UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADA EN INGENIERÍA DEL SOFTWARE

Simulación de Diagramas de Secuencia con la herramienta de modelado USE

Sequence Diagrams Simulation with the USE Modeling Tool

Realizado por
Paula Muñoz Ariza

Tutorizado por
Antonio Vallecillo Moreno
Lola Burgueño Caballero

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2019

Fecha defensa:

Fdo. El/la Secretario/a del Tribunal

Resumen

Este proyecto se enmarca en el paradigma de la ingeniería del software dirigida por modelos, puesto que se pretende crear una herramienta que facilite la comprensión y la validación del comportamiento de modelos de software definidos en una herramienta de modelado denominada USE.

La realización de este proyecto tiene como objetivo principal desarrollar una extensión para USE que permita la simulación de diagramas de secuencia generales. Actualmente, la herramienta USE permite la simulación de diagramas de secuencia que correspondan con una secuencia dada de interacciones entre los objetos del sistema. Nuestra herramienta permitiría que se describieran diagramas de secuencia generales como los descritos en UML 2.0 y que se obtuvieran las secuencias de interacciones ejecutables en USE para todos y cada uno de los casos concretos que se derivaran de estos diagramas generales.

Para expresar los diagramas de secuencia generales de forma textual se ha decidido emplear la recomendación Z.120 de la ITU-T que se denomina Message Sequence Chart (MSC). Los diagramas definidos en UML 2.0 son un subconjunto de esta recomendación.

El módulo principal del proyecto es un generador automático de instancias a partir de una descripción general del diagrama de secuencia.

Palabras clave: modelado, validación, automatización, UML, diagramas de secuencia, MSC, USE

Abstract

This project comes under the Model Driven Software Engineering paradigm, because we intend to create a tool that facilitates validation and comprehension of software models defined in a modeling tool called USE.

The aim of this project is to develop a plug-in for USE which allows general sequence diagrams simulation. In its current version, this tool supports simulation of sequence diagrams which correspond to a certain sequence of interactions between system objects. Our tool allows to describe sequence diagrams as the ones described in UML 2.0 and to generate sequences of interactions which may be executed in USE. These sequences correspond to every possible outcome of the execution of the general sequence diagram.

To define textually general sequence diagrams, we chose the ITU-T recommendation z.120, also called Message Sequence Chart (MSC). The diagrams defined in UML 2.0 are a subset of this recommendation.

The sequence diagram instance generator has been developed as an Eclipse plugin.

Keywords: modeling, validation, automation, UML, sequence diagrams, MSC, USE

Índice

1

Introducción

Durante este capítulo presentaremos el contexto que ha llevado a la decisión de desarrollar esta aplicación. En las próximas páginas desarrollaremos la motivación y los objetivos enmarcados en este proyecto. Adicionalmente, explicaremos la estructura de la memoria, las tareas que se van a llevar a cabo y la metodología aplicada para el desarrollo del proyecto.

1.1. Motivación

La ingeniería de software dirigida por modelos (o MDSE por sus siglas en inglés, Model-Driven Software Engineering) es un paradigma de ingeniería de software en donde los modelos (es decir, representaciones abstractas de los conocimientos y actividades que rigen un dominio de aplicación particular) y las transformaciones entre ellos, se convierten en las piezas claves de los procesos de ingeniería (Brambilla, Cabot, & Wimmer, 2017).

Este nuevo paradigma está tomando cuerpo y siendo adoptado cada vez más por las empresas de desarrollo de software (Vallecillo, 2014) (Selic, 2012) y se ha convertido en una materia propia de cualquier titulación universitaria en Ingeniería Informática.

Dos aspectos fundamentales para conseguir una mayor adopción de MDE en la industria pasan por mejorar tanto la calidad de las herramientas, como la calidad de la docencia en estos temas (Selic, 2012). En muchas universidades se ha comenzado a trabajar tratando de aunar ambos temas y se han obtenido resultados muy positivos (Burgueño, Vallecillo, & Gogolla, 2018). Además, hay que tener en cuenta que, al tratarse la Ingeniería Informática de una disciplina "ingenieril", es fundamental dar un enfoque adecuado a la docencia que se imparte a los alumnos de estas titulaciones (Offutt, 2013). En particular, es importante que los alumnos cuenten con herramientas que les permitan "aprender haciendo". Por ello, en MDE los modelos han pasado de ser meros dibujos y bosquejos, cuya única función es la de representar un sistema, a artefactos software que pueden ser utilizados para simular los sistemas que modelan, probar propiedades sobre ellos y construir prototipos.

Una de las herramientas de modelado comúnmente utilizada es "USE" (UML-based Specification Environment) (Gogolla, Büttner, & Richters, 2007), pues permite no sólo especificar sistemas software mediante UML y OCL, sino también analizarlos de distintas formas y simularlos (Büttner & Gogolla, 2014). Esta herramienta es Open Source y actualmente muy popular, sobre todo en investigación y en docencia (Agner & Lethbridge, 2017), y es una de las que se utiliza en distintas universidades para enseñar tanto modelado como métodos formales en ingeniería del software.

Por otro lado, los diagramas de secuencia constituyen una de las principales notaciones a la hora de modelar sistemas, pues permiten capturar las interacciones entre sus distintos objetos de un sistema, centrándose en el orden parcial en el que se pueden realizar los intercambios de mensajes entre ellos. Este tipo de notación ha sido clave a la hora de definir y especificar protocolos de comunicación, y fue estandarizada por la Unión Internacional de Telecomunicaciones (ITU-T) en 2011, con la recomendación Z.120 que define los Message Sequence Chart (MSC) (ITU-T, Message Sequence Chart,

Annex B: Formal semantics of Message Sequence Chart, 1998). UML también implementa los Diagramas de Secuencia, que no son sino un subconjunto de esa notación, y totalmente compatible con la recomendación Z.120 a partir de la versión 2.0 de UML.

Aunque aparentemente bastante intuitivos, los diagramas de secuencia no son tan fáciles de comprender ni de utilizar (Keng & Poi-Peng, 2006). Una de las principales razones es que su semántica no es tan simple como aparenta. Esto se une a que normalmente las herramientas con las que se aprenden no permiten simular ni prototipar (ejecutar) los diagramas de secuencia que se especifican, lo cual dificulta notablemente su aprendizaje. Es como si se tratase de enseñar un lenguaje de programación sin un entorno que permitiera ejecutar los programas.

Actualmente la herramienta USE permite especificar diagramas de secuencia sólo de forma parcial: dado un diagrama de objetos, a partir de una secuencia de instrucciones que representan invocaciones a métodos de los objetos, USE construye el diagrama de secuencia que describe dichas interacciones. De este modo, los diagramas de secuencia representan una única ejecución del sistema. Sin embargo, la herramienta no permite especificar diagramas de secuencia generales que representen todas las posibles interacciones válidas en un sistema.

El desarrollo de esta proyecto, pretende mejorar la situación anteriormente descrita, proporcionando a USE una funcionalidad adicional, apoyada en una descripción empleando la recomendación Z.120, que permita la definición de diagramas de secuencia generales y la ejecución de las distintas instancias derivadas de los mismos.

1.2. Objetivos

El objetivo principal de este proyecto es desarrollar una herramienta que dote a USE de las funcionalidades necesarias para soportar la simulación de diagramas de secuencia generales. Esto permitiría a los usuarios
generar diferentes escenarios posibles, que puedan ser validados (corrección, satisfacibilidad), ejecutados y representados de forma gráfica.

A partir de un modelo de clases definido en USE y un diagrama de secuencia general definido de acuerdo a la recomendación Z.120, seremos capaces de generar archivos con secuencias que den lugar a todas las ejecuciones posibles del diagrama. Las instancias y las funciones que tomen parte deberán estar definidas a partir de las clases del modelo de USE.

Con estos archivos el usuario podría comprobar si el sistema alcanza estados inesperados o si la ejecución alcanza estados de error. Introduciendo los archivos en USE, podrá comprobar el estado final del sistema y si las condiciones impuestas satisfacen. Esto permitirá sin duda a los modeladores comprender mejor el significado y comportamiento de los diagramas de secuencia que desarrollen. Este proyecto también permitirá mejorar la docencia de este tipo de notación, pues los alumnos serán capaces de comprender mejor el significado de los diagramas de secuencia de UML y su uso.

Por último, al ser una herramienta creada como Trabajo de Fin de Grado, en los objetivos también debe incluirse todo el aprendizaje que ha sido necesario para acometer la tarea. Durante el desarrollo, se han tenido que poner en práctica muchos conocimientos obtenidos durante la carrera, a la vez que se han tenido que adquirir

muchos otros, lo que ha hecho de este proyecto una experiencia doblemente enriquecedora.

1.4. Tareas a desarrollar

Para llevar a cabo los objetivos expuestos en el apartado anterior, se han desarrollado las siguientes tareas:

Tarea 1. Definir un metamodelo que sea capaz de representar el lenguaje textual para describir diagramas de secuencia de acuerdo a la recomendación Z.120 o MSCs.

Tarea 2. Construir un analizador sintáctico (en inglés, *parser*) que convierta un MSC en un modelo conforme al metamodelo anterior.

Tarea 3. A partir del modelo anterior, desarrollar un generador que permita generar secuencias de comandos SOIL que describan todas las ejecuciones válidas del sistema con respecto al diagrama de secuencia correspondiente.

1.5. Metodología y fases de trabajo

1.5.1. Metodología

Durante el desarrollo del proyecto, se ha seguido una metodología de desarrollo iterativa incremental. Esta metodología está enmarcada en las metodologías ágiles para el desarrollo de software, y en ella, el software se desarrolla en iteraciones. En cada una de estas iteraciones, una parte del sistema es desarrollado, testado y entregado. Esto quiere decir que en cada una de las iteraciones la funcionalidad será mejorada y que en cada lanzamiento habrá una nueva funcionalidad añadida. (Abbas, Gravell, & Wills, 2008)

Para el desarrollo del proyecto, se ha decidido emplear esta metodología sobre otras, por la complejidad del lenguaje MSC. En cada una de las iteraciones, se irá trabajando sobre un subconjunto cada vez más grande del lenguaje. Dado el gran número de constructores que posee, en cada iteración se identificará un subconjunto apropiado del lenguaje el cual será una extensión del lenguaje considerado en la fase anterior. Cada fase dará lugar a un generador de secuencias SOIL para el subconjunto seleccionado.

En cuanto al trabajo, también seguimos un enfoque incremental que permitió realizar continuas iteraciones de tiempo variable con un máximo de dos semanas en cada una de las tareas que componen el proyecto. En cada iteración se obtuvo una nueva versión del producto que se esté desarrollando y permita su evaluación.

Simulando un caso real de desarrollo de un producto, al final de cada iteración, el autor del proyecto (estudiante), se reunió con los clientes (tutores), para valorar el resultado obtenido y establecer los objetivos de la siguiente iteración. De esta forma, se permitía la corrección rápida de errores debidos a falta de entendimiento o de especificación en algunos requisitos.

1.5.2. Fases de trabajo

El proyecto se dividió en las siguientes fases para su desarrollo:

Fase 1 - Estado de la cuestión

- Estudio del lenguaje de especificación de diagramas de secuencia, descrito en la recomendación Z.120 del ITU-T
- Estudio de la herramienta USE y del lenguaje SOIL

Fase 2 - Formación en las herramientas de desarrollo

- Estudio de la herramienta Xtext y del lenguaje Xtend para realizar el analizador sintáctico y el generador de archivos.

- ❖ Realización de pequeños ejemplos iniciales para entender por completo el empleo de la herramienta.

Fase 3 - Selección del subconjunto de la gramática del MSC

- ❖ Selección del subconjunto de la gramática a emplear para la realización del proyecto completo, de acuerdo a las funcionalidades que deseábamos cubrir.

Fase 4 - Desarrollo del generador de secuencias SOIL

- ❖ Seleccionar un subconjunto del subconjunto seleccionado en la fase anterior.
- ❖ Definir las reglas de la gramática correspondiente en Xtext.
- ❖ Implementar el generador de archivos correspondiente para el subconjunto selecciónalo.

Esta fase se repetirá de forma iterativa hasta completar la implementación para el subconjunto completo seleccionado en la fase anterior.

1.6. Estructura de la memoria

En este apartado se puede encontrar un breve resumen de cada uno de los siguientes apartados de la memoria.

Capítulo 2 - Estado del arte

En este capítulo expondremos las bases paradigma de la Ingeniería del Software Dirigida por Modelos, que es la rama en la que se enmarcar nuestro proyecto. Además hablaremos de algunos de los problemas a los que se enfrenta esta disciplina a día de hoy. Finalmente, compararemos nuestro proyecto con otros similares.

Capítulo 3 - Tecnologías empleadas

En este capítulo se expondrán las características de las tecnologías empleadas para el desarrollo del proyecto. Nos centraremos en las características que puedan estar relacionadas de forma directa con el proyecto.

Capítulo 4 - Estructura e implementación

En este capítulo explicaremos el funcionamiento concreto del algoritmo, con todos sus entresijos. Nos centraremos en qué problemas hemos resuelto y en qué enfoques se han seguido para su resolución, desgranando su funcionamiento.

Capítulo 5 - Validación y pruebas

En este capítulo enumeraremos las pruebas a las que hemos sometido al algoritmo para su validación y explicaremos el por qué hemos elegido la metodología empleada.

Capítulo 6 - Conclusiones

Para efectuar el cierre de la memoria, se expondrán las conclusiones obtenidas en el desarrollo de la herramienta, hablando de los problemas encontrados y los conocimientos que ha sido necesario adquirir para superarlos. Adicionalmente, nombramos algunas posibles líneas futuras para extender la aplicación.

2

Estado del arte

En este capítulo hablaremos del estado actual de los ámbitos con los que está relacionado el proyecto: la Ingeniería del Software Dirigida por Modelos y la generación automática de casos de prueba a partir de diagramas de secuencia.

Con respecto a la Ingeniería del Software Dirigida por Modelos, se expondrá su definición y principios básicos, además de una de las problemáticas con las que se le relaciona en el momento actual. En el siguiente apartado, hablaremos de la generación automática de casos de prueba a partir de diagramas de secuencia y mencionaremos algunos trabajos relacionados.

2.1. Ingeniería del Software Dirigida por Modelos

El principio básico de la Ingeniería del Software Dirigida por Modelos (por sus siglas en inglés, MDSE) es que *Todo es un modelo*. Con esto, podemos concluir que los modelos son el elemento fundamental para comprender y compartir el conocimiento sobre software complejo. (Brambilla, Cabot, & Wimmer, 2017)

En este paradigma, todos los elementos del software deben comprenderse como modelos que interactúan entre ellos conformando a su vez otro modelo adicional. Por ejemplo, las transformaciones deben ser entendidas como modelos que se aplican sobre otros modelos. Adicionalmente, si subimos el nivel de abstracción, un lenguaje de modelado debe ser también considerado un modelo y MDSE define esta circunstancia, denominando a dicho lenguaje como un metamodelo. La definición de modelos sobre modelos se vuelve recursiva y para definir o modelar un metamodelo, se deberá definir el meta-metamodelo.

El nacimiento de los primeros lenguajes de programación de más alto nivel tuvo como objetivo el separar a los desarrolladores de las abstracciones y dificultades que planteaban los entornos para los que se desarrollaba el software. Sin ir más lejos, el nacimiento de los sistemas operativos pretendía simplificar la experiencia de usuario, separándolo de las dificultades y conocimientos que había que tener para comunicarse de forma directa con el hardware.

Aunque los lenguajes de programación han permitido elevar el nivel de abstracción en el desarrollo, el avance tecnológico nos ha llevado a un incremento de la complejidad de las plataformas que se emplean para el desarrollo. Además, la mayoría de aplicaciones que se desarrollan actualmente, se mantienen de forma manual lo que implica un gasto excesivo de tiempo y esfuerzo. No sólo en su desarrollo, sino en las tareas de despliegue, configuración y validación. (Schmidt, 2006)

Este constante incremento en la complejidad de las plataformas, obliga a los desarrolladores a centrarse en detalles muy concretos de la programación y no pueden centrarse en realizar una estrategia que se centre en mejorar características arquitectónicas del sistema o de su rendimiento.

Actualmente, algunas de las soluciones que se proponen para solucionar este problema es el desarrollo de tecnologías enmarcadas en el paradigma de la Ingeniería del Software Dirigida por modelos.

En primer lugar, una de las soluciones que se consideran podría ser el empleo de lenguajes de modelado específicos de dominio (Schmidt, 2006). El empleo de estos lenguajes permite el desarrollo de aplicaciones empleando elementos tipados del sistema a desarrollar que se han capturado mediante el empleo de un metamodelo. Seguir este enfoque permite a los desarrolladores expresar los conceptos en el software de una forma más declarativa que imperativa, permitiéndoles visualizar el sistema desde un nivel de abstracción más alto.

