

El modelo, en caso de ser necesaria su especificación, se representa como una clase con el estereotipo 'Model'. Para especificar el método scheduler se utiliza la anotación 'scheduler'. Este método puede tener cualquier nombre, no debe recibir parámetros ni poseer valor de retorno. Solo puede haber una clase con esta anotación por realidad.

Los agentes se pueden especificar tanto como localizados y no localizados con los estereotipos 'LocatedAgent' y 'Agent' respectivamente sobre una clase. El método step de los agentes se especifica con la anotación 'step'. Este método puede tener cualquier nombre, no debe recibir parámetros ni poseer valor de retorno.

Los recursos se pueden especificar tanto como localizados y no localizados con los estereotipos 'LocatedResource' y 'Resource' respectivamente sobre una clase.

El ambiente se divide en una grilla 2D compuesta por celdas, las cuales se especifican con la anotación 'Cell'. Solo puede haber una clase con esta anotación por realidad.

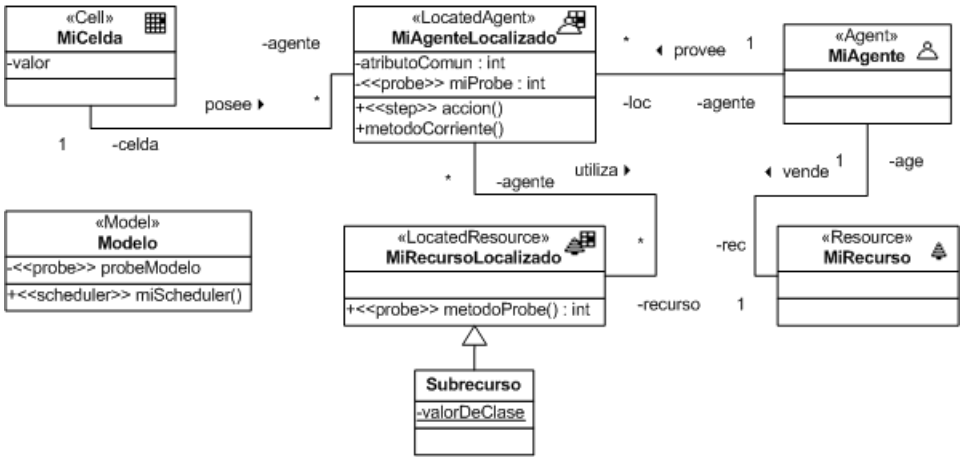

Figura C.1: Construcciones de estructura estática

Tanto los agentes como los recursos pueden ser heredados. La subclase no debe tener ninguna anotación, ya que su naturaleza se deriva de la clase base.

C.1.2. Estructura Dinámica

En la figura C.2 se muestran todas las construcciones ofrecidas por DSL4MASS para especificar la estructura dinámica.

Los diagramas de actividad representan la lógica de un método de una entidad de la realidad. Las únicas construcciones soportadas son: actividades, condiciones, ciclos e invocaciones a métodos.

Una actividad solo tiene un nombre y un destino al cual se debe continuar una vez que esta terminó.

Las invocaciones a métodos representan el envío de mensajes entre los diferentes elementos de la realidad. Tienen un destino al cual continuar luego de su ejecución y un objeto sobre el cual se ejecutan. Este objeto puede ser él mimo o uno perteneciente a una asociación.

Las condiciones representan una toma de decisión con dos posibles salidas. Para decidir por cual rama continuar se evalúa una expresión booleana. Esta expresión booleana puede estar compuesta de comparaciones aritméticas, operaciones booleanas, invocaciones a métodos sin parámetros cuyo tipo de retorno sea booleano, o valores booleanos.

Como vimos que podían ser de utilidad decidimos crear una construcción explícita par los ciclos. Los ciclos pueden ejecutarse sobre elementos de una colección, una cantidad fija de pasos (desde 1 hasta N) o hasta que una condición booleana evalúe false. También tienen un destino de transición para ir luego de culminar la ejecución.

C.2. DSL4MASS Textual

El lenguaje textual ofrece las mismas construcciones que el gráfico. Se definieron dos gramáticas diferentes, una para la representación de la estructura estática y otra para la estructura dinámica.

Las gramáticas se especificaron utilizando ANTLR [8] e incluimos las mismas en las siguientes subsecciones. Al principio de cada subsección se dan primero ejemplos de como definir cada una de las entidades principales.

C.2.1. Estructura Estática

El siguiente fragmento de código muestra el esqueleto de definición del modelo.