En segundo lugar, se propone el empleo de sistemas de transformación y generación (Schmidt, 2006). Estas herramientas analizan ciertos aspectos de los modelos y generan a partir de ellos distintos tipos de artefactos software útiles en el desarrollo y el despliegue.

Nuestro proyecto está enmarcado en ambos puntos. Por un lado, definiremos un lenguaje específico de domino seleccionando un subconjunto de MSC que adaptaremos con el fin de alcanzar los objetivos establecidos. Por otro lado, definiremos un sistema de generación que se encargará de producir archivos con las distintas ejecuciones.

El enfoque dado en la realización de nuestro proyecto tiene como uno de los objetivos el expuesto en los párrafos anteriores. Durante este proyecto hemos pretendido simplificar la situación de los usuarios, aumentando el nivel de abstracción para que puedan centrarse en el diseño de alto nivel del sistema.

2.2. Generación automática de casos de prueba a partir de diagramas de secuencia

Obteniendo información sobre posibles trabajos relacionados en el campo, dimos con un amplio abanico de artículos y estudios que también generaban todas las instancias a partir de la definición de diagramas de secuencia generales.

El principal uso que hacían todos estos estudios era el de la generación automática de casos de prueba. En estos estudios, el usuario introducía un diagrama de secuencia de UML y se obtenían casos de prueba referentes al mismo en distintos formatos.

Uno de ellos es por ejemplo el estudio desarrollado por Vikas Panthi y Durga Prasad Mohapatra, para el National Institute of Technology de Rourkela (India) en 2013, llamado *Automatic Test Case Generation Using Sequence Diagram*. Su enfoque consistía en extraer toda la información posible de los diagramas de secuencia de UML y crear casos de prueba en la librería de Java JUnit. Uno de los problemas básicos eso es el expresar los diagramas de UML 2.0 en un formato escrito del que se pueda extraer la información fácilmente. En su estudio, ellos proponen el empleo de la librería ModelJUnit, que extiende la librería JUnit para permitir escribir diagramas de secuencia como clase de Java.

Otro de los trabajos es el desarrollado por Monalisa Sarma, Desasish Kundu y Rajib Mall, para el Indian Institute of Technology en el 2007, llamado *Automatic Test Case Generation from UML Sequence Diagrams*. En su enfoque, empleaban MagicDraw, herramienta de modelado comúnmente usada, para definir un diagrama de secuencia UML. Este diagrama se exportaba a XML y a partir de ahí, ellos se encargaban de extraer la información necesaria y generar un grafo que recorrerían para dar lugar a todos los casos de prueba.

Entre estos y otros proyectos similares, encontramos ciertas constantes. En primer lugar, todos se ven obligados a tomar una decisión acerca de cómo textualizar los diagramas de secuencia, puesto que UML sólo aporta una definición gráfica de los mismos. Además, el producto final siempre está enfocado a casos de prueba que permitan una alta cobertura de código en distintas categorías. Finalmente, muchos de ellos no permiten todas las combinaciones posibles en cuanto a fragmentos complejos como bucles o condicionales combinados o anidados.

Nuestro proyecto está creado en un contexto diferente con una intencionalidad algo distinta. Nosotros hemos seleccionado una recomendación para la textualización de UML, dando al usuario unas reglas precisas a las que ceñirse. Gracias a esta decisión, no le obligamos a contar con ningún software concreto para la definición de los diagramas, sólo para la obtención de los archivos de salida. Además, nuestro software permite todas las combinaciones posibles de fragmentos complejos, dando al usuario total libertad para la definición de sus diagramas. Finalmente, nuestro objetivo principal con este proyecto, como ya hemos expuesto en apartados anteriores, es dar al usuario una salida que ejecutar en un software de modelado, en una fase anterior al desarrollo de código. Este proyecto pretende alarmar de errores en etapas anteriores a la implementación, de forma que puede ser considerada un arma de prevención temprana. Adicionalmente, en su empleo en la docencia, también es una herramienta que permite a los estudiantes analizar el comportamiento de sus diagramas de secuencia de acuerdo a sus definiciones generales.

3

Tecnologías empleadas

En las siguientes páginas describiremos las tecnologías empleadas en el desarrollo de la herramienta. En primer lugar analizaremos las áreas que nos interesan de UML 2.0 y continuaremos relacionándolo con el lenguaje MSC. Después presentaremos la herramienta USE y sus funcionalidades con respecto a los diagramas de secuencia. Finalmente, analizaremos el funcionamiento de Xtend y Xtext, las herramientas empleadas en el desarrollo del generador de instancias que es el objetivo de este proyecto.

3.1. Diagramas de Secuencia en UML 2.0

Las siglas UML hacen referencia a Unified Modelling Language, traducido al español como *Lenguaje Unificado de Modelado.* Este lenguaje fue definido por el Object Management Group (OMG) y su primera versión oficial fue publicada en 1997. La versión a la que haremos referencia es a la 2.0, la cual fue publicada en 2005.

UML se define como un lenguaje visual para especificar, construir y documentar artefactos pertenecientes a sistemas (Group, UML 2.0 Infrastructure, 2005). Este lenguaje posee un abanico tan amplio de capacidades para la definición de sistemas e interacciones que puede ser usado para gran multitud de propósitos en distintos dominios de aplicación. Uno de sus principales objetivos es permitir la interoperabilidad entre herramientas de modelado, estableciendo unas normas generales a seguir en las definiciones de los modelos.

A pesar de los numerosos tipos de representaciones con los que cuenta UML, en este proyecto, nos centraremos en su capacidad para describir las interacciones entre distintos objectos que constituyen un sistema. Esta descripción se realiza mediante los denominados Diagramas de Interacción, aunque su variante más utilizada es el Diagrama de Secuencia.

Los Diagramas de Secuencia se enmarcan en la parte de UML creada con el fin de definir el comportamiento, es decir, la parte dinámica de los sistemas. En esta sección, la acción es la unidad fundamental de descripción. Una acción toma un conjunto de valores de entrada y los transforma en otro conjunto de valores de salida (ambos conjuntos pueden ser vacíos) (Group, UML 2.0 Superstructure, 2005).

Las interacciones son un tipo concreto de acción y serán el elemento central en los Diagramas de Secuencia. Las interacciones son intercambios de mensajes entre las *Lifelines.* Estas últimas representan a una entidad que interactúa en la secuencia de intercambio de mensajes (Group, UML 2.0 Superstructure, 2005).

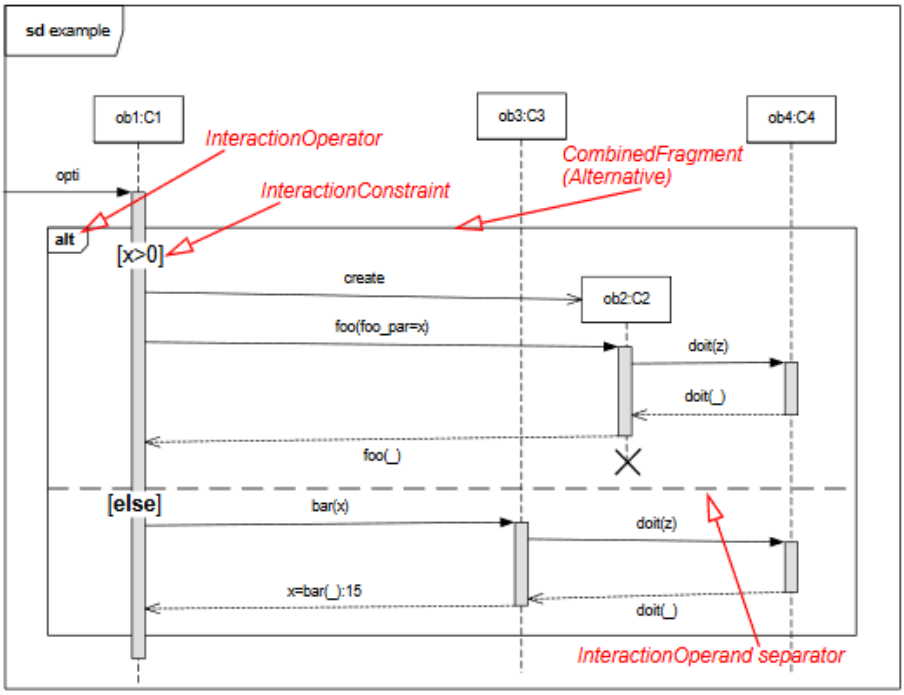


Figura 2.2.1. sd example (Group, UML 2.0 Superstructure, 2005)

En la figura 2.2.1. podemos ver un diagrama de secuencia con 4 Lifelines: *ob1, ob2, ob3, ob4.* La clase a la que pertenecen de estas instancias es el especificado tras los dos puntos: *C1, C2, C3, C4.* Cada una de estas *Lifelines* invoca una función de otra *Lifeline,* y esto se expresa mediante las flechas: las continuas son las llamadas a las funciones y las discontinuas son las respuestas a estas llamadas. Cuando las llamadas incluyen la respuesta, se habla de que la interacción es síncrona.

Por otro lado, también observamos que las interacciones están contenidas en un recuadro, esto les aplica distintos efectos dependiendo de su tipo, especificado en la esquina superior derecha. Estos recuadros son denominados por el manual de UML como *CombinedFragments* y su tipo es determinado por los llamados *interactionOperator*. Cada una de las secciones separadas por las líneas discontinuas horizontales, son lo que se denomina *interactionOperand*, al que nos referiremos como operando.

Los *interactionOperator* que serán interesantes para el desarrollo del proyecto serán los siguientes:

- **Alternatives (alt):** este operador permite describir una elección entre varios operandos. UML permite especificar una guarda que será evaluada a la hora de decidir cuál de los operandos será el que se aplique. Si alguno de los operandos tiene como guarda *else*, significa que su guarda es la negada a todas las demás.
- **Option (opt):** al igual que el anterior con la salvedad de que sólo puede especificarse un operando. Este último será ejecutado o no.
- **Paralell (par):** este operador indica que todos los operandos que contiene pueden ejecutarse de forma entrelazada en cualquier orden, respetando el orden interno de los operando.
- **Loop (loop):** este operador permite repetir el operando contenido un número de veces especificado. Este número de veces puede ser representado por un par de valores que serán el valor máximo y mínimo de repeticiones de pueden darse. UML también permite incluir una expresión booleana para determinar las repeticiones.

3.2. Z.120: Message Sequence Chart

Message Sequence Chart (MSC) es una recomendación desarrollada por la ITU Telecommunication Standarization Sector (ITU-T) y su primera versión fue publicada en 1993. Los documentos que emplearemos como referencia en este trabajo son el documento oficial publicado en 2011 y el Anexo B publicado en 1998.

El objetivo principal de la recomendación Z.120 es la de proveer un lenguaje que permita especificar y describir las distintas interacciones que se dan durante el intercambio de mensajes entre componentes de un sistema y su entorno (ITU-T, Message Sequence Chart (MSC), 2011).

Este lenguaje permite la definición de interacciones de la misma forma en que lo hacen los Diagramas de Secuencia de UML expuestos en el apartado anterior. La diferencia principal entre ambas es que el MSC tiene un mayor rango de elementos para la descripción de las interacciones. Las funcionalidades definidas para los Diagramas de Secuencia en UML 2.0 son sólo un subconjunto de las que son descritas en el MSC.

Esta recomendación provee tanto de una definición textual como de una escrita. Esto permite al usuario definir interacciones de forma unívoca sin necesidad de un software gráfico, empleando sólo la descripción textual.

Aunque la recomendación oficial propone unas reglas en muchos casos poco intuitivas, en el Anexo B, se propone un subconjunto del lenguaje con una simplificación en las interacciones que pretende proveer a cualquier usuario de una versión usable del sistema con las funcionalidades básicas. En este anexo se simplifican las expresiones de forma que sólo quede lo estrictamente necesario para emplear las funcionalidades básicas que permite la recomendación.

Tomando como referencia este anexo, podemos visualizar en la figura 2.2.3 uno de los ejemplos más sencillos de un MSC. En él, podemos observar las llamadas asíncronas entre las distintas instancias, que al igual que en UML se expresan mediante flechas. Las instancias, denominadas en el ejemplo como *i1, i2, i3 e i4,* son lo equivalente a las *Lifelines* que se definen en UML.

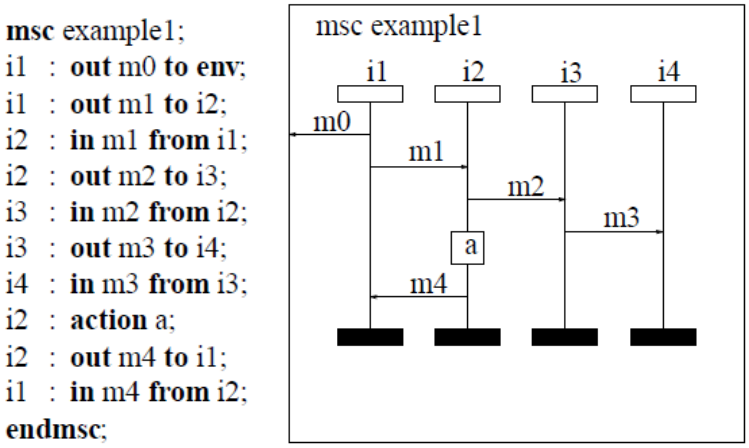


Figura 3.2.1. Representación equivalente escrita y gráfica de un MSC. (ITU-T, Message Sequence Chart, Annex B: Formal semantics of Message Sequence Chart, 1998)

En el MSC, cada llamada se especifica en dos pasos:

- El envío del mensaje, especificando que se ha realizado la llamada (*<emisor> : out <mensaje> to <receptor>)*
- La recepción del mensaje, especificando que la llamada ha sido recibida. (*<receptor> : in <mensaje> from <emisor>)*

Al igual que en UML 2.0, también se definen las sentencias de selección, iteración y paralelización. Estas sentencias son equivalentes a las expuestas con anterioridad con la salvedad de que en el *alt* la palabra reservada para expresar que la guarda es la opuesta a todas las anteriores es *otherwise*. A pesar de que en la recomendación se permite la definición de guardas en este tipo de sentencias, nosotros no las consideraremos. A excepción de la de loop, que tendrá un límite superior y un límite inferior.

El formato gráfico para la representación de estas sentencias es igual al empleado en UML y lo podemos observar en la figura 2.3.2.

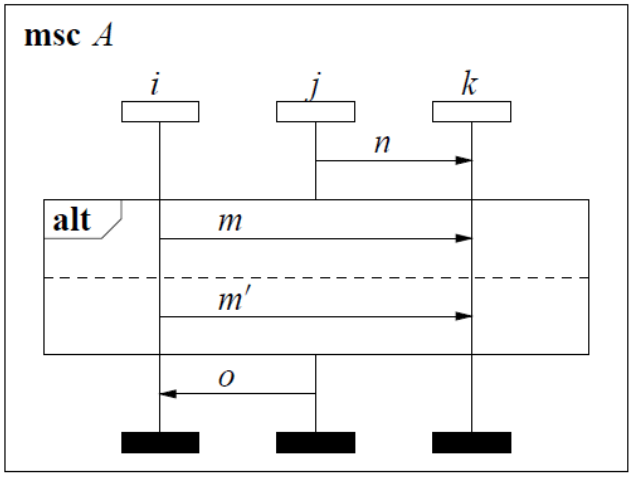


Figura 3.2.2. Ejemplo de alt. (ITU-T, Message Sequence Chart, Annex B: Formal semantics of Message Sequence Chart, 1998)

El formato textual de todas las sentencias anteriormente nombradas puede observarse en la figura 2.3.3. En todas las sentencias, antes de las palabras reservadas *<tipo de sentencia> begin,* se enumeran todas las instancias que participarán en el intercambio de mensajes en su interior. La palabra detrás de la palabra reservada *begin* expresa el nombre del conjunto que compone la sentencia, no aporta ninguna otra información adicional más que añadir una forma de referirse a ella. En el interior de la sentencia, entre el *begin* y el *end*, se pueden añadir otras sentencias e interacciones.

```
msc ejemplo01
(inst uno:Uno, dos:Dos, tres:Tres)

Ch, uno: opt begin optEjemplo;
    Ch : call unoX to uno;
    uno : receive unoX from Ch;
    Ch : replyout dosX to uno;
    uno : replyin dosX from Ch;
opt end;

Ch, uno: alt begin altEjemplo;
    Ch : call unoX to uno;
    uno : receive unoX from Ch;
    Ch : replyout dosX to uno;
    uno : replyin dosX from Ch;
alt;
    Ch : call unoX to uno;
    uno : receive unoX from Ch;
    Ch : replyout dosX to uno;
    uno : replyin dosX from Ch;
alt end;

Ch, uno: par begin parEjemplo;
    Ch : call unoX to uno;
    uno : receive unoX from Ch;
    Ch : replyout dosX to uno;
    uno : replyin dosX from Ch;
par;
    Ch : call unoX to uno;
    uno : receive unoX from Ch;
    Ch : replyout dosX to uno;
    uno : replyin dosX from Ch;
par end;

Ch, uno: loop<1,2> begin loopEjemplo;
    Ch : call unoX to uno;
    uno : receive unoX from Ch;
    Ch : replyout dosX to uno;
    uno : replyin dosX from Ch;
loop end;

endmsc;
```

Figura 3.2.3. Ejemplos de todas las sentencias en formato textual.

A pesar de todo lo que incluye el Anexo, la simplificación no permite especificar algunos aspectos que hemos considerado esenciales para el proyecto pero que sí están incluidos en la versión completa de la recomendación. Todos estos elementos seleccionados se enumerarán en los siguientes párrafos.

La definición de los tipos de las instancias en el MSC se realiza antes de comenzar la interacción y con el siguiente formato: (inst <nombre de la instancia>:<tipo de la instancia> (, <nombre de la instancia>:<tipo de la instancia>)+). En la figura 2.3.4.

podemos observar cómo se especifica, por ejemplo, la instanciación del objeto sA que es de tipo ServerA.

```
msc Test01
 (inst sA:ServerA, sB:ServerB, cA:ClientA, cB:ClientB)
cB : call sA() to cA;
 cA : receive sA() from cB;
 cA : replyout sA() to cB;
 cB : replyin sA() from cA;
 cA : call sB() to cB;
 cB : receive sB() from cA;
 cB : replyout sB() to cA;
 cA : replyin sB() from cB;
 cA : call sB() to cB;
 cB : receive sB() from cA;
 cB : replyout sB() to cA;
 cA : replyin sB() from cB;
 endmsc;
```

Figura 3.2.4. Ejemplo de interacción

Para la especificación de llamadas síncronas, es necesario emplear una especificación distinta al in y out que vimos en los párrafos anteriores. Como podemos apreciar en la figura 2.3.4., emplearemos las sentencias *call, receive, replyout, replyin.* Estas sentencias al igual que las anteriores, se especifican en parejas y con el siguiente formato:

- **Call:** envío del mensaje por el emisor *(<emisor> : call <mensaje> to <receptor>)*
- **Receive:** recepción del mensaje por el receptor *(<receptor> : receive <mensaje> from <emisor>)*
- **Replyout:** envío de la respuesta al mensaje por el receptor *(<receptor> : replyout <mensaje> to <emisor>)*
- **Replyin:** recepción de la respuesta al mensaje por el emisor *(<emisor> : replyin <mensaje> from <receptor>)*

La representación gráfica de las llamadas síncronas coincide con la representación que se les da en UML: flecha continua para la llamada, flecha discontinua para la respuesta.

3.3. USE Modeling Tool

USE es una herramienta de modelado para la especificación de sistemas de información. Está basado en un subconjunto de UML (Database System Group Bremen University, 2007).

USE permite la definición y validación de sistemas mediante su especificación textual. Sus funcionalidades principales son:

- Especificación de clases con sus atributos, relaciones y funciones típicos de un Diagrama de Clases UML.
- Definición de máquinas de estado para las clases ya definidas.
- La especificación invariantes sobre el sistema en formato OCL.
- Creación de diagramas de secuencia para ejecuciones concretas del sistema, en formato SOIL.

El Object Constraint Language (OCL), mencionado en el párrafo anterior, es un lenguaje formal definido por OMG, que se emplea para describir expresiones en modelos de UML. Estas expresiones se utilizan para especificar condiciones que debe cumplir el sistema (Group, Object Constraint Language 2.4, 2003). En USE se puede emplear tanto para definir las precondiciones y postcondiciones de las funciones y para definir invariantes sobre el sistema.

Simple OCL-based Imperative Language (SOIL) es el lenguaje de programación imperativo que se definió para el empleo de USE (USE-OCL, 2014). Las sentencias de SOIL se introducen a través de la línea de comando que ejecuta la máquina virtual de Java de USE. Mediante estas sentencias imperativas, se especifican las llamadas que se realizan en los diagramas de secuencia.

Todas estas llamadas especificadas en formato SOIL son realizadas por un elemento denominado actor. Sin embargo, no se contempla que los objetos se llamen entre ellos sin la intervención del actor. Por otro lado, tampoco se pueden especificar diagramas de secuencia generales como los especificados en UML y MSC. Con lo que las sentencias condicionales e iterativas no se pueden emplear. Sólo pueden especificarse ejecuciones concretas de un diagrama de secuencia especificadas por el usuario.

Como podemos ver en la figura 3.3.1, todas las descripciones textuales son tomadas por la interfaz gráfica que muestra en forma de diagramas la información, después de validar todo lo definido en los archivos de texto.

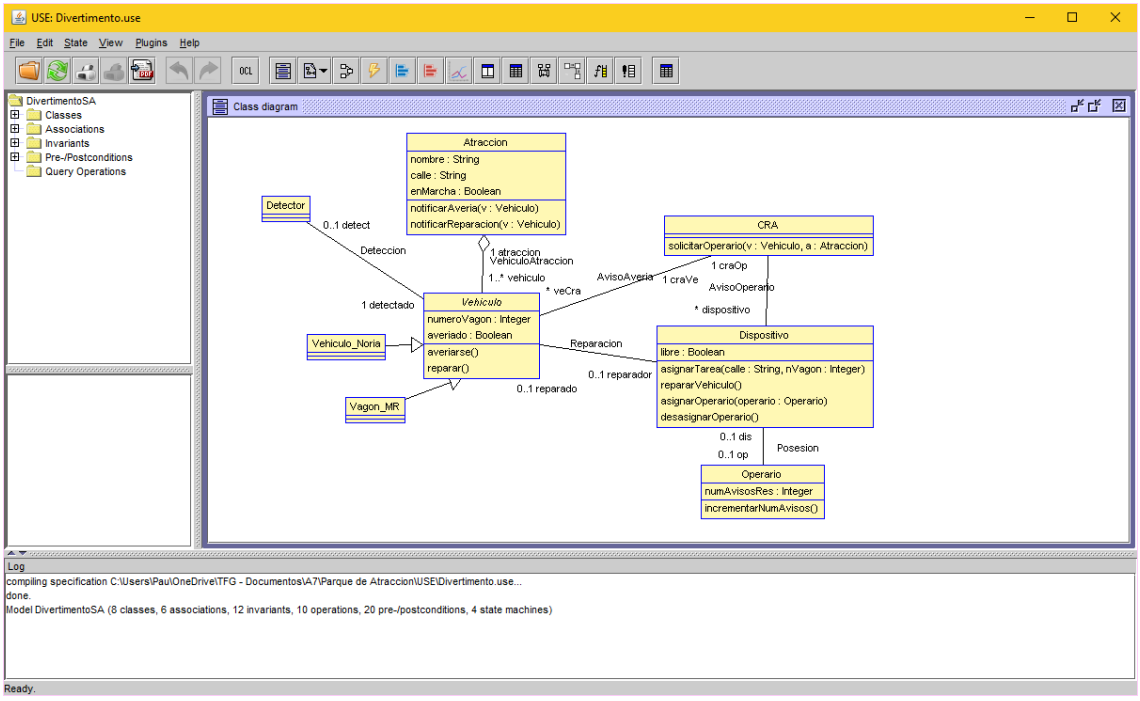


Figura 3.3.1. Ejemplo de Diagrama de Clase en USE

Con respecto a los diagramas de secuencia, también obtenemos una interpretación que da lugar a un diagrama concreto, como podemos ver en la figura 3.3.2.

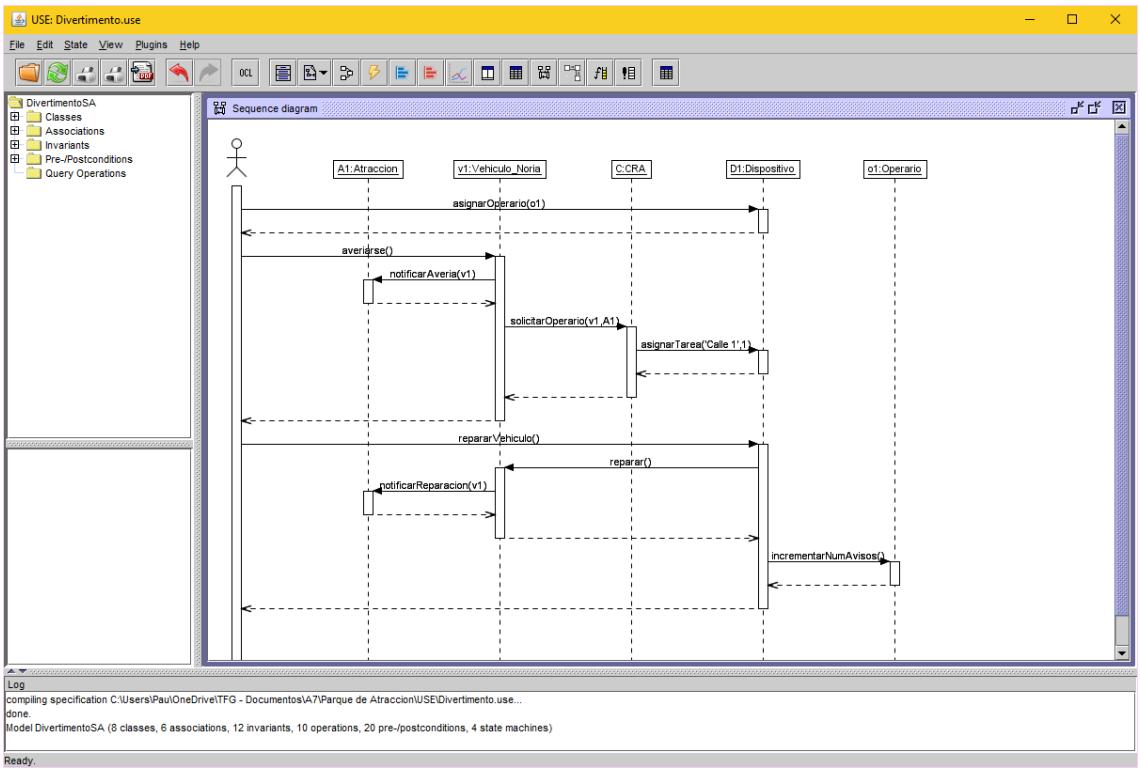


Figura 3.3.2. Ejemplo de diagrama de secuencia

```
1   !new CRA('C');
2   !new Dispositivo('D1');
3   !new Operario('o1');
4   !insert (C,D1) into AvisoOperario;
5   !D1.libre:=false;
6   ------------------------------------
7   !D1.asignarOperario(o1);
```

Figura 3.3.3. Ejemplo de archivo SOIL

En la figura 3.2.3. podemos ver un ejemplo de un archivo de SOIL:

- En las tres primeras líneas creamos instancias de 3 de las clases, el argumento entre paréntesis será el nombre de cada instancia.
- En la línea 4, especificamos una instancia de la relación entre ambos objetos. El nombre de la relación que se instancia es *AvisoOperario*
- En la línea 5, asignamos un valor a uno de los atributos de la instancia D1.
- En la línea 7, el actor llama a una función de D1 y le pasa como argumento el objeto o2.

3.5. Xtext

Xtext es un framework para el desarrollo de lenguajes de programación y de lenguajes específicos de dominio (Domain Specific Languages, DSL) (Xtext, 2019). Xtext fue lanzado en 2006 bajo el proyecto openArchitectureWare. A partir de 2008 y hasta la versión actual, ha sido desarrollado por la fundación Eclipse.

Este framework provee a los usuarios de una infraestructura completa para facilitar todos los pasos que van desde el análisis gramatical, hasta la compilación. Además, todo el sistema está incrustado en el editor de Eclipse. El usuario sólo debe limitarse a describir la gramática de acuerdo a las reglas de Xtext y el framework se encarga de prepararla para su uso posterior. Una vez definida la gramática, se puede ejecutar una instancia adicional de Eclipse, que evaluará los textos de acuerdo a las normas anteriormente establecidas. Cada vez que se introduce un texto, se evalúa si el contenido es correcto, y si lo es, genera una estructura arbórea con toda la información que haya incluido el usuario.

Esta estructura arbórea nace de la aplicación del Eclipse Modeling Framework (EMF). Este Framework provee herramientas y soporte en tiempo de ejecución para producir conjuntos de clases Java para el modelo definido (Eclipse, Eclipse Modeling Framework (EMF) - Eclipse, 2019). Estas clases serán las que se empleen combinadas con Xtend para definir la generación de archivos, de la que hablaremos en el siguiente apartado.

Xtext emplea los modelos EMF como representaciones en memoria para la información obtenida de los archivos de texto de entrada. Este grafo en memoria al que hemos llamado estructura arbórea es el denominado *abstract syntax tree (AST)* (Eclipse, Eclipse Modeling Framework (EMF) - Eclipse, 2019).

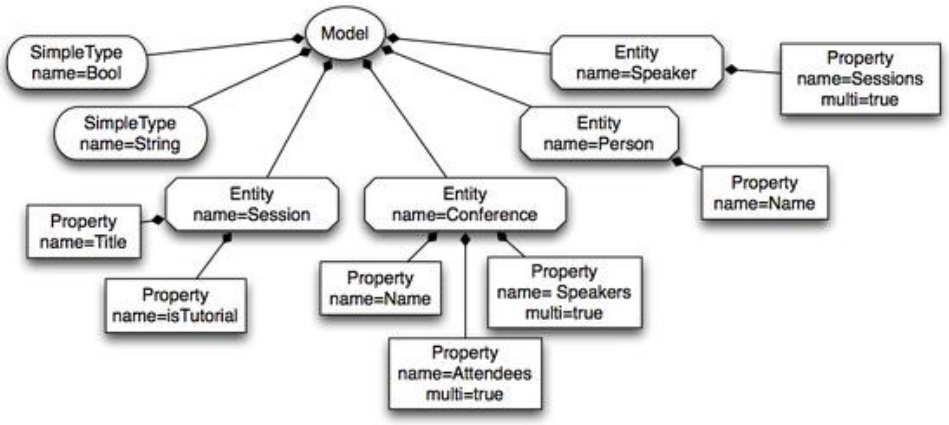


Figura 3.4.1. Ejemplo de abstract syntax tree, generado por Xtext (Eclipse, Eclipse Modeling Framework (EMF) - Eclipse, 2019)

Un ejemplo de esta estructura puede visualizarse en la figura 3.4.1. En ella, podemos ver que el modelo cuenta con una serie de entidades y tipos. Las entidades a su vez cuentan con propiedades que puede ser de tipo simple o de otras entidades.

La sintaxis de Xtext es muy similar a la empleada en la definición textual de las gramáticas en la mayoría de estándares, por lo que es muy intuitiva la definición de tipos. Se emplean expresiones regulares para establecer la repetición de los elementos.

3.6. Xtend

Xtend es un dialecto de Java, que cuenta con numerosas mejoras en algunos aspectos como la inferencia de tipos o expresiones específicas para la creación de *templates* (Xtend, 2019). Xtend se creó en sus inicios como una adición necesaria para el empleo de Xtext, pero actualmente es un proyecto independiente de la fundación Eclipse.

En el desarrollo de lenguajes, Xtend permite al usuario recorrer la estructura arbórea generada por Xtext y definir las funciones que generarán los archivos, producto final del proceso. Su funcionalidad para la creación de *templates*, permite definir de forma muy sencilla la estructura de los archivos que se generarán, permitiendo emplear la información tomada del árbol de forma muy sencilla. En el interior de la definición de

los *templates* se pueden emplear sentencias iterativas y condicionales, como podemos visualizar en la figura 3.5.1.

```
def compile(Entity e) '''
    «IF e.eContainer.fullyQualifiedName !== null»
        package «e.eContainer.fullyQualifiedName»;
    «ENDIF»

    public class «e.name» «IF e.superType !== null
            »extends «e.superType.fullyQualifiedName» «ENDIF»{
        «FOR f:e.features»
            «f.compile»
        «ENDFOR»
    }
'''
```

Figura 3.5.1. Ejemplo de función empleando los templates (Eclipse-Xtext, 2019)

4

Estructura e implementación

En este capítulo nos centraremos en el desarrollo de la herramienta, destacando sus aspectos más esenciales. Dividiremos el capítulo en dos partes, coincidiendo con las dos unidades fundamentales en las que se divide la herramienta: la gramática y el generador de código.

En el primer apartado analizaremos la estructura completa de la aplicación, dando una vista general del funcionamiento. En el siguiente apartado, nos centraremos en los elementos de la gramática elegidos de la recomendación y en cómo sería la estructura arbórea general que se deriva de la definición de la gramática. En el último apartado nos centraremos en los distintos enfoques y algoritmos que han sido necesarios para conseguir la implementación completa de la herramienta.

4.1. Flujo de la aplicación

En este apartado, antes de adentrarnos en los entresijos de la aplicación, expondremos el funcionamiento dependiendo de las entradas y mencionaremos las distintas validaciones que se llevan a cabo.