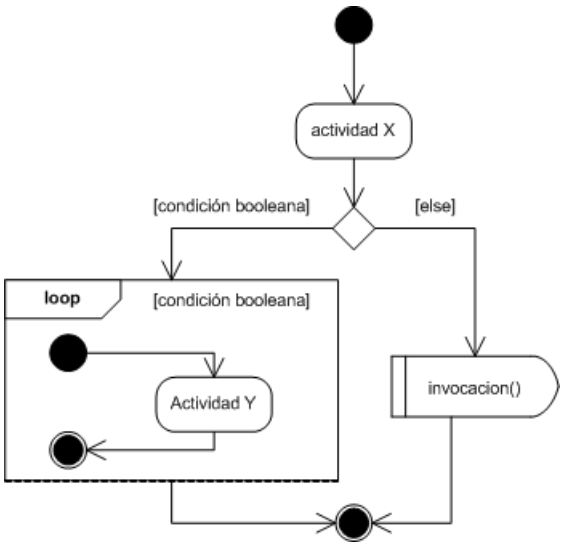

Figura C.2: Construcciones de estructura dinámica

```
1 Model <<identificador>> {
2    <<cuerpo de clase>>
3 }
```

A continuación se muestra la definición de un agente localizado. La definición de agentes no localizados simplemente no utiliza la palabra 'Located' al principio. Lo mismo sucede con los recursos localizados y no localizados.

```
1 Located Agent <<identificador>> {
2    <<cuerpo de clase>>
3 }
```

```
1 Located Resource <<identificador>> {
2    <<cuerpo de clase>>
3 }
```

```
1 Cell <<identificador>> {
2    <<cuerpo de clase>>
3 }
```

El cuerpo de las clases pueden tener definición de métodos y atributos como se muestra a continuación.

```
1  instance var <<tipo>> <<nombre>>;
2  static var <<tipo>> <<nombre>>;
3  instance var <<tipo>> <<nombre>> isProbe;
4  static var <<tipo>> <<nombre>> isProbe;
5
6  instance method <<tipo>> <<nombre>>(<<lista parametros>>);
7  instance method <<tipo>> <<nombre>>() isProbe;
8  static method <<tipo>> <<nombre>>(<<lista parametros>>);
9  instance method void <<nombre>>() isStep;
10 instance method void <<nombre>>() isScheduler;
```

A continuación mostramos los tokens usados por la gramática:

```
MODEL='Model';
AGENT = 'Agent';
LOCATED='Located';
RESOURCE = 'Resource';
CELL = 'Cell';
STATIC = 'static';
METHOD = 'method';
INSTANCE = 'instance';
VAR = 'var';
INITIALIZER = 'isInitializer';
SCHEDULER = 'isScheduler';
PROBE = 'isProbe';
STEP = 'isStep';
EXTENDS = 'extends';
PTC = ';';
INT = 'int';
FLOAT = 'float';
CHAR = 'char';
STRING = 'string';
LIST = 'list';
SET = 'set';
OBJECT = 'object';
BEGIN = '{';
END = '}';
ASSOCIATION = 'association';
TO_MANY = 'toMany';
TO_ONE = 'toOne';
VOID = 'void';
BOOL = 'bool';
```

La gramática definidas es:

```
dsl4mass_def:
   ( model_def
   | agent_def
   | agent_extension
   | resource_def
   | resource_extension
   | environment_cell_def )* EOF;

model_def:
   MODEL IDENTIFIER
   BEGIN (attribute_def | method_def)* END;

agent_def:
   (LOCATED)? AGENT IDENTIFIER
   BEGIN entity_body* END;

entity_body:
   ( attribute_def
   | method_def
   | association_def );