Como podemos ver representado en la figura 4.1.1., la entrada de la aplicación es un único archivo MSC. Este archivo deberá ser añadido a un proyecto creado en el interior del entorno que Xtext proporciona y en el que se han debido de cargar las reglas de la gramática anteriormente definida. Este entorno validará que el texto esté expresado de acuerdo a la gramática: si el archivo es incorrecto, el editor subrayará las líneas donde se encuentren los errores sintácticos y no se realizarán más acciones; si el archivo es correcto, pasaremos a la siguiente fase.

Una vez el archivo ha pasado la validación sintáctica, se procesará la información y ésta pasará al algoritmo de generación de código, desarrollado en Xtend. Si el archivo no cumple las normas impuestas en el algoritmo, creará un archivo denominado FORMAT_ERROR que mostrará hasta donde ha procesado el algoritmo antes de encontrarse el error.
Si el archivo es correcto, se generarán los archivos de salida. Hay dos tipos de archivos de salida:

- Los archivos en formato SOIL los cuales estarán numerados. En cada uno de ellos encontraremos una ejecución posible del diagrama introducido.
- Un archivo en formato USE. Este archivo incluirá las clases y relaciones necesarias para emplear los archivos SOIL anteriormente definidos. El contenido de este archivo deberá colocarse al final del fichero USE para el que se quieran emplear las secuencias SOIL.

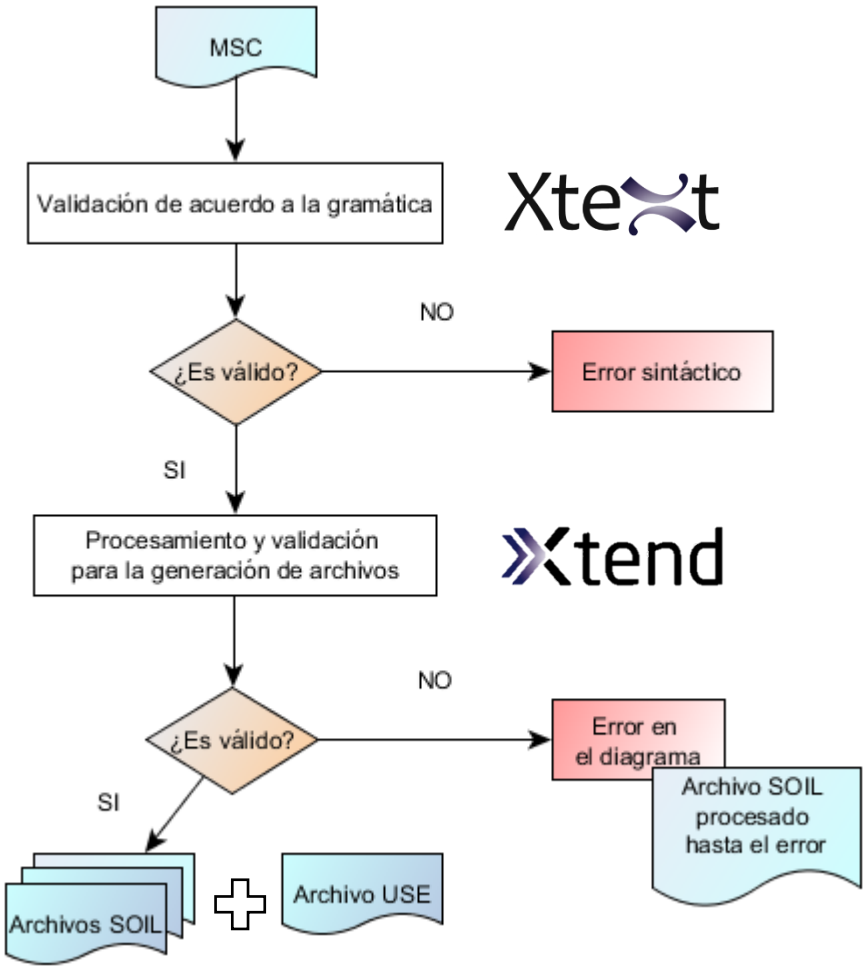


Figura 4.1.1. Flujo de datos de la aplicación

4.2. Definición de la gramática

4.2.1. Estructura general

En las siguientes secciones analizaremos las distintas partes de la gramática que seleccionado y definido para el proyecto. En la figura 4.2.1. podemos ver la estructura arbórea general que genera Xtext a partir de las reglas definidas.

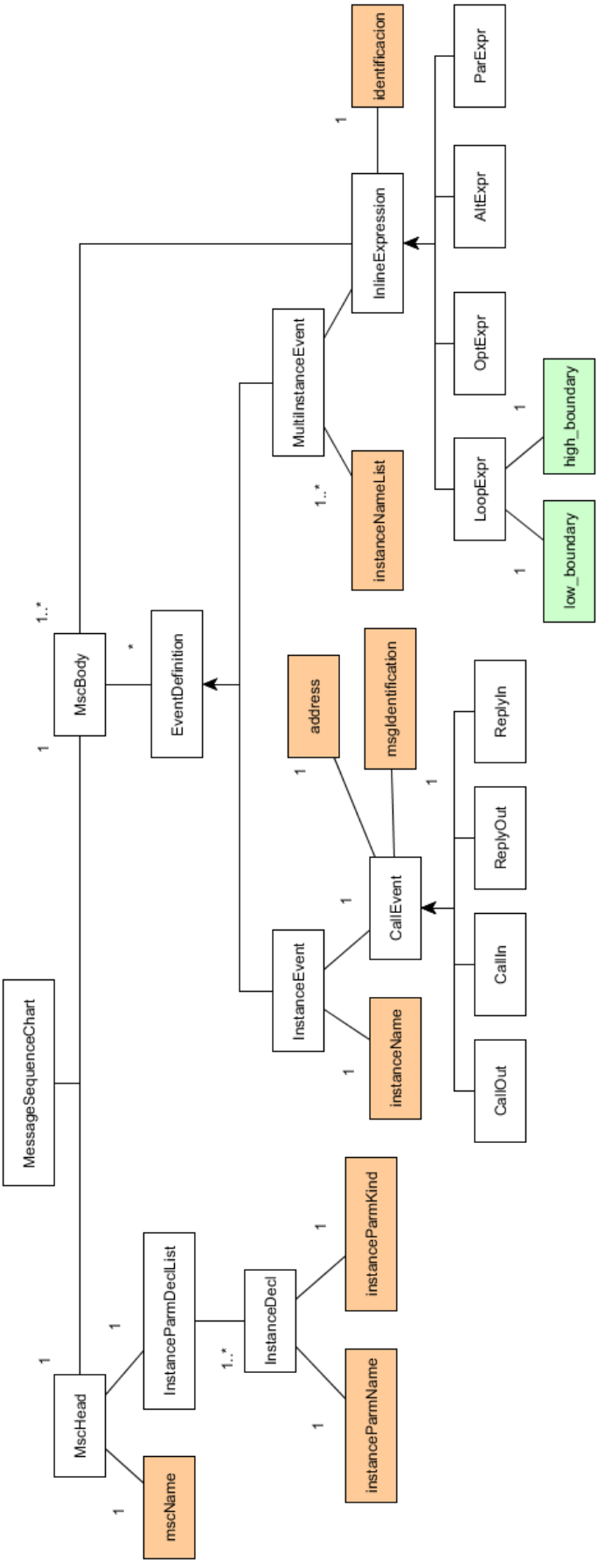


Figura 4.2.1.1. Diagrama de la estructura completa de la gramática

Leyenda del diagrama de la figura 4.2.1.

- **Multiplicidades:** los extremos no especificados tienen cardinalidad 1.
- **Recuadros con la inicial en mayúscula:** tipos complejos definidos en la gramática.
- **Recuadros con la inicial en minúscula:** cadenas de texto a excepción de los que incluyen *boundary* en su nombre, que son valores enteros.

En el apartado 3.2., hicimos referencia a todas las normas gramaticales extraídas de la recomendación que se han empleado en este proyecto y que están representadas en el esquema anterior. En este apartado no volveremos a insistir en su sintaxis, sino que relacionaremos lo expuesto en la tercera sección con lo plasmado en esta para permitir al lector asimilar los conceptos del diagrama para posteriormente comprender mejor el algoritmo.

La definición completa de la gramática puede encontrarse en el Anexo C en caso de que el lector tenga interés en ella.

4.2.2. MscHead - Declaración de las instancias

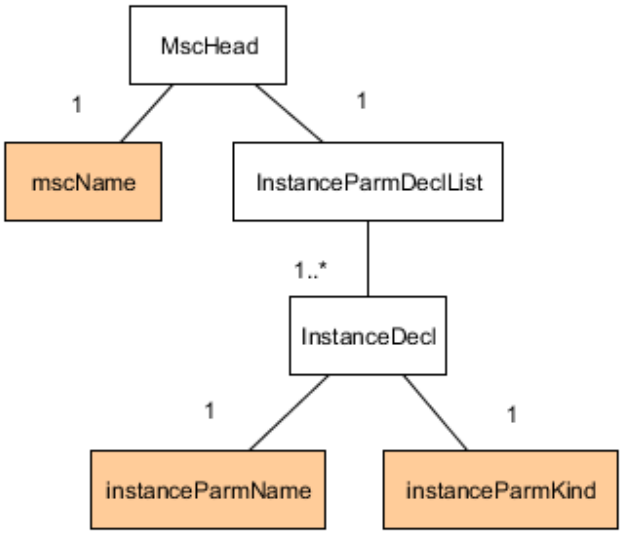


Figura 4.2.2.1. Regla MscHead

La parte izquierda del diagrama, figura 4.2.2.1., hace referencia a las dos primeras líneas que se definen en un MSC. En primer lugar, el elemento *mscName* es un identificador que lleva el MSC, aunque no tiene ningún propósito en la generación de archivos, es una mera referencia identificativa. En la segunda rama nos encontramos el módulo para la declaración de las variables que tomarán parte en el diagrama de secuencia y cuyo formato ya fue especificado en el apartado 3.2.. Los dos elementos *instanceParmName* e *instanceParmKind*, se emplean para almacenar el nombre y el tipo de las variables intervinientes respectivamente.

4.2.3. MscBody - Interacciones entre instancias

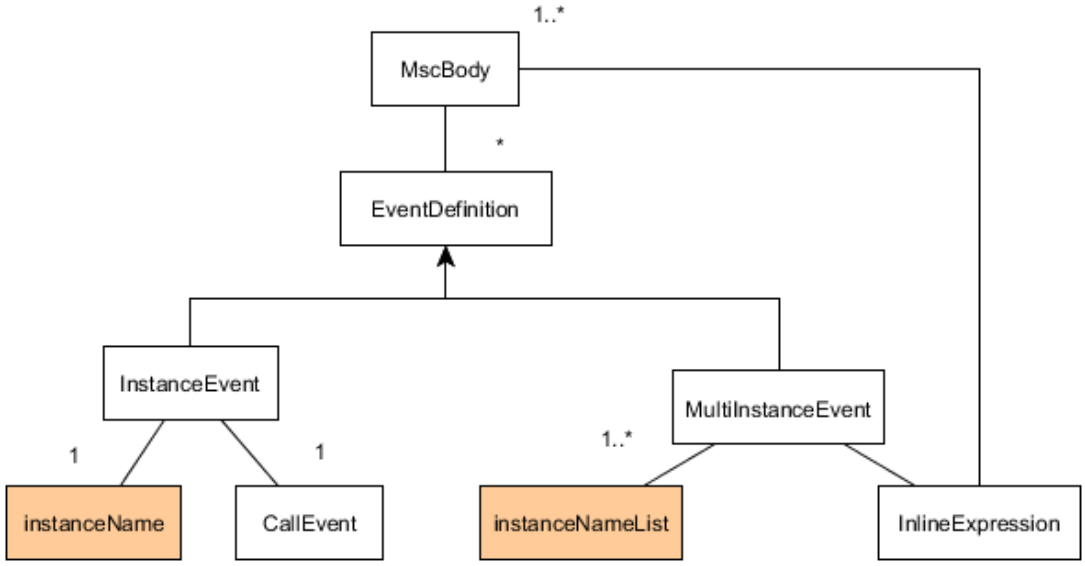


Figura 4.2.3.1. Regla parcial del MscBody

La parte derecha del diagrama, figura 4.2.3.1., nos muestra la pieza angular de todo el sistema de interacción entre instancias: el *MscBody*. Esta es la estructura que almacena todas las interacciones entre instancias.

El *MscBody* cuenta con una lista de un número indefinido de elementos del tipo abstracto *EventDefinition* que se concreta en dos tipos:

- ***InstanceEvent*:** interacción simple en la que la acción la ejecuta una variable. El nombre de la variable se almacena en el campo *instanceName*. La interacción concreta es del tipo abstracto *CallEvent*, el cual se concreta en todas las llamadas síncronas que son hechas por las distintas instancias con el formato *call-receive* y *replyout-replyin*, definido en el apartado 3.2.

- ***MultiInstanceEvent*:** interacción compleja en la que intervienen dos o más variables. En la variable *instanceNameList* se almacenan los nombres de las variables que intervendrán. El tipo abstracto *InlineExpression* se concreta en todas las estructuras complejas expuestas en el apartado 3.2.: *alt, opt, par* y *loop*.

Tal como muestra el diagrama, las *InlineExpression* pueden contar con uno o varios *MscBody* en su interior. De esta forma, todas las expresiones complejas pueden anidarse o combinarse unas con otras. Esta característica será la más problemática y la que nos obligará a adaptar varias veces el algoritmo para la generación de código automática. Un ejemplo de la estructura contenida en un mscBody sería lo representado en la figura 4.2.3.2..

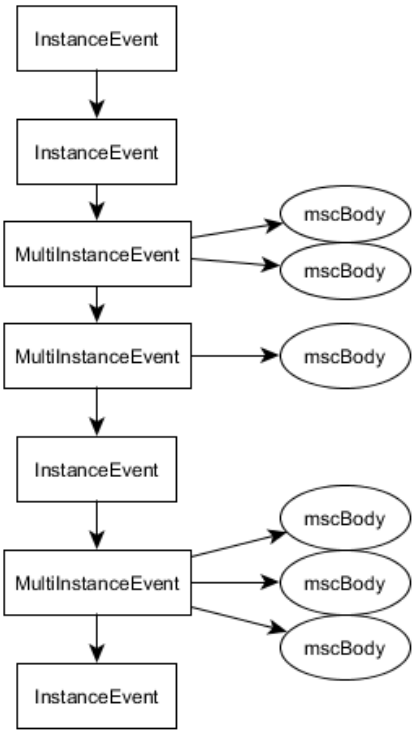


Figura 4.2.3.2. ejemplo de una instancia de MscBody.

4.3. Definición del algoritmo

En los próximos apartados describiremos las distintas versiones del algoritmo implementado y los problemas a resolver. Durante el desarrollo han sido necesarias diversas modificaciones para optimizar el rendimiento de la aplicación y asegurar un tiempo de respuesta razonable.

Finalmente, nombraremos y explicaremos algunas clases auxiliares y algoritmos que se han empleado en el desarrollo.

4.3.1. El problema del actor

Como se expuso en el apartado 3.4., USE cuenta con muchas limitaciones en el campo de la definición de diagramas de secuencia generales. Uno de los principales problemas es que todas las instrucciones que se ejecutan en un diagrama de secuencia deben están

llamadas por un elemento denominado Actor. USE no contempla que el objeto pueda tomar iniciativa y llamar a una función de otro objeto por sí mismo. Las acciones siempre deben estar motivadas por una llamada anterior hecha por el actor.

En este proyecto se pretende que los diagramas de secuencia no tengan esa limitación y que el usuario pueda definir los modelos y diagramas sin tener en cuenta este aspecto de USE.

Para paliar este problema, crearemos una clase USE adicional que se denominará *Choreographer* (en español, coreógrafo), esta clase será la que se encargue de decirles a las distintas clases que deben ejecutar una acción. Adicionalmente, se añadirán clases que extiendan a las que tomen parte en el diagrama de clases y que tengan las funciones que el coreógrafo usará para hacerles saber que es su turno, y que deberán ejecutar la función que se había definido en el diagrama de secuencia. Para permitir la comunicación entre las clases adicionales y el coreógrafo, se crearán relaciones ente ellos.

Esta clase se visualizará en USE como una nueva *Lifeline* que al ser ocultada, nos dejará ver el diagrama de secuencia tal y como lo sería sin la limitación que impone USE. En la figura 4.3.1.1. podemos visualizar el resultado al aplicar el coreógrafo y en la 4.3.1.2 podemos ver el resultado al ocultarlo. Esta ejecución es la salida del ejemplo que pusimos en la figura 3.2.4.

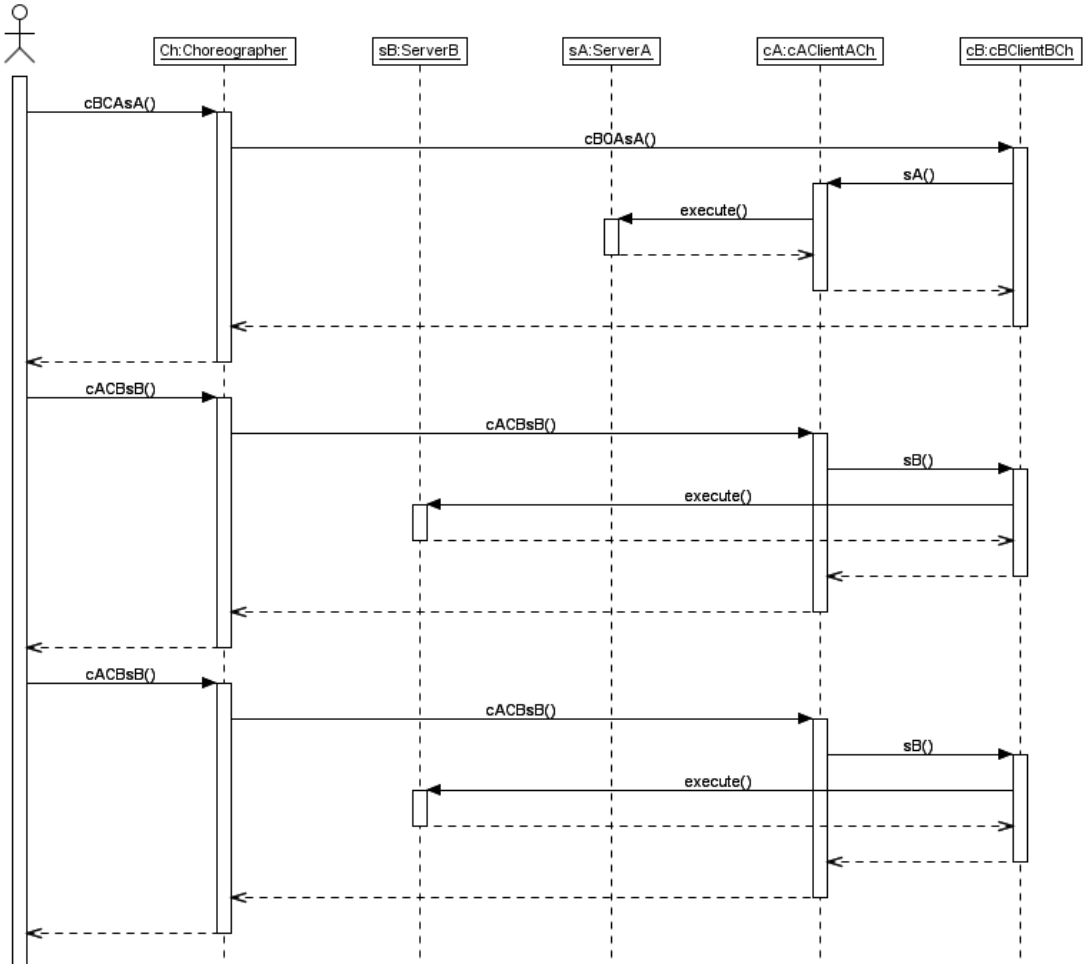


Figura 4.3.1.1. Ejemplo de diagrama de secuencia de USE mostrando el coreógrafo.

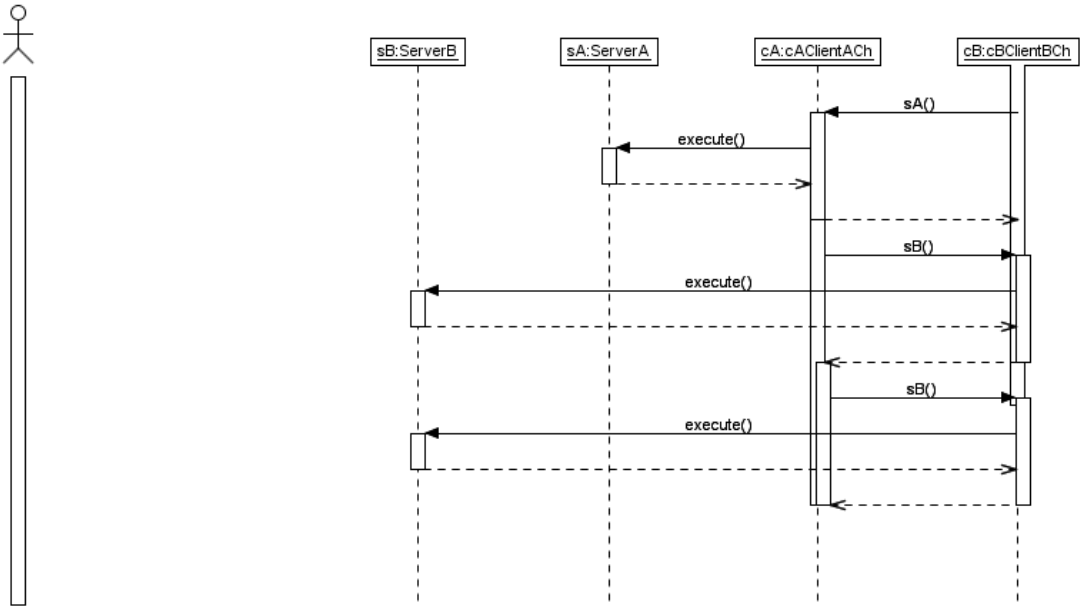


Figura 4.3.1.2. Ejemplo de diagrama de secuencia de USE ocultando el coreógrafo.

```
!new ServerA ('sA');
!new ServerB ('sB');
!new Choreographer('Ch');
!new cAClientACh ('cA');
!new cBClientBCh ('cB');
!insert (cA, Ch) into RelChClientAcA;
!insert (cB, cA) into RelChExtClientAcAcB;
!insert (cB, Ch) into RelChClientBcB;
!insert (cA, cB) into RelChExtClientBcBcA;
-- EXTRA CONNECTIONS HERE
!insert (sA, sB) into RelsAsB;
!insert (cA, sA) into RelCaSa;
!insert (cA, sB) into RelCaSb;
!insert (cB, sA) into RelCbSa;
!insert (cB, sB) into RelCbSb;
--
!Ch.cBCAsA();
!Ch.cACBsB();
!Ch.cACBsB();
```

Figura 4.3.1.3. Ejemplo de secuencia de SOIL salida del algoritmo.

En la salida de SOIL que obtenemos y de la que podemos visualizar un ejemplo en la figura 4.3.1.3., hay un espacio que dejamos en blanco justo debajo de *EXTRA CONNECTIONS HERE.* Para que el usuario pueda ejecutar el script correctamente, debe inicializar ahí cualquier otra relación o variable que sea necesaria para hacer funcionar el sistema. Las únicas relaciones que se inicializan son las relacionadas con el coreógrafo.

4.3.2. Implementación básica

El primer enfoque planteado para solucionar el problema de la generación de archivos fue implementar un algoritmo recursivo que recorriera todos y cada uno de los posibles caminos de raíz a hojas. La recursión tenía dos comportamientos principales, dependiendo de si es una instrucción simple o una compleja.

En caso de encontrarse con una instrucción simple, *InstanceEvent,* el algoritmo se encarga de procesarla y traducirla a la cadena de caracteres que se imprimirá en el archivo de salida. En cada recursión se pasan los siguientes elementos:

- El *mscBody* principal eliminando la sentencia que ya haya sido procesada.
- La cadena de caracteres que contiene todas las instrucciones ya procesadas.
- El número que será el nombre del siguiente archivo a generar.

La recursión continúa hasta que no queda ninguna sentencia en el mscBody. Entonces, la variable del tipo cadena de caracteres se plasma en un archivo de texto, constituyendo el archivo de salida.

Como en este proyecto sólo consideramos llamadas síncronas, el algoritmo busca hasta encontrar una instrucción *call* en la que intervienen un emisor y un receptor. Una vez encontrada, el algoritmo ignorará las instrucciones siguientes hasta que encuentre una instrucción *replyin* en la que el emisor y el receptor coincidan. Entonces, procesa el mensaje que iba en el call y lo añade a la cadena de caracteres a plasmar en el archivo de salida. El resto de instrucciones son ignoradas porque se consideran que pertenecen a la ejecución interna de la primera instrucción. Las instrucciones *receive* y *replyout* no se tienen en cuenta en la validación.

En caso de encontrarse con una instrucción compleja, *MultiInstanceEvent,* el comportamiento dependerá del tipo de la misma. La regla general es llamar a tantas recursiones como posibilidades dé lugar empleo de la estructura compleja. En cada una de estas recursiones, la estructura del mscBody es clonada para evitar modificaciones entre las distintas líneas de recursión al hacer las eliminaciones.

Ahora nos dispondremos a analizar el procedimiento para cada una de las estructuras complejas.

❖ **Opt:** se extrae el *mscBody*, en el diagrama 4.3.2.1. optMscBody, de su interior y se lanzan dos recursiones, una con el *mscBody* concatenado al *mscBody* principal y otra sin él.

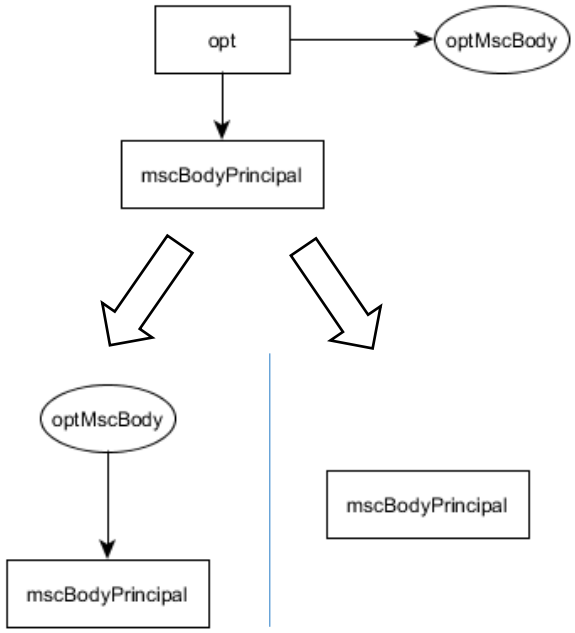


Figura 4.3.2.1. Aplicación del algoritmo al opt. Los elementos de la parte inferior de la figura son los que serán el mscBody entrante en cada recursión.

❖ **Alt:** se lanzan n recursiones y cada una contará con una copia del mscBody principal concatenado con cada uno de los *mscBody* contenidos en el alt, altMscBody<número> en el diagrama 4.3.2.2..

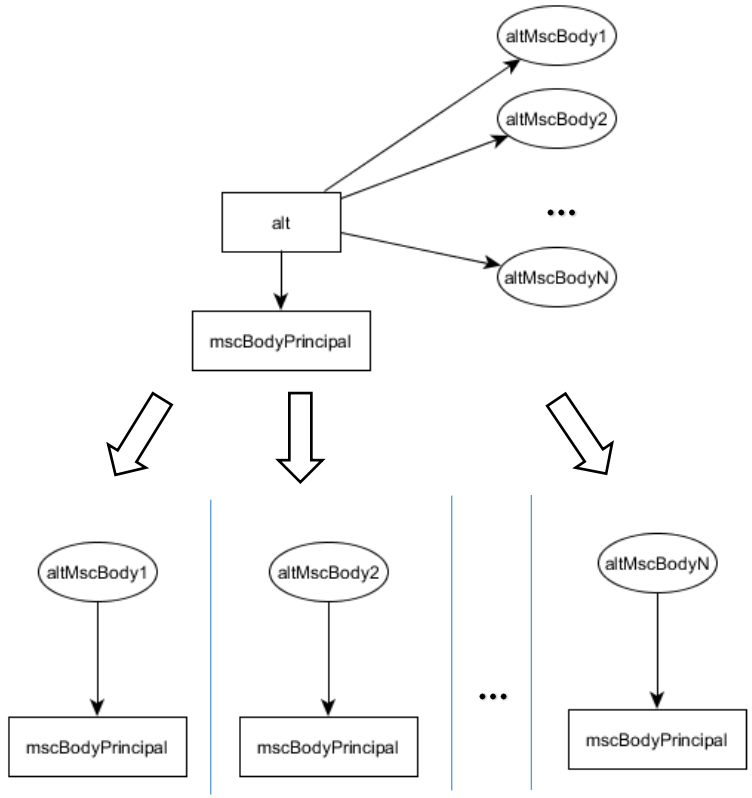


Figura 4.3.2.2. Aplicación del algoritmo al alt. Los elementos de la parte inferior de la figura son los que serán el mscBody entrante en cada recursión.

- **Par:** se empleará el Algoritmo de Heap, que se explicará en apartados posteriores, para obtener todas las permutaciones posibles de los *mscBody* contenidos en la instrucción, parMscBody<número> en el diagrama 4.3.2.3.. Por cada permutación se compondrá un *mscBody* con una de las permutaciones concatenada al *mscBody* principal. Cada una de estas posibilidades es una entrada a una nueva recursión.

Figura 4.3.2.3. Aplicación del algoritmo al par. Los elementos de la parte inferior de la figura son los que serán el mscBody entrante en cada recursión.

- **Loop:** se extrae el mscBody de su interior, y se hacen tantas concatenaciones del mscBody del loop, loopMscBody en el diagrama 4.3.2.4., como sea la diferencia entre el *high_boundary* y el *low_boundary*. En cada nuevo mscBody resultante se encuentra el *mscBody* del *loop* repetido un cierto número de veces. Este número de veces es igual a todos y cada uno de los valores entre *low_boundary* y *high_boundary*. A cada una de estas posibilidades se le concatena el *mscBody* principal. Cada una de estas posibilidades es una entrada a una nueva recursión.

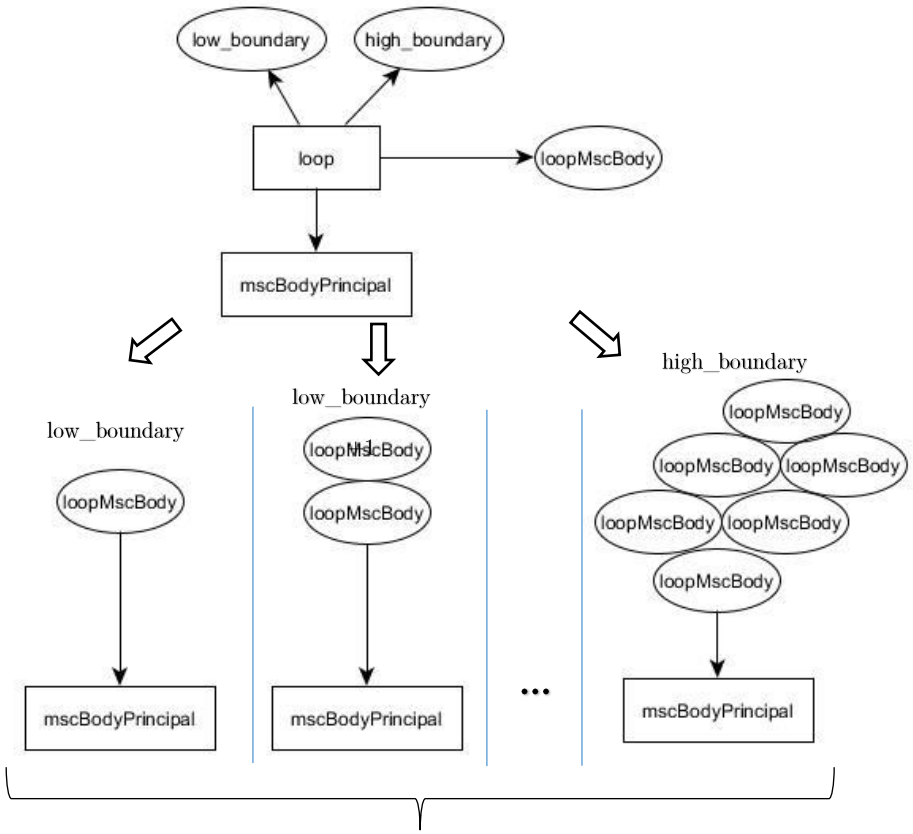


Figura 4.3.2.4. Aplicación del algoritmo al loop con 1 como valor de low_boundary. Los elementos de la parte inferior de la figura son los que serán el mscBody entrante en cada recursión.

Esta implementación funcionaba perfectamente para todos los casos simples y anidaciones de elementos complejos. Sin embargo, cuando se introducía cualquier elemento complejo en un bucle, el entorno empezaba a sufrir una sobrecarga de memoria, dando igual lo simple que fuera el anidamiento. Tras cierta investigación, nos percatamos de que el problema residía en la optimización de la estructura arbórea que proveía el entorno. Al hacer repetidas concatenaciones y duplicaciones necesarias para el procesamiento de los bucles, el entorno no podía soportarlo.