```

```
22 agent_extension:
23    IDENTIFIER EXTENDS AGENT IDENTIFIER
24    BEGIN entity_body* END;
25
26 resource_def:
27    (LOCATED)? RESOURCE IDENTIFIER
28    BEGIN entity_body* END;
29
30 resource_extension:
31    IDENTIFIER EXTENDS RESOURCE IDENTIFIER
32    BEGIN entity_body* END;
33
34 environment_cell_def:
35    CELL IDENTIFIER
36    BEGIN entity_body* END;
37
38 attribute_def:
39    member_modifier VAR basic_type IDENTIFIER (PROBE)? PTC;
40
41 member_modifier:
42    INSTANCE | STATIC;
43
44 method_def:
45    member_modifier METHOD basic_type IDENTIFIER
46    '(' (paramList)? ')' (stereotype)?
47    BEGIN methodBody_def END;
48
49 stereotype:
50    INITIALIZER | PROBE | SCHEDULER | STEP;
51
52 paramList:
53    param ( ',' param )*;
54
55 param:
56    basic_type IDENTIFIER;
57
58 methodBody_def:
59   (COMMENT)*;
60
61 association_def:
62    ASSOCIATION (IDENTIFIER)? (TO_MANY | TO_ONE)
63    IDENTIFIER IDENTIFIER PTC;
64
65 basic_type:
66    VOID | INT | FLOAT | CHAR | STRING
67    | LIST | SET | OBJECT | BOOL;
68
69 COMMENT:
70    '/*' .* '*/';
71
72 IDENTIFIER:
73    ('a'..'z'|'A'..'Z')
74    ('a'..'z'|'A'..'Z'|'0'..'9'|'-'|'_')*;
```

C.2.2. Estructura Dinámica

Los diagramas de actividad están contenidos en un elemento el tipo 'ActivityFlow' que los representa. Como indica la gramática pueden haber varios de estos elementos en el mismo archivo especificando diferentes diagrama de actividad. Cada 'ActivityFlow' esta asociado a un método particular de una entidad de la realidad (agente, recurso, celda). Este elemento también define cual es el elemento que inicia el diagrama de actividad. Un 'ActivityFlow' se define de la siguiente manera:

```
1 ActivityFlow for <<método>> of Entity <<entidad>> {
2    start destination <<identificador>>
3    ...
4 }
```

Donde <<método>>debe ser reemplazado por el método para el cual se esta haciendo el diagrama, <<entidad>>la entidad de la realidad que posee dicho método, y <<identificador>>es el nombre del elemento que inicia el diagrama de actividad. Dentro del 'ActivityFlow' (entre los símbolos '{' y '}') se puede especificar un conjunto de elementos que representan las actividades, invocaciones a métodos, condiciones y ciclos del diagrama.

Al igual que 'ActivityFlow' todos los elementos tienen un nombre que los identifica y un destino final, a excepción de las condiciones que tienen dos posibles destinos finales.

Las actividades solo tienen un nombre que las identifica y el elemento del diagrama por el cual continuar. Se definen de la siguiente manera:

```
1 Activity <<nombre>> destination <<identificador>>
```

Las invocaciones a métodos especifican sobre que elemento se ejecutan (el cual debe ser visible a través de una asociación de cardinalidad uno), el método a ejecutar y una lista de parámetros entre paréntesis (que por el momento debe ser vacía). Se definen de la siguiente manera:

```
1 call <<nombre>> (<<identificador>>.)?<<identificador>>()
2 destination <<identificador>>
```

Las condiciones están definidas por sus dos posibles destinos y una condición booleana que determina por cual de ellas se continúa. Esta se define de la siguiente manera:

```
1 condition <<nombre>>
2 trueDestination <<identificador>>
3 falseDestination <<identificador>>
4 with <<condicion booleana>>
```

Los ciclos se definen a partir de una condición de iteración, y el cuerpo de un 'ActivityFlow'. La condición puede tomar 3 formas: expresión booleana, cantidad fija de iteraciones o la iteración de una colección asociada al elemento para el que se define el diagrama. El ciclo se define de la siguiente manera:

```
1 loop <<nombre>> destination <<identificador>>
2 <<loop condition>> {
3    start destination <<identificador>>
4 }
```

<<loop condition>>puede tomar las formas:

```
with <<expresión booleana>>
```

Para iterar respecto de una condición booleana

```
for <<identifier>> = <<valor>> to <<valor>>
```

Para iterar una cantidad fija de veces y tener el valor actual disponible durante la iteración

```
foreach <<identifier>> in <<identifier>>
```

Para iterar respecto a una colección y tener el elemento actual disponible durante la iteración
A continuación mostramos los tokens usados por la gramática:

```
ACTIVITYFLOW='ActivityFlow';
FOR='for';
METHOD='method';
BEGIN='{';
END='}';
OF='of';
ENTITY='Entity';
START='start';
DESTINATION='destination';
ACTIVITY='activity';
CONDITION='condition';
TRUEDESTINATION='trueDestination';
FALSEDESTINATION='falseDestination';
WITH='with';
LOOP='loop';
EQUALS='=';
TO='to';
FOREACH='foreach';
IN='in';
CALL='call';
TARGET='target';
DOT='.';
PAR_OPEN='(';
PAR_CLOSE=')';
B_OPEN='¿';
B_CLOSE='?';
COLON=',';
PTC=';';
COMMENT_STRING='comment';
```

La gramática definidas es:

```
activity4mass:
   (activity_body_def)* EOF;

activity_body_def:
   ACTIVITYFLOW FOR METHOD IDENTIFIER OF ENTITY IDENTIFIER
   BEGIN activity_flow_body_def END;
```

```
 7
 8  activity_flow_body_def:
 9     start_def PTC (element_def PTC)*;
10
11  start_def:
12     START DESTINATION IDENTIFIER;
13
14  element_def:
15     activity_def (COMMENT_STRING STRING_VALUE)?
16    | condition_def (COMMENT_STRING STRING_VALUE)?
17    | loop_def (COMMENT_STRING STRING_VALUE )?
18    | call_def (COMMENT_STRING STRING_VALUE)?;
19
20  activity_def:
21     ACTIVITY IDENTIFIER DESTINATION IDENTIFIER;
22
23  condition_def:
24     CONDITION IDENTIFIER
25     TRUEDESTINATION IDENTIFIER FALSEDESTINATION IDENTIFIER
26     WITH bool_condition ;
27
28  bool_condition:
29     bool_condition_atom
30     (BOOLEAN_BINARY_OPERATOR bool_condition_atom)*;
31
32  bool_condition_atom:
33     BOOL_VALUE
34     | '(bool)' method_invocation
35     | arithmetic_expresion
36       ARITHMETIC_COMPARATOR
37       arithmetic_expresion
38     | B_OPEN bool_condition B_CLOSE
39     | BOOLEAN_UNARY_OPERATOR bool_condition;
40
41  arithmetic_expresion:
42     arithmetic_expresion_low
43     (ARITMETIC_OPERATOR_HIGH arithmetic_expresion_low)*;
44
45  arithmetic_expresion_low:
46     arithmetic_expresion_atom
47     (ARITMETIC_OPERATOR_LOW arithmetic_expresion_atom)*;
48
49  arithmetic_expresion_atom:
50     NUMBER_VALUE
51     | '(number)' method_invocation
52     | PAR_OPEN arithmetic_expresion PAR_CLOSE;
53
54  loop_def:
55     LOOP IDENTIFIER DESTINATION IDENTIFIER
56     loop_condition BEGIN activity_flow_body_def END;
57
58  loop_condition:
59     WITH bool_condition
60     | for_condition
61     | foreach_condition;
62
```

```
63  for_condition:
64     FOR IDENTIFIER EQUALS NUMBER_VALUE TO NUMBER_VALUE;
65
66  foreach_condition:
67     FOREACH IDENTIFIER IN IDENTIFIER;
68
69  method_invocation:
70     (IDENTIFIER DOT)? IDENTIFIER argument_list PAR_CLOSE;
71
72  argument_list:
73     argument (COLON argument)* | /*vacio*/ ;
74
75  argument:
76     IDENTIFIER
77     | n=NUMBER_VALUE
78     | c=CHAR_VALUE
79     | s=STRING_VALUE;
80
81  call_def:
82     CALL IDENTIFIER
83     DESTINATION IDENTIFIER TARGET method_invocation;
84
85  NUMBER_VALUE:
86     ('0'..'9')* (DOT ('0'..'9')*)?;
87
88  CHAR_VALUE:
89     '\''.'\'';
90
91  STRING_VALUE:
92     '\"'.*'\"';
93
94  ARITHMETIC_COMPARATOR:
95     '<' | '>' | '==' | '<=' | '>=' | '!=' ;
96
97  ARITMETIC_OPERATOR_HIGH:
98     '*' | '/' | '%' ;
99
100 ARITMETIC_OPERATOR_LOW:
101    '+' | '-'  ;
102
103 BOOL_VALUE:
104    'true' | 'false' ;
105
106 BOOLEAN_BINARY_OPERATOR:
107    'and' | 'or' ;
108
109 BOOLEAN_UNARY_OPERATOR:
110    'not' ;
111
112 COMMENT:
113    '/*' .* '*/';
114
115 IDENTIFIER:
116    ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'-'|'_')*;
```

Apéndice D

Gestión del Proyecto

En este anexo se detallan aspectos de la gestión del proyecto. Esto incluye las tareas llevadas a cabo y la duración de las mismas.

El equipo estaba desarrollado por dos personas, quienes participamos en todas las tareas. La mayor parte de estas fueron realizadas en conjunto.