4.3.3. Procesamiento de los bucles a partir de archivos auxiliares

Para evitar la sobrecarga generada por el anidamiento de estructuras complejas en los bucles, se modificó el algoritmo para el procesamiento concreto de los bucles y se creó

una nueva función. Esta es también recursiva y tiene como objetivo procesar el interior del bucle por separado y plasmar en archivos auxiliares las distintas posibilidades que dan lugar a partir del mscBody del interior del bucle.

Una vez se tiene procesado el interior del bucle como un MSC independiente e introducido en archivos auxiliares sus posibles ejecuciones, debemos encargarnos de generar las repeticiones del bucle. Para ello, tenemos que tener en cuenta que en cada vuelta del bucle pueden tomarse distintas decisiones.

Esta circunstancia nos lleva a darnos cuenta de que existen ejecuciones en las que siempre se darán las mismas circunstancias, ejecuciones en las que en cada vuelta se da una distinta o ejecuciones en las que hay vueltas del bucle que se repiten con respecto a vueltas anteriores.

A través de esto, analizamos el tipo de posibilidades que pueden darse y concluimos que estas coinciden con el número de variaciones con repetición de n elementos seleccionándolos en grupos de k. En nuestro caso n sería el número de archivos seleccionados y k sería el número de vueltas que da el bucle en esa ejecución.

Para computar todas estas posibilidades, se optó por la creación de una clase auxiliar, que es capaz de generar números en los que se establecía el módulo y el número de cifras. Las numeraciones tienen la característica de que dan lugar a la situación de combinaciones que buscamos para el algoritmo. Por ejemplo, para 3 elementos seleccionados en grupos de 2:

[0, 0]

[0, 1]

[0, 2]

[1, 0]

[1, 1]

[1, 2]

[2, 0]

[2, 1]

[2, 2]

Generando todos los números posibles para cada configuración, damos con todas las combinaciones. Estos números son almacenados en arrays para permitir que cada posición pueda tener un número ilimitado de cifras y que sean fácilmente procesables.

Los archivos de bucle generados tienen un nombre específico loop_<profundidad>_<número de archivo>. Como se permite el anidamiento de bucles en otras estructuras y en otros bucles, necesitamos saber a qué bucle pertenecen los archivos. Para ello, contamos con un valor de profundidad, que se le especifica a los archivos. De esta forma, conseguimos identificarlos al salir de la recursión para emplearlos.

Volviendo a la definición concreta del algoritmo, ahora nos disponemos a añadir todo lo procesado en los archivos auxiliares a la cadena de caracteres de la recursión

principal. Una vez se han generado todos los archivos con todas las posibilidades del interior del bucle, salimos de la recursión del algoritmo adicional y volvemos al inicial. Entonces, sabiendo cuántos archivos tenemos y cuántas vueltas del bucle, iremos generando todos los números correspondientes. Una vez se obtiene el número, se irán recorriendo todas sus posiciones y añadiendo el contenido del archivo cuyo número de archivo coincida con el valor de la posición del número generado a la cadena de caracteres resultante que será entrada de una nueva recursión. De esta forma continua el procesamiento como se expuso en el apartado anterior, pero con el bucle procesado.

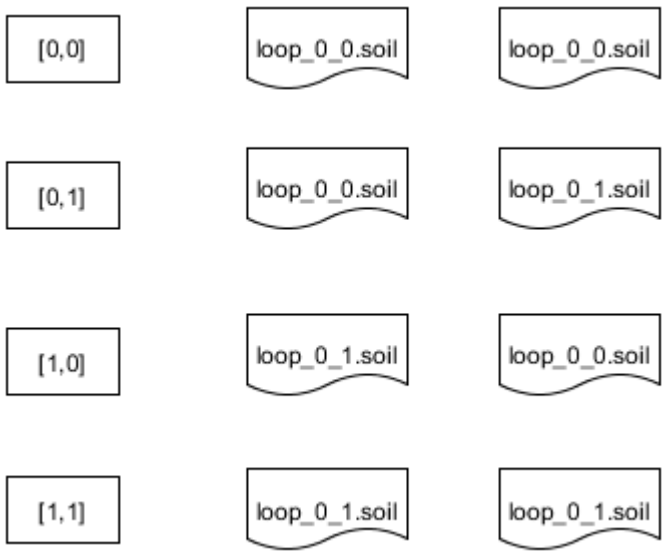


Figura 4.3.3.1. Ejemplo de selección de archivos del algoritmo de los bucles para un bucle con 2 vueltas y 2 archivos.

En la figura 4.3.3.1. podemos ver un ejemplo para un bucle en el que se dan 2 vueltas y un bucle cuyo contenido a dado lugar a dos posibilidades. Todas las probabilidades posibles en este caso son las expuestas en la figura. Para cada posibilidad se tomará el contenido de los archivos y se añadirá a la cadena de texto de resultado que entrará a una nueva recursión. Se generará una nueva recursión por cada posibilidad.

4.4. El algoritmo de Heap

El algoritmo de Heap, fue propuesto por B. R. Heap en 1936. Este algoritmo genera todas las permutaciones posibles de N objetos, haciendo intercambios entre dos elementos pertenecientes al conjunto. Una permutación difiere de la siguiente en la posición de dos objetos que estarán intercambiados entre sí (Heap, 1936).

Este algoritmo tal y como lo hemos adaptado para el proyecto, funciona de la siguiente forma. El algoritmo cuenta con un contador que empieza en el número total de elementos a permutar que denominaremos n. Este contador decrementa en cada una de las llamadas y el algoritmo termina cuando este contador alcanza el 0. Los intercambios se hacen de acuerdo a la paridad de este valor. Dentro de cada llamada habrá un bucle que generará un número de permutaciones igual al factorial del contador mediante el cambio de pares de elementos. Dependiendo de si es par o no, haremos el intercambio con el elemento en la primera posición o con el número correspondiente a la vuelta del bucle.

El siguiente gráfico extraído del artículo de Wikipedia correspondiente al algoritmo nos permite ver claramente cómo funciona el intercambio de elementos para conseguir las permutaciones.

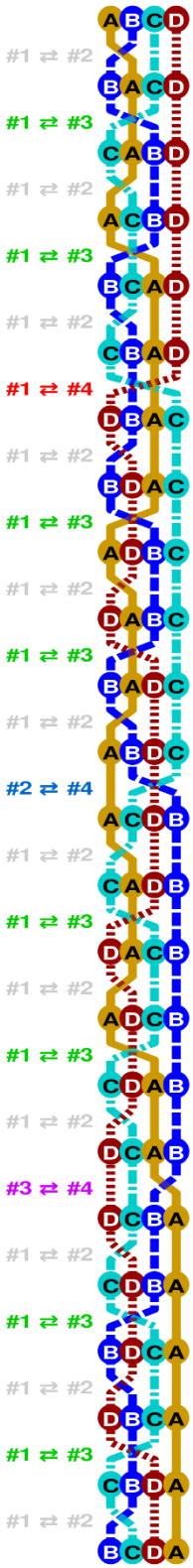


Figura 4.4.1. Funcionamiento del algoritmo de Heap con 4 elementos. (Wikipedia, 2019)

5

Validación y pruebas

En este apartado expondremos la estrategia empleada para efectuar las pruebas que validan el funcionamiento del algoritmo. En un primer apartado hablaremos de la problemática que hemos encontrado a la hora de decidir el método de validación. En un segundo apartado enumeraremos los casos de prueba seleccionados en cada una de las iteraciones.

La gramática no se sometió a pruebas de validación o depuración porque fue una selección directa de la recomendación Z.120.

5.1. Problemática

En primer lugar, por la dificultad que añadida suponía el introducir una herramienta para la ejecución automática de casos de prueba en un entorno de desarrollo como el del proyecto, hemos ejecutado cada uno de los chequeos manualmente.

En segundo lugar, por la naturaleza del algoritmo, es prácticamente imposible desarrollar pruebas que constituyan todas las posibles entradas y salidas del algoritmo. Esto es debido a que los MSC pueden tener una longitud indeterminada y el número

de MSCs que se pueden formar de acuerdo a las reglas de la gramática también son prácticamente infinitos.

Por estas razones, para la comprobación del correcto funcionamiento del algoritmo, hemos empleado pruebas unitarias para una serie de casos de uso seleccionados. Como se ha empleado una metodología de desarrollo incremental e iterativa, se han ido desarrollando casos de prueba para cada una de las nuevas funcionalidades conforme estas se desarrollaban.

En cada una de las iteraciones se han desarrollado nuevas pruebas para validar lo desarrollado en la nueva iteración, y se han ejecutado las desarrolladas anteriormente para comprobar que no se ha realizado ninguna modificación que afectase a la parte desarrollada en iteraciones anteriores.

5.2. Pruebas realizadas

En este apartado mencionaremos las distintas pruebas realizadas en cada una de las iteraciones. Las pruebas tienen como finalidad cubrir los casos de uso más básicos, incluyendo errores. En todas las pruebas estimamos el número de archivos esperado y comprobamos que el contenido de los mismos sea el correcto.

Para todas las pruebas se ha empleado un modelo cliente-servidor. En este modelo existen dos clientes: ClienteA y ClienteB; y dos servidores: ServerA y ServerB. Ambos clientes pueden hacer llamadas a ambos servidores y el ServerA puede efectuar llamadas al ServerB.

5.2.1. Primera iteración - Sentencias simples

En la primera iteración seleccionamos toda la parte de la gramática referente a los mensajes simples: las llamadas síncronas. El sistema era capaz de tomar un MSC con llamadas síncronas y obtener los ficheros de salida correspondientes, además de detectar si el usuario había incluido la llamada replyin correspondiente por cada call que se procese.

Los casos de prueba que se establecieron fueron los siguientes:

- **Secuencia de interacciones simples con una instancia de cada clase:** en este caso de prueba se crea una instancia por clase y se ejecuta una llamada de cada una de ellas. Con esta prueba comprobamos si el funcionamiento básico del algoritmo es correcto.
- **Secuencia de interacciones simples con dos instancias de cada clase:** en este caso de prueba se crean dos instancias por clase y se ejecuta al menos una llamada de cada una de ellas. Con esta prueba comprobamos si en el archivo USE ocurren duplicaciones en la creación de clases u otros eventos relacionados.
- **Secuencia de interacciones simples faltando un *replyin*:** en este caso de prueba se crean una serie de instancias y se introduce un *call* que carece de *replyin*. La salida debe ser la creación de un archivo *FORMAT_ERROR*.
- **Secuencia de interacciones simples sin *call*:** en este caso de prueba se crean una serie de instancias y no se introduce ningún *call*. La salida de este caso de prueba debe ser la creación de un archivo *FORMAT_ERROR*.

5.2.2. Segunda iteración - Sentencia compleja *opt*

En la segunda iteración se añadió la primera sentencia compleja, *opt.* El sistema era capaz de tomar un MSC con las mismas características que en la iteración anterior, añadiendo la capacidad de procesar sentencias opt.

Los casos de prueba seleccionados para esta iteración fueron los siguientes:

- **Secuencia de interacciones simples con 1 sentencia *opt* situada entre ellas:** en este caso de prueba se crean una serie de instancias y se introduce una sentencia *opt.* La salida de este caso de prueba debe ser la creación de 2 archivos, uno con el contenido del *mscBody* del opt, y otro sin él.
- **Secuencia de interacciones simples con más de una sentencia *opt* situada entre ellas:** en este caso de prueba se crean una serie de instancias y se introducen varias sentencias opt, sin anidaciones entre ellas. La salida esperada son un número de archivos igual al cuadrado del número de sentencias *opt* introducidas. No puede haber dos archivos iguales y deben darse todas las combinaciones de *aparición-desaparición* del *mscBody* de cada sentencia *opt.*
- **Secuencia de interacciones simples con una sentencia *opt* con otra sentencia *opt* anidada entre ellas:** en este caso de prueba se crean una serie de instancias y se introduce una sentencia opt, con otra sentencia *opt* anidada. La salida esperada son 3 archivos: uno en el que aparezcan dos los *mscBody,* uno en el que no aparezca el *mscBody* anidado, y uno en el que no aparezca ninguno de los dos.
- **Secuencia de interacciones simples con una sentencia *opt* con dos niveles de anidación entre ellas:** en este caso se crean una serie de instancias y se introduce una sentencia *opt* con una sentencia *opt* en su interior, que a su vez tiene otra sentencia *opt* dentro. La salida esperada son 4 archivos: uno en el

que aparezcan todos los *mscBody,* uno en el que no aparezca el de la última anidación pero los demás sí, uno en el sólo aparezca el *mscBody* más exterior, y uno en el que no aparezca ninguno de ellos.

5.2.3. Tercera iteración - Sentencia compleja *alt*

En esta iteración añadimos incrementalmente la sentencia compleja *alt.*

Los casos de prueba seleccionados para esta iteración fueron los siguientes:

- **Secuencia de interacciones simples con una sentencia *alt* de dos opciones situada entre ellas:** en este caso de prueba se crean una serie de instancias y se introduce una sentencia *alt* entre algunas interacciones síncronas. La salida esperada incluye dos archivos: uno debe incluir una de las opciones y el otro, la otra.
- **Secuencia de interacciones simples con una sentencia *alt* de más de dos opciones situada entre ellas:** este caso es igual al anterior, añadiendo una opción más al *alt.* Se han de generar tantos archivos como opciones tuviera el *alt.* En cada archivo debe encontrarse una de las opciones. Este caso de prueba ha sido probado en cada iteración con un número limitado de opciones aleatorias.
- **Secuencia de interacciones simples con más de una sentencias *alt* de más de dos opciones situadas entre ellas:** este caso amplía el anterior añadiendo una sentencia un número indeterminado de sentencias *alt* al final. Se deberán generar tantos archivos como el valor de multiplicar el número de opciones de cada uno de los *alt.* De esta forma para el caso de dos *alt* en el que 2 de ellos tengan 3 opciones y uno de ellos tenga 2, el resultado será $3 \cdot 3 \cdot 2 = 18$ archivos generados. Los archivos tendrán todas las elecciones posibles teniendo en cuenta las opciones disponibles. Este caso de prueba ha sido probado en cada iteración con un número limitado de valores aleatorios con respecto al número de sentencias y el número de opciones para cada una de ellas.

- **Secuencia de interacciones simples con una sentencia *alt* que cuenta con otra sentencia *alt* anidada en cada una de sus opciones:** en este caso se introduce el anidamiento de la instrucción. Para este caso serán sentencias *alt* con dos opciones y en cada opción habrá otro *alt* de dos opciones. La salida esperada es de 4 archivos: en los dos primer archivos estarán cada una de las opciones del anidamiento de la primera opción más el contenido de su mscBody y en los otros dos archivos pasará lo mismo con la segunda opción.

5.2.4. Cuarta iteración - Sentencia compleja *par*

En esta iteración añadimos incrementalmente la sentencia compleja denominada *par.*

Los casos de prueba seleccionados para esta iteración fueron los siguientes:

- **Secuencia de interacciones simples con una sentencia *par* de cuatro opciones entre una ellas:** en este caso de prueba se crean una serie de instancias y se introduce una sentencia *par* entre algunas interacciones síncronas. La salida tiene que ser un número de archivos igual al número de permutaciones de cuatro, que es el factorial de 4 que es 24. En cada archivo debe encontrarse una de las permutaciones posibles.
- **Secuencia de interacciones simples con una sentencia *par* de cuatro opciones con una sentencia *par* de dos opciones anidada en una de sus opciones:** La salida esperada en este caso son 48 archivos, que es el resultado de multiplicar el factorial de 4 por el factorial de 2. En general, el resultado de la anidación es un número de archivos igual a la multiplicación del factorial del número de opciones de cada sentencia *par.*

5.2.5. Quinta iteración - Sentencia compleja *loop*

En esta iteración añadimos incrementalmente la sentencia compleja denominada *loop.*

Los casos de prueba seleccionados para esta iteración fueron los siguientes:

- **Secuencia de interacciones simples con una sentencia *loop* que se ejecuta desde un mínimo de 2 veces hasta un máximo de 20:** en esta prueba se pretendía comprobar la capacidad de carga del sistema tras las mejoras en el rendimiento. La salida debe ser de 19 archivos, uno para cada número de posibles repeticiones del bucle.
- **Secuencia de interacciones simples con una sentencia *loop* que se ejecuta desde un mínimo de 1 vez hasta un máximo de 2 con una sentencia loop de las mismas características anidada:** en esta prueba comprobamos la capacidad de anidamiento de la sentencia, después de aplicar el sistema de archivos temporales. La salida debe ser de 6 archivos con todas las posibles combinaciones con el número de repeticiones del bucle interno y externo.
- **Secuencia de interacciones simples con una sentencia *loop* que se ejecuta desde un mínimo de 1 vez hasta un máximo de 2 con dos sentencias loop de las mismas características anidadas:** en esta prueba comprobamos la capacidad de anidamiento de la sentencia, después de aplicar el sistema de archivos temporales, para comprobar qué nivel de anidamiento se soporta. La salida debe ser de 42 archivos con todas las posibles combinaciones con el número de repeticiones del bucle interno y externo.

5.2.6. Última iteración - Batería final de pruebas

En esta iteración quisimos emplear secuencias de instrucciones que incluyeran ejemplos de las distintas sentencias definidas combinadas entre sí, para concluir que su comportamiento en conjunto era correcto. A continuación expondremos una de esas pruebas de la batería final y analizaremos cuál debe ser su salida.

La entrada se compone de una serie de sentencias simples que tienen entre ellas un *opt,* seguido de un *loop* con un mínimo de una vuelta y un máximo de 2 que tiene en su interior una serie de sentencias simples y un *alt* de dos opciones. Finalmente, fuera del bucle nos encontramos un *par* con dos opciones y un par de sentencias simples.

Vamos a analizar ahora el número de archivos generados:

- El *par* dará lugar a dos posibilidades, puesto que el número de permutaciones posibles es el valor del factorial de 2, que es 2.
- El *opt* dará lugar a otras dos posibilidades, puesto a que una de ellas será la aparición de su contenido y la no aparición.
- El *alt* del interior del bucle dará lugar a 2 posibilidades, una por cada opción definida.
- El bucle tendrá 2 vueltas que teniendo en cuenta que existen dos posibilidades para dos vueltas del bucle nos encontramos que el número de posibilidades es 6: 2 elevado a 1 repetición más 2 elevado a 2 repeticiones, que son el número de variaciones en cada vuelta del bucle.

Sabiendo que el número total de archivos es la multiplicación de todas estas posibilidades tenemos 24 archivos de salida. Para terminar la comprobación habrá que comprobar archivo por archivo asegurándose de que el contenido es el esperado.

6

Conclusiones

En este capítulo expondremos las conclusiones obtenidas tras el desarrollo de la herramienta. En el primer apartado ofreceremos una visión global de los conocimientos y experiencias obtenidos durante el desarrollo y de las dificultades a las que nos hemos enfrentado. En el segundo apartado enumeraremos algunas de las posibles líneas futuras de ampliación para mejorar usabilidad y capacidades de la herramienta.

6.1. Desarrollo del proyecto

Durante el desarrollo de este proyecto hemos tenido la oportunidad de aprender desde cero varías tecnologías. En primer lugar, hemos adquirido conocimientos para desarrollar nuestro propio generador de código a partir de la definición de una gramática con Xtext y Xtend. Además, hemos tenido que ser capaces de analizar e interpretar una gramática definida a alto nivel de y elegir los elementos interesantes. Con ello, hemos podido bucear en la recomendación Z.120 (ITU-T, Message Sequence Chart (MSC), 2011), lo he nos ha llevado conocerla y entenderla más profundamente.

Además de los conocimientos directamente relacionados con el desarrollo, hemos adquirido conocimientos propios de la Ingeniería del Software Dirigida por Modelos y

la parte en la que se enmarca nuestra herramienta dentro de su paradigma actual. De esta forma, hemos adquirido conocimientos sobre los lenguajes de dominio y su uso en la resolución de problemas de la Ingeniería del Software Dirigida por Modelos.

Por otro lado, hemos aprendido a enfrentarnos a las distintas dificultades que se pueden encontrar durante el desarrollo de un algoritmo de una complejidad elevada. De todas las partes que constituyen la herramienta, el algoritmo recursivo para generar todos los archivos de salida posibles fue la parte más problemática y costosa de idear y desarrollar. Había ocasiones en las que no estábamos muy seguros de ser capaces de encontrar un algoritmo que cubriera el mar de posibilidades que el proyecto pretendía abarcar. Sin embargo, tras diversos intentos y algoritmos a medias, conseguimos una implementación convincente. Lamentablemente, esta implementación tenía problemas de rendimiento y tuvimos que ponernos de nuevo manos a la obra con el objetivo de conseguir una implementación eficiente y correcta.

Una vez conseguida la implementación esperada, nos enfrentamos a una validación compleja por el gran número de casos de prueba que se podían desarrollar. En este caso decidimos hacer una batería de pruebas que cubriera los casos de error y éxito más comunes.

En conclusión, el desarrollo del algoritmo ha sido una experiencia enriquecedora que ha aportado conocimientos tanto específicos sobre tecnología como experiencia en el desarrollo de proyectos de una complejidad concreta.

6.2. Líneas futuras de ampliación

Este trabajo cuenta con un gran potencial de ampliación a partir de su punto actual.

En primer lugar, el proyecto puede ampliarse para ser capaz de contemplar más partes de la gramática de la recomendación. Podrían considerarse llamadas síncronas, composición de fragmentos de MSCs u otros muchos aspectos del estándar. Una de las principales adiciones más directas podría ser el permitir al usuario definir guardas lógicas para las distintas sentencias complejas, de forma que se pudieran establecer simulaciones del valor de las variables y a partir de ellas, obtener las distintas salidas.

En segundo lugar, el proyecto está enfocado a ser una herramienta que complemente a USE y en su punto actual es un elemento completamente independiente a la herramienta. Un punto de ampliación, podría ser crear un complemento que se incrustara en la interfaz de USE para evitar que el usuario tuviera que hacer instalaciones adicionales y que contase con toda la funcionalidad en una sola herramienta. De esta forma el usuario podría introducir los distintos MSCs y ver las distintas salidas directamente en la interfaz.

Finalmente, se podría considerar una validación más completa de los diagramas de secuencia de entrada. Actualmente, la validación que se hace de los MSCs es un poco limitada y sólo contempla casos de errores sintácticos u errores lógicos básicos, como la falta de respuesta en una llamada síncrona. Lo que se propone es comprobar que las distintas interacciones definidas son posibles de acuerdo al modelo definido en USE. De esta forma, aparte de procesar el archivo MSC también habría que procesar el USE en busca de inconsistencias con respecto a lo representado en el MSC. Por ejemplo, si un usuario especifica una secuencia de llamadas dentro de una llamada síncrona, el software deberá ser capaz de decir si esa secuencia es posible o no de acuerdo al modelo. Este proceso es costoso y difícil, pero sería el complemento perfecto para hacer de esta una herramienta imprescindible en cualquier validación.

Referencias

Abbas, N., Gravell, A. M., & Wills, G. B. (2008). Historical roots of agile methods: Where did "Agile thinking" come from? *International conference on agile processes and extreme programming in software engineering*, (págs. 94-103). Springer, Berlin, Heidelberg.

Agner, L. T., & Lethbridge, T. C. (2017). A Survey of Tool Use in Modeling Education. *Proc. of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, (págs. 303-211).

Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-Driven Software Engineering in Practice (2nd ed.)*. Morgan & Claypool Publishers.

Burgueño, L., Vallecillo, A., & Gogolla, M. (2018). Teaching UML and OCL models and their validation to software engineering students: an experience report. *Computer Science Education, 28(1)*, 23-41.

Büttner, F., & Gogolla, M. (2014). On OCL-based imperative languages. *Sci. Comput. Program. 92*, 162-178.

Database System Group Bremen University. (2007). *USE: A UML based Specification Environment.* Bremen.

Eclipse. (2019). *Eclipse Modeling Framework (EMF) - Eclipse.* Obtenido de https://www.eclipse.org/modeling/emf/

Eclipse. (2019). *Integration with EMF - Eclipse.* Obtenido de https://www.eclipse.org/Xtext/documentation/308_emf_integration.html

Eclipse-Xtext. (2019). *Writing a Code Generator with Xtend - Eclipse.* Obtenido de https://www.eclipse.org/Xtext/documentation/103_domainmodelnextsteps.html

Gogolla, M., Büttner, F., & Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program. 69(1-3)*, 27-34.

Group, O. M. (2003). *Object Constraint Language 2.4.* Obtenido de Object Management Group: https://www.omg.org/spec/OCL/2.4/PDF

Group, O. M. (2005). *UML 2.0 Infrastructure.* Obtenido de Object Management Group: https://www.omg.org/spec/UML/2.0/Infrastructure/PDF

Group, O. M. (2005). *UML 2.0 Superstructure.* Obtenido de Object Management Group: https://www.omg.org/spec/UML/2.0/Superstructure/PDF

Heap, B. R. (1936). *Permutations by interchanges.* Obtenido de https://academic.oup.com/comjnl/article/6/3/293/360213

ITU-T. (1998). *Message Sequence Chart, Annex B: Formal semantics of Message Sequence Chart.*

ITU-T. (2011). *Message Sequence Chart (MSC).* Obtenido de ITU-T Recommendation Z.120 (02/11): https://www.itu.int/rec/T-REC-Z.120

Keng, S., & Poi-Peng, L. (2006). Identifying Difficulties in Learning UML. *IS Management, 23(3)*, 43-51.

Offutt, J. (2013). Putting the Engineering into Software Engineering Education. *IEEE Software, 30(1)*, 94-96.

Schmidt, D. C. (2006). *Model-Driven Engineering.* Obtenido de http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.9720&rep=rep1&type=pdf

Selic, B. (2012). What will it take? A view on adoption of model-based methods in practice. *Software and System Modeling, 11(4)*, 513-526.

USE-OCL. (2014). *SOIL.* Obtenido de http://useocl.sourceforge.net/w/index.php/SOIL

Vallecillo, A. (2014). On the Industrial Adoption of Model Driven Engineering. Is your company ready for MDE? *International Journal of Information Systems and Software Engineering for Big Companies (IJISEBC), 1(1)*, 52-68.

Wikipedia. (2019). *File:Heap algorithm with 4 elements.svg.* Obtenido de https://en.wikipedia.org/wiki/File:Heap_algorithm_with_4_elements.svg

Xtend. (2019). *Xtend - Documentation.* Obtenido de https://www.eclipse.org/xtend/documentation/index.html

Xtext. (2019). *Xtext - Language Engineering Made Easy!* Obtenido de https://www.eclipse.org/Xtext/#intro-quotes

Apéndice A

Manual de Instalación

En las siguientes páginas enumeraremos cuáles son las herramientas que es necesario instalar para permitir el empleo de la herramienta. El desarrollo se ha realizado sobre el sistema operativo Windows y que la instalación será especificada para el mismo. En otros sistemas operativos los pasos de la instalación pueden diferir.

A.1. Java JDK

Como ya se ha expuesto en capítulos anteriores, algunas de las herramientas empleadas se apoyan sobre Java, así que es necesario tener instalado el Java Development Kit. Durante el desarrollo se ha empleado la versión 1.8.

Para descargarla, hay que dirigirse a la página oficial de descarga de Oracle bajo la siguiente dirección: https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html. En ella, el usuario deberá dirigirse a la sección correspondiente a la versión que desee descargar y seleccionar su sistema operativo.

Una vez descargado el archivo de la dirección anterior, bastará con ejecutarlo y seguir las instrucciones del instalador.

A.2. Eclipse

Para el desarrollo de este proyecto se ha utilizado la versión *Eclipse Java 2018-09*. Para efectuar la descarga basta con dirigirse a la siguiente dirección https://www.eclipse.org/downloads/packages/release. En ella, deberemos seleccionar la versión del IDE que deseemos instalar.

Una vez seleccionada, nos llevará a una página en la que nos dejará descargar la versión genérica. Nosotros buscaremos una opción que ofrezca la opción de descargar sus variantes, bajo el lema *Download Packages.* Una vez seleccionada esta última opción nos llevará a otra página con la lista de opciones y deberemos seleccionar *Eclipse IDE for Java Developers.*

Una vez ahí descargaremos el paquete y lo instalaremos siguiendo las instrucciones del instalador incluido en el paquete.
Es posible que la herramienta funcione en otras versiones del entorno siempre y cuando contemos con Java y el complemento que instalaremos en el siguiente apartado. Sin embargo, esta es la única versión en la que hemos probado que su funcionamiento es correcto.

A.3. Xtext y Xtend

Una vez instalado el entorno de desarrollo Eclipse, deberemos instalar el complemento que contiene las herramientas para el desarrollo con Xtext y Xtend. Para ello deberemos seguir estos pasos:

1. Abrir el entorno de Eclipse y en la barra superior seleccionar la opción *Help*. En el menú desplegable seleccionaremos la opción *Install New Software...* y se abrirá una nueva ventana.

2. En esta nueva ventana deberemos introducir la siguiente dirección en el campo *Work With:* https://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/. Pulsamos *Add...* y pulsamos aceptar en la ventana emergente que nos aparecerá.

3. En la ventana principal escribimos en el filtro 2.18.0 que son las versiones empleadas en el desarrollo. Seleccionamos el Xtend IDE y el Xtext Complete SDK de dicha versión, pulsamos *Next* y aceptamos en todas las ventanas siguientes.

En caso de que no aparezca la versión es posible que esté seleccionada la opción *Show only the latest version of available software.* Una vez deseleccionada debería aparecer la versión que buscamos.

4. Una vez instalado, Eclipse se reiniciará para completar la instalación.

A.4. Nuestra Herramienta

Una vez se haya instalado todo el software anterior, deberemos abrir Eclipse y seguir los siguientes pasos para instalar la herramienta.

1. Seleccionar la opción *File* en el menú superior, y en el menú desplegable seleccionar *New -> Project.*

2. En la ventana emergente deberemos seleccionar Xtext -> Xtext Project. Pulsaremos *Next*. En la siguiente ventana deberemos rellenar los campos con la siguiente información:

- Project Name: z120ToSoil
- Name: z120ToSoil.Z120
- Extensions: z120

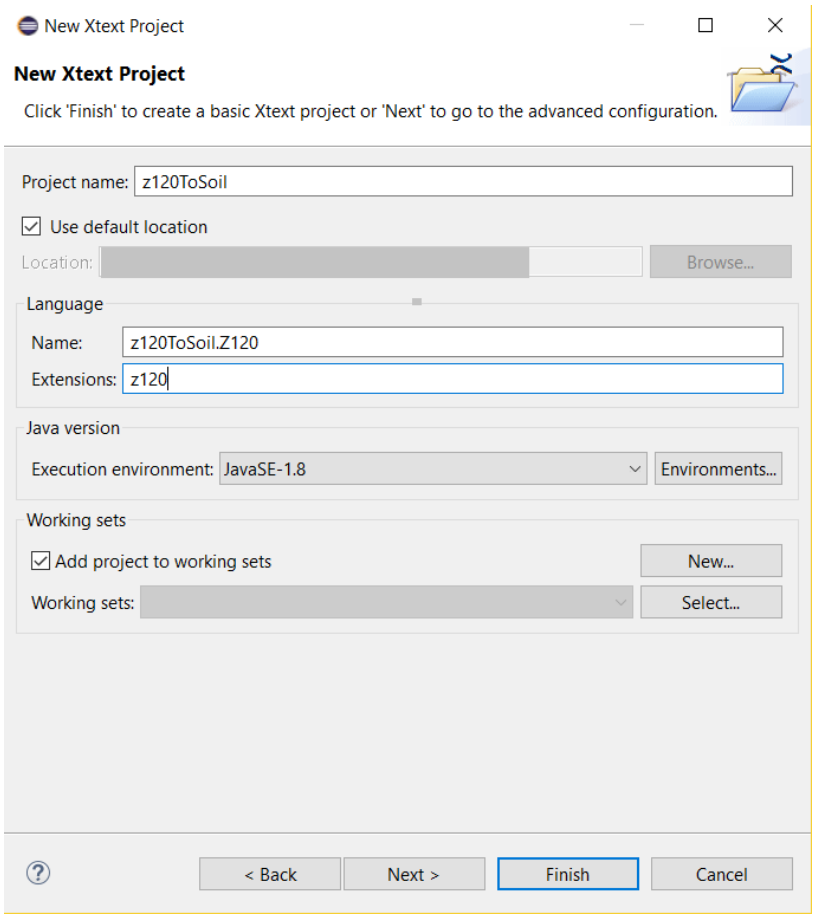


Figura A.4.1. Campos de configuración rellenados

Una vez rellenados los campos seleccionaremos *Finish* y el proyecto habrá sido creado. Se habrán generado diversas carpetas en el proyecto, y deberemos sustituir el contenido de varias de ellas por las carpetas homónimas proporcionadas.

3. Sustituimos el contenido del archivo Z120.xtext, que incluye la gramática. Este se encuentra en esta localización z120ToSoil/src/z120ToSoil/Z120.xtext.