Las tareas fueron planificadas al comienzo del proyecto, pero la estimación de plazos de las mismas se fueron ajustando en el transcurso del proyecto. En las siguientes secciones se describen las tareas realizadas y se finaliza con un cronograma que muestra los tiempos de las tareas.

Estado del Arte

Consistió en la búsqueda de material teórico referente a sistemas multiagente, simulación y DSL, así como también herramientas y lenguajes existentes que cubran parcial o totalmente el desarrollo de simulaciones basadas en la tecnología multiagentes.

Si bien el progreso no fue estrictamente en cascada, las etapas llevadas a cabo fueron:

- Búsqueda en amplitud de material.
- Estudio de los conceptos y el contexto del proyecto, clasificando el material obtenido en la etapa anterior.
- Análisis de las herramientas, definiendo las características deseadas.

Definición del Lenguaje

Luego de estudiar los conceptos intervinientes en el contexto del proyecto y haber evaluado el soporte existente, se diseñó el DSL y se implementó la herramienta para resolver los problemas planteados al proyecto. Las principales etapas fueron:

- Especificación del lenguaje para describir la estructura estática de las simulaciones.
- Especificación para describir la estructura dinámica.
- Implementación de la gramática y generador de código. La gramática se escribía mientras se definían las construcciones.

Caso de Estudio

Con el fin de evaluar el DSL y la herramienta construida, se desarrolló un caso de estudio basado en una simulación real. En cada etapa del desarrollo se observaron las mejoras introducidas.

Las principales tareas de esta etapa fueron:

- Estudio detallado de la realidad.
- Implementación con DSL4MASS y la herramienta construida.
- Observación de esfuerzo realizado.
- Comparación con la metodología tradicional durante todo el desarrollo del caso de estudio.

Documentación

Consistió en la documentación del trabajo realizado. Esto requirió:

- Recopilación de documentación generada en etapas previas.
- Generación, organización y categorización de contenido.
- Revisiones intermedias.

Cronograma

En la figura D.1 se presenta el cronograma del proyecto, el cual comienza a principios de abril del año 2008 y culmina en julio del 2009.

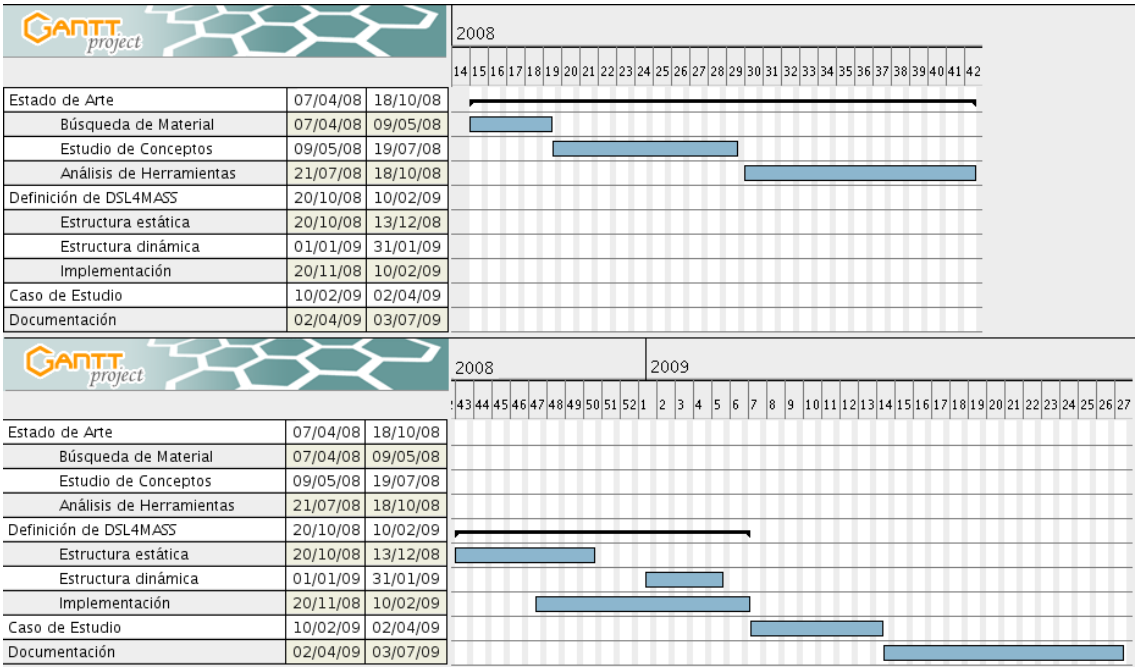

Figura D.1: Cronograma del proyecto