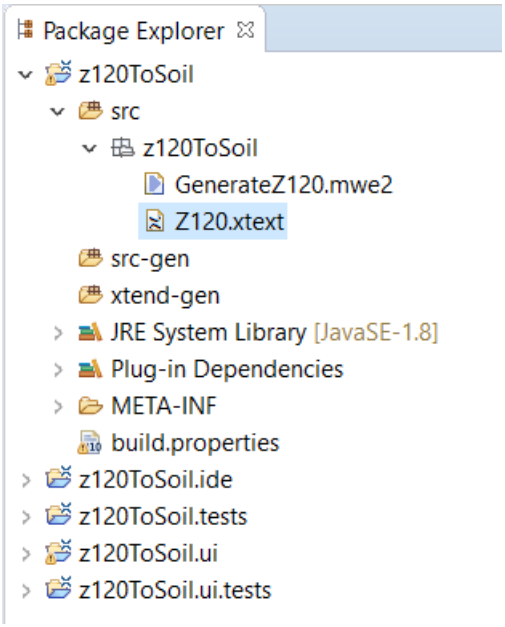
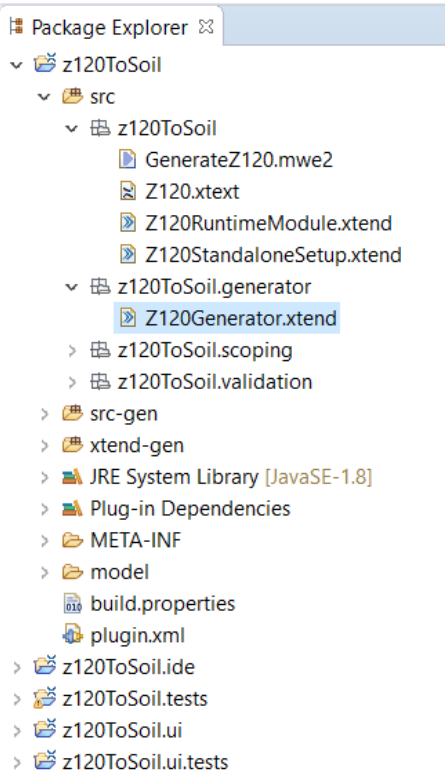


Figura A.4.2. Localización del archivo xtext.

4. Una vez sustituido el contenido de este archivo, deberemos hacer click derecho sobre la él y seleccionar *Run As... -> Generate Xtext Artifacts.* Esta acción generará distintos archivos y carpetas.

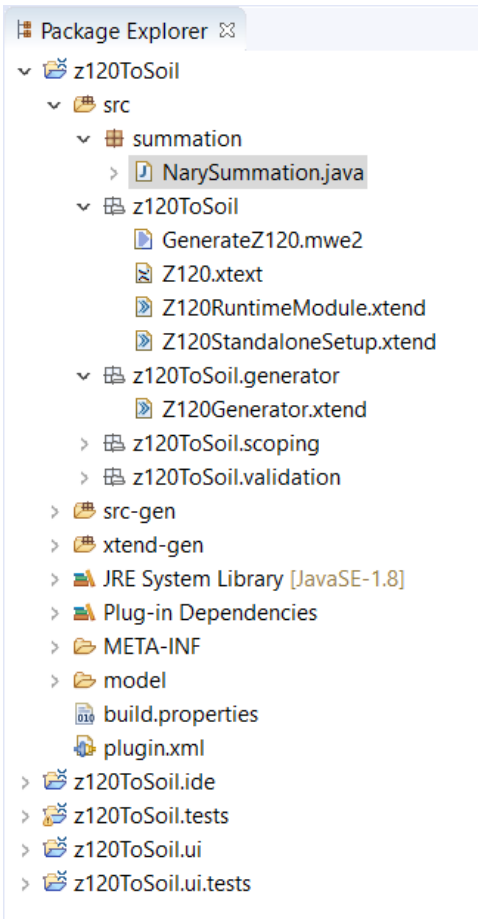
5. Sustituimos el contenido del archivo Z120Generator.xtend. Este archivo se encuentra en la siguiente localización: z120ToSoil/src/z120ToSoil/z120ToSoilGenerator.xtend



A.4.3. Estructura tras la generación de los artefactos y localización del archivo xtend.

6. Dentro de la carpeta z120ToSoil/src/ crearemos un *package* para contener una clase auxiliar. Seleccionamos la carpera src, click derecho *New -> Package* y lo denominamos *summation*.

7. Dentro crearemos un archivo denominado *NarySummation.java* y sustituiremos su contenido por el del archivo homónimo. Para crear el archivo hacemos click derecho sobre la *package* que hemos creado en el paso anterior, seleccionamos *New -> Class.* la denominamos NarySummation y pulsamos aceptar. Sustituimos su contenido por el del archivo proporcionado.



A.4.4. Estructura final del proyecto

De esta forma la instalación estaría completada y la herramienta lista para usarse.

Apéndice B

Manual de usuario

En las siguientes páginas explicaremos los pasos básicos para emplear la herramienta una vez se ha finalizado la instalación. Los pasos a seguir para emplear la funcionalidad principal son los siguientes:

1. Para abrir un segundo entorno Eclipse que tenga cargadas las reglas de la gramática impuestas, el usuario deberá seleccionar la primera carpeta asignada al proyecto, hacer click derecho, seleccionar la opción *Run As -> Eclipse Application.*

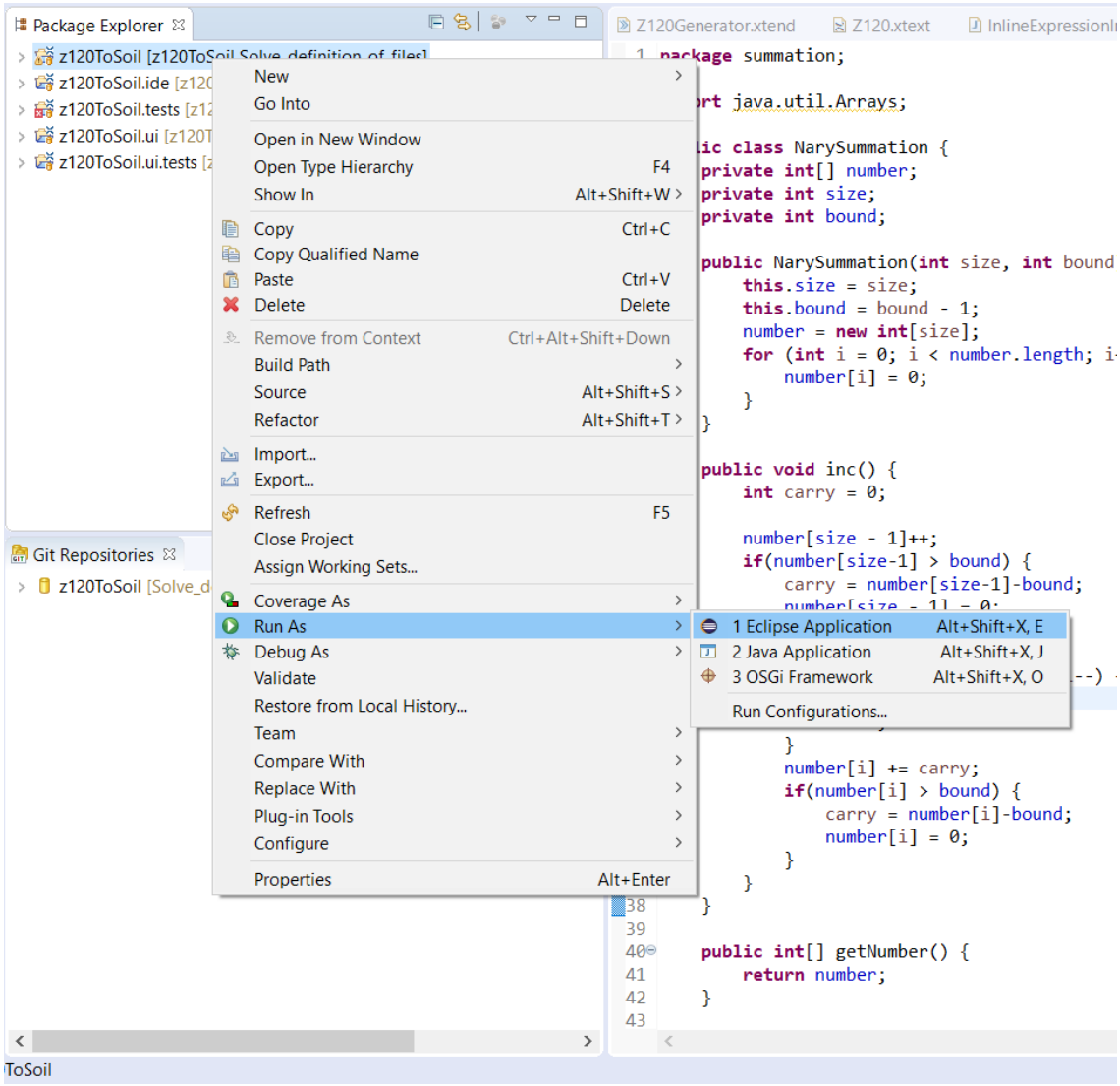


Figura A.B.1. Método para ejecutar el entorno desarrollado.

2. Una vez seleccionado, se abrirá un nuevo entorno Eclipse. En este entorno, el usuario deberá crear un proyecto seleccionando la opción *File -> New -> Project.* Introducimos el nombre del proyecto y pulsamos aceptar.

3. Haciendo click derecho sobre el proyecto creado, deberemos seleccionar la opción *Properties.*

4. Se abre una ventana de diálogo adicional y deberemos seleccionar en el cuadro derecho la opción *Project Natures.*

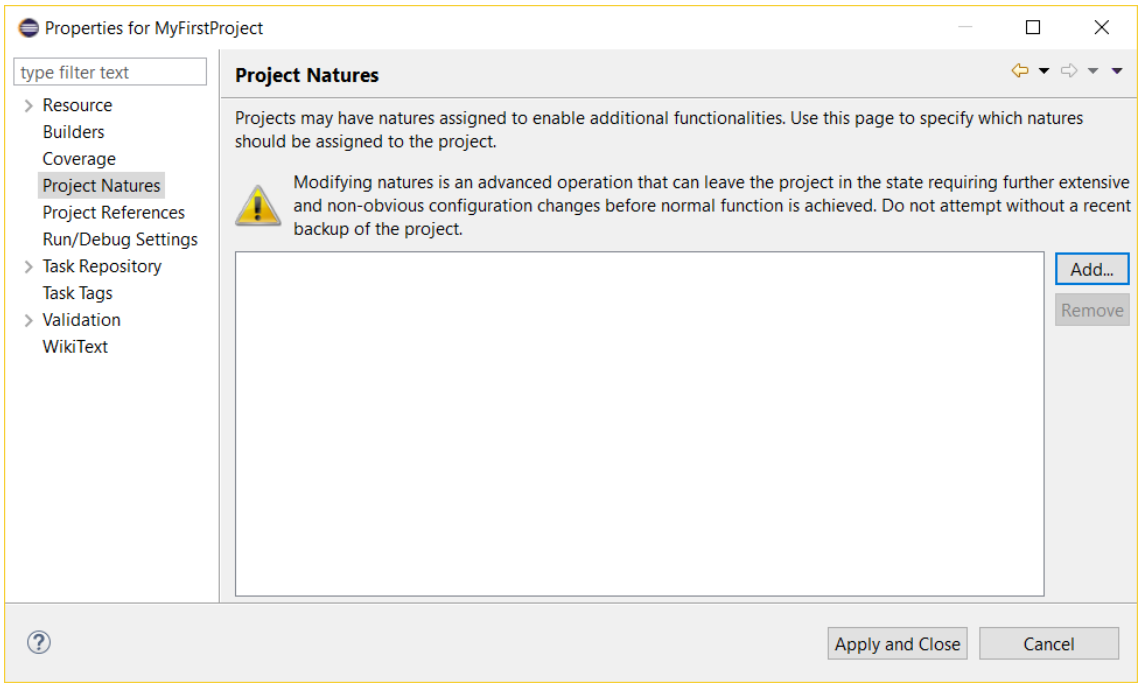
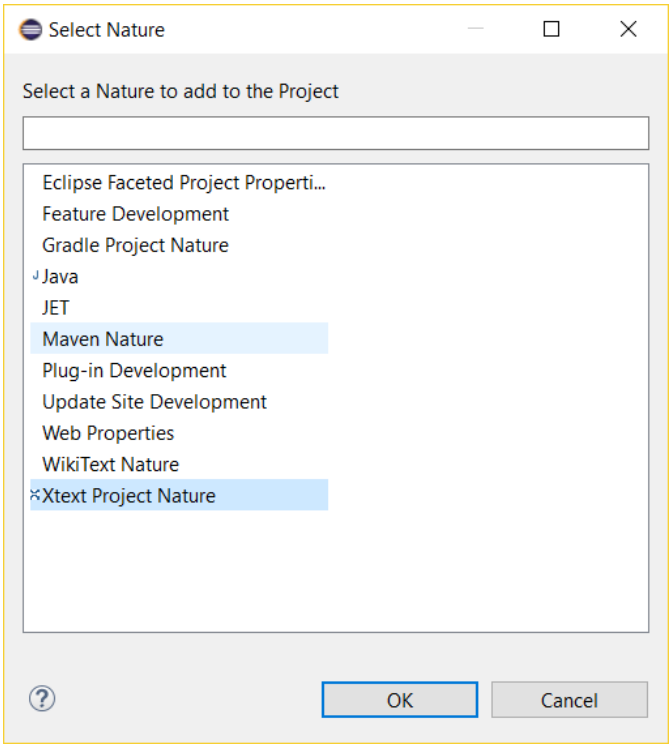


Figura A.B.2. Selección de propiedades del proyecto.

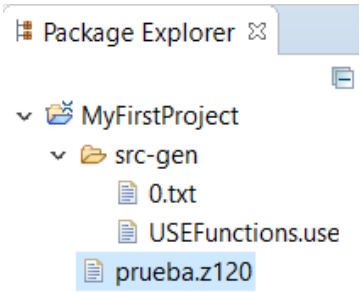
5. En esa sección seleccionamos *Add* y en la ventana emergente seleccionaremos *Xtext Project Nature.* Pulsamos aceptar.



A.B.3. Selección de naturaleza del proyecto.

6. Finalmente, crearemos un archivo vacío en el interior del proyecto y le pondremos la extensión z120. Para crearlo haremos click derecho en el proyecto y seleccionaremos *New -> File*.

7. Para activar la funcionalidad, basta con escribir el MSC en el interior del archivo con la extensión z120 que acabamos de crear. Cada vez que guardemos el archivo, se generarán todos los archivos resultantes en una carpeta denominada *src-gen* en el interior de nuestro proyecto.



A.B.4. Estructura final del proyecto generado

Apéndice C

Definición de la gramática MSC

En este apéndice tenemos la gramática completa desarrollada para el proyecto a partir de una selección de la recomendación Z.120 de la ITU-T (ITU-T, Message Sequence Chart (MSC), 2011).

```
grammar z120ToSoil.Z120 with org.eclipse.xtext.common.Terminals

generate z120 "http://www.Z120.z120ToSoil"

MessageSequenceChart:
  'msc' Msc 'endmsc;'
;

FuncName:
        ID+ ('(' ( ID| ',')* ')')?
;

Msc:
        mscHead=MscHead
        mscBody=MscBody
;

MscHead:
        mscName=ID
        // En la gramatica se define opcional
```

```
        mscParameterDecl=InstanceParmDeclList
;

InstanceParmDeclList:
        '(inst' (instanceDecl+=InstanceDecl) (',' instanceDecl+=InstanceDecl)* ')'
;

InstanceDecl:
        instanceParmName=ID ':' instanceParmKind=ID
;

MscBody:
        (eventDefinition += EventDefinition)*
;

InstanceEvent:
        instanceName=ID ':' instanceEvent=CallEvent ';'
;

MultiInstanceEvent:
        instanceNameList+=ID (',' instanceNameList+=ID)* ':'
multiInstanceEvent=InlineExpression ';'
;

EventDefinition:
        InstanceEvent | MultiInstanceEvent
;

CallEvent:
        CallOut | CallIn | ReplyOut | ReplyIn
;

CallOut:
        'call' msgIdentification=FuncName 'to' address=ID
;

CallIn:
```

```
        'receive' msgIdentification=FuncName 'from' address=ID
;

ReplyOut:
        'replyout' msgIdentification=FuncName 'to' address=ID
;

ReplyIn:
        'replyin' msgIdentification=FuncName 'from' address=ID
;

InlineExpression:
        LoopExpr | OptExpr | AltExpr | ParExpr
;

LoopExpr:
        'loop'  '<'low_boundary=INT','high_boundary=INT'>'  'begin'  (identification=ID)?  ';'
mscBody+=MscBody 'loop' 'end'
;

OptExpr:
        'opt' 'begin' (identification=ID)? ';' mscBody+=MscBody 'opt' 'end'
;

AltExpr:
        'alt' 'begin' (identification=ID)? ';' mscBody+=MscBody
        ('alt;' mscBody+=MscBody)*
        'alt' 'end'
;

ParExpr:
        'par' 'begin' (identification=ID)? ';' mscBody+=MscBody
        ('par;' mscBody+=MscBody)*
        'par' 'end'
;
